Programming Languages
J. J. Horning Editor

Proving the Correctness of Heuristically Optimized Code

Hanan Samet
University of Maryland, College Park

A system for proving that programs written in a high level language are correctly translated to a low level language is described. A primary use of the system is as a postoptimization step in code generation. The low level language programs need not be generated by a compiler and in fact could be hand coded. Examples of the usefulness of such a system are given. Some interesting results are the ability to handle programs that implement recursion by bypassing the start of the program, and the detection and pinpointing of a wide class of errors in the low level language programs. The examples demonstrate that optimization of the genre of this paper can result in substantially faster operation and the saving of memory in terms of program and stack sizes.

Key Words and Phrases: compilers, correctness, code optimization, debugging, program verification, Lisp

CR Categories: 4.12, 4.21, 4.22, 5.24

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.

Author's address: Computer Science Department, University of Maryland, College Park, MD 20742.

1. Introduction

In [23], a formalism is presented for proving that programs written in a high level language are correctly translated to assembly language. The prime motivation for the work is a desire to prove that optimizations performed during the translation process are correct. In particular, these optimizations are often of a heuristic, nonrepeatable nature—e.g. the type of improvements that could be performed by an individual while peering over an object program. The unpredictability of the behavior of such individuals poses the requirement that the proof procedure must be independent of the intermediary mechanism which transforms the source program into the object program (e.g. a compiler or any hand-coding procedure).

This previous requirement means that the notion of correctness must be carefully defined and that capabilities of such a proof must be precisely identified. This in turn leads to a need for a representation which reflects both the high- and low level programs. In order to be able to find and make use of such a representation, critical semantic properties of the high level language in question must be identified as well as their interrelationship to the instruction set of the computer executing the object programs.

In this paper we present an overview of the proof system reported in [23]. First, we state the relationship of our goals to previous work. This is followed by a scenario of a typical heuristic optimization process which demonstrates the capabilities and limitations of the proof system. Next, we present the intermediate representation that we have chosen, the manner in which it is obtained, and the techniques used in a proof. Finally, we briefly discuss the implementation status of the system.

2. Relation to Other Work

We are interested in proving that programs are correctly translated. A similar problem that has been receiving much attention in the past few years is that of proving programs correct [16]. Most of the attempts have been along the lines of assertions [8, 11] about the intent of the program which are then proved to hold. The difficulties with such methods are numerous. The most notable problems are encountered in specifying the assertions [7, 27] and the actual proof method. Proofs using such methods reduce to showing that a set of assertions hold. However, no allowance is made for the possibility that the assertions might be inadequate to specify all of the effects of the program in question. Thus we are led to a belief that the concept of intent is too imprecise for proving correctness of compilation whereas it is justifiable in proving equivalence between algorithms.

In the case of computer programs written in a higher level language we are primarily interested in the correctness of the translation. In this case, there is no need for

any knowledge about the purpose of the program to be translated. For example, there exist a number of different algorithms for sorting (e.g. quicksort, shellsort, etc.). In order to prove the equivalence of two of these algorithms we must resort to demonstrating that they both possess a common input-output pair characterization. Thus a conventional proof attempts to show that the algorithms yield identical results for all possible inputs. The problem of proving the equivalence of different algorithms is known to be generally unsolvable by use of halting problemlike arguments.

In order to avoid the unsolvability problem we must be more precise in our definition of equivalence. By equivalence we mean that the two programs must be capable of being proved to be structurally equivalent [14], that is, they have identical execution sequences (e.g. they must test the same conditions) except for certain valid rearrangements of computations. Such rearrangements include transformations classified as "low level" optimizations [17]. However, more ambitious transformations classified as "source level" [5, 9, 29] are precluded. Note also that our criterion of equivalence is a more stringent requirement than that posed by the conventional definition of equivalence 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. In our process of demonstrating equivalence no use is made of the purpose of the program.

Notice that we prove the correctness of the translation. One method of achieving this is to prove that the translator (for example a compiler) is correct—e.g. to prove that there does not exist a program which is incorrectly translated by the compiler. In this case we would revert to the intent characterization of correctness set forth previously. Instead, we prove for each program input to the translation process, that the translated version is equivalent to the original program. Thus, we are not making any claims with respect to the general correctness of the translation process. A proof must be generated for each input to the translation process. However, this has several important advantages, especially when the translator is a compiler. First, as long as the compiler does its job for each program put into it, its correctness is of a secondary nature—that is, we can attribute an effective correctness to the compiler. Second, the proof process is independent of the compiler. This means that if another compiler were used no difference would result, thereby implying that programs could be compiled by hand or mechanically translated. This is quite important and identifies the proof as belonging to the semantics of the high- and low level languages in which the input and output, respectively, are expressed rather than belonging to the translation process. Third, any proof method that would prove a compiler correct would be limited with respect to the types of optimizations that it could allow. This is because such a method would rely on the identification of all the possible optimizing transformations. This is the type of approach taken in the proof of the correctness of LCOM0 and LCOM4 [15]. In contrast, we are interested in a heuristic code generation procedure which often relies on a "hypothesis and test" [20] approach. This is illustrated in the next section.

3. Example

The main motivation for this work has been a desire to write an optimizing compiler. In order to achieve this goal, it was determined that a correctness proof of the optimization process was necessary. In particular, we are interested in proving that small perturbations in the code leave the effect of the function unchanged. In this section we present a scenario of a typical optimization process to which our work is addressed. Some of the optimizations that we examine include common subexpression elimination, changing calling sequence conventions, elimination of recursion by iteration, and bypassing the start of the program when recursion is in order. The primary goal of the optimizations is to reduce the amount of work necessary to set up linkages between functions and to use as much local information as possible—for example, values of conditions tested, contents of accumulators, etc. Such techniques, despite seeming "less than earth shattering" when viewed individually, yield significant reductions in time and space when viewed collectively. The purpose of the examples in this section is to give the reader a feeling for the concepts which must be handled by a proof procedure and for the power of the results that can be achieved despite our rather narrow definition of equivalence. We use Lisp [18] as the high level language and Lap [21] (a variant of the PDP-10 [6] assembly language) as the object language.

As an example, consider the function REVERSE which takes as its argument a list L and returns as its result a pointer to a copy of the list where all the links have been reversed. For example, application of the function to the list (ABC) results in the list (CBA). A formulation of this function in a dialect of Lisp known as Mlisp [26] (i.e. meta Lisp) is given in Figure 1. This formulation of the function will be referred to as Algorithm 1.

In order to be able to examine some low level language programs, we must have an execution-level definition of the high level language. In our case, such a definition enables us to make sense of the following Lap programs in the context of a Lisp environment. Briefly, each PDP-10 word is 36 bits wide and can be partitioned into two 18 bit halves. 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 the atom NIL. The PDP-10 has a hardware stack and functions return via a return address which has been placed on the stack by the invoking function. A Lap program expects to find its parameters in the accumulators (on the PDP-10 all accumulators are general purpose regis-

Fig. 1. Algorithm 1 for REVERSE.

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

Fig. 2. Lisp 1.6 Compiler-generated encoding of Figure 1.

```
PC1  (PUSH 12 1)            save L on the stack
     (JUMPE 1 TAG1)         jump to TAG1 if L is NIL
     (HRRZ@ 1 0 12)         load accumulator 1 with CDR(L)
     (CALL 1 (E REVERSE))   compute REVERSE(CDR(L))
     (PUSH 12 1)            save REVERSE(CDR(L)) on the stack
     (HLRZ@ 1 -1 12)        load accumulator 1 with CAR(L)
     (CALL 1 (E NCONS))     compute CONS(CAR(L),NIL)
PC8  (MOVE 2 1)             load accumulator 2 with CONS(CAR(L),NIL)
     (POP 12 1)             load accumulator 1 with REVERSE(CDR(L)) from the stack
     (CALL 2 (E *APPEND))   compute *APPEND(REVERSE(CDR(L)),CONS(CAR(L),NIL))
TAG1 (SUB 12 (C 0 0 1 1))   undo the first push operation
     (POPJ 12)              return
```

Fig. 3. Result of optimizing Figure 2.

```
     (SKIPN 2 1)            load accumulator 2 with L and skip if not NIL
     (POPJ 12)              return NIL
PC3  (HLRZ 1 0 1)           load accumulator 1 with CAR(L)
     (CALL 1 (E NCONS))     compute CONS(CAR(L),NIL)
     (PUSH 12 1)            save CONS(CAR(L),NIL) on the stack
     (HRRZ 1 0 2)           load accumulator 1 with CDR(L)
     (CALL 1 (E REVERSE))   compute REVERSE(CDR(L))
     (POP 12 2)             load accumulator 2 with CONS(CAR(L),NIL) from the stack
     (JCALL 2 (E *APPEND))  compute *APPEND(REVERSE(CDR(L)),CONS(CAR(L),NIL))
```

Fig. 4. Algorithm 2 for REVERSE.

```
REVERSE(L) = REVERS1(NIL,L)
REVERS1(RL,L) = if NULL(L) then RL
                else REVERS1(CONS(CAR(L),RL),CDR(L))
```

Fig. 5. Lisp 1.6 compiler generated encoding for Figure 4.

```
PC1  (PUSH 12 1)            save RL on the stack
PC2  (PUSH 12 2)            save L on the stack
PC3  (JUMPN 2 TAG2)         jump to TAG2 if L is not NIL
PC4  (JRST 0 TAG1)          jump to TAG1
TAG2 (MOVE 2 -1 12)         load accumulator 2 with RL
PC6  (HLRZ@ 1 0 12)         load accumulator 1 with CAR(L)
     (CALL 2 (E CONS))      compute CONS(CAR(L),RL)
     (HRRZ@ 2 0 12)         load accumulator 2 with CDR(L)
PC9  (CALL 2 (E REVERS1))   compute REVERS1(CONS(CAR(L),RL),CDR(L))
TAG1 (SUB 12 (C 0 0 2 2))   undo the first two push operations
PC11 (POPJ 12)              return
```

ters and can be used for indexing) and also returns its result in accumulator 1. In the case of REVERSE, parameter L is 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. A program is entered at its first instruction and a return address is situated in the top entry of a stack whose pointer is in accumulator 12. Whenever recursion or a function call to an external function (via the CALL or JCALL mechanism) occurs, the contents of all the accumulators are assumed to have been destroyed unless otherwise known. Exceptions include CONS and XCONS (XCONS(A,B) = CONS(B,A)), and NCONS (NCONS(A) = CONS(A,NIL)) which are known to leave unchanged all accumulators other than those containing the arguments. In other words, CONS and XCONS only affect accumulators 1 and 2 while NCONS only affects accumulator 1. It should be clear that in any case certain accumulators used by the Lisp system such as 12 (i.e. the stack pointer), the free storage list, etc. are not assumed to have changed. This is not a problem since the proof system is aware of what accumulators a user may read and overwrite and likewise for locations on the stack (that is, all locations above the return address).

The Lisp 1.6 [21] compiler generates the Lap code given in Figure 2 for this function. 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. The AC and INDEX fields contain numbers between 0 and decimal 15. ADDR denotes the address field. A list of the form (C 0 0 *num*1 *num*2) appearing in the address field of an instruction is interpreted as an address of a word containing *num*1 and *num*2 in its left and right halves, respectively (assuming *num*1 and *num*2 are less than or equal to 0.15). The meanings of the instructions should be clear from the adjoining comments. Nevertheless, the appendix contains verbal descriptions of all of the instructions used in our examples.

There are a number of unnecessary operations in this encoding. First, there is no need to save L on the stack at PC1 prior to determining if it is NIL. Second, we may rearrange the order of computing the arguments to *APPEND thereby taking advantage of the fact that NCONS leaves the contents of all accumulators besides 1 un-

Fig. 6. Result of optimizing Figure 5.

```
PC1  (JUMPE 2 TAG1)          jump to TAG1 if L is NIL
REV  (PUSH 12 2)             save L on the stack
     (HLRZ 2 0 2)            load accumulator 2 with CAR(L)
PC4  (CALL 2 (E XCONS))      compute CONS(CAR(L),RL)
PC5  (HRRZ@ 2 0 12)          load accumulator 2 with CDR(L)
PC6  (SUB 12 (C 0 0 1 1))    undo the first push operation
PC7  (JUMPN 2 REV)           if CDR(L) is not NIL then compute
                             REVERS1(CONS(CAR(L),RL),CDR(L))
TAG1 (POPJ 12)               return
```

Fig. 7. Result of optimizing Figure 6.

```
     (JUMPE 2 TAG1)          jump to TAG1 if L is NIL
PC2  (PUSH 12 2)             save L on the stack
REV  (HLRZ 2 0 2)            load accumulator 2 with CAR(L)
     (CALL 2 (E XCONS))      compute CONS(CAR(L),RL)
PC5  (HRRZS@ 2 0 12)         load accumulator 2 and the top of the stack with CDR(L)
PC6  (JUMPN 2 REV)           if CDR(L) is not NIL then compute
                             REVERS1(CONS(CAR(L),RL),CDR(L))
     (SUB 12 (C 0 0 1 1))    adjust the stack pointer
TAG1 (POPJ 12)               return
```

Fig. 8. Optimal Lap encoding corresponding to Figure 4.

```
     (SKIPN 3 2)             load accumulator 3 with L and skip if not NIL
     (POPJ 12)               return NIL
REV  (HLRZ 2 0 3)            load accumulator 2 with CAR(L)
     (CALL 2 (E XCONS))      compute CONS(CAR(L),RL)
     (HRRZ 3 0 3)            load accumulator 3 with CDR(L)
     (JUMPN 3 REV)           if CDR(L) is not NIL then compute
                             REVERS1(CONS(CAR(L),RL),CDR(L))
     (POPJ 12)               return
```

changed. This means that L need not be saved on the stack. Therefore, we save it in accumulator 2 while at the same time a test is performed to determine if it is NIL. Third, we observe that since we no longer save any variables on the stack, there is no need to invoke *APPEND recursively. Instead, a JCALL (the unconditional jump equivalent of CALL) is used. Figure 3 contains the result of these optimizations.

The above optimizations have resulted in the reduction of the lengths of the inner loop and the overall program from 12 and 12 to 8 and 9, respectively. This is about as good as encoding as we can get for this formulation of the REVERSE algorithm because six operations are required for each iteration. These operations are the computation of CAR, CDR, CONS, *APPEND, recursion, and the testing of the nullness of L. In addition, we must temporarily save and restore the value of one of the arguments to *APPEND while computing the other one. Thus the length of the inner loop cannot be reduced further without changing the algorithm. Such an alternate formulation, referred to as Algorithm 2, is given in Figure 4.

Algorithm 2 makes use of an auxiliary function REVERS1. This function has an additional variable which serves to accumulate the result as the algorithm is applied. It is clear that Algorithm 2 is more efficient than Algorithm 1 by noting that no *APPEND operations need to be performed. Moreover, the algorithm is iterative in the sense that the call to REVERS1 is the final step of the algorithm. The Lap encoding produced by the Lisp 1.6 compiler is shown in Figure 5.

This encoding abounds with unnecessary operations. Thus we carry out the following sequence of optimization steps to obtain the encoding in Figure 6. There is no need to save RL on the stack at location PC1 since accumulator 1 is never stored into prior to the last reference to it. By making use of an XCONS operation there is no longer any need to load accumulator 2 with RL at TAG2. This implies that accumulator 2 must be loaded with CAR(L) at location PC6. The pair of branch instructions at locations PC3 and PC4 can be placed before saving L at PC2. In fact, a more optimal move is to simply replace them by a JUMPE to PC11. Recursion at PC9 may be replaced by iteration provided that the stack pointer is adjusted prior to the jump. Further reduction in execution time can be achieved by observing that instead of an unconditional jump at PC9 to a conditional jump prior to PC2, we may perform a conditional jump at PC9 with the sense of the test reversed. This will be referred to as *loop shortcutting.* (See [28] for a similar idea.) To a Sail [22] programmer this concept is somewhat analogous to the similarity between a FOR loop and a DO UNTIL loop. Thus Figure 6 contains a jump to location REV from PC7 rather than to the start of the program.

The encoding in Figure 6 contains a PUSH operation at location REV which has the effect of recycling the stack location released at location PC6. The length of the inner loop could be decreased by two instructions if we would place the value of CDR(L) on the stack as well as in accumulator 2. Thus the PUSH and the stack adjustment operations at locations REV and PC6, respectively, could be moved out of the inner loop. Figure 7 shows one possible way to achieve this effect by use of a HRRZS instruction. This instruction forms a word containing zero and the right half of the contents of the effective address and stores it in both the accumulator specified by the accumulator field and the location addressed by

the effective address. Such an optimization has the interesting effect of modifying the calling sequence from one where the arguments are in accumulators 1 and 2 when there is an external invocation of the function to one where one of the arguments is now also on the top of the stack when the function is invoked internally (that is, recursion via PC6).

At a first glance it would seem that we have succeeded in reducing the length of the inner loop to four instructions. However, use of the proof system reveals that we have erred. Unfortunately, the HRRZS@ instruction at PC5 places its result back in the location designated by the effective address. Thus instead of CDR(L) being placed on the top of the stack we have succeeded in changing the right half of the location pointed at by L to be 0. That is, CAR(L) becomes 0 or NIL. The left half of the location pointed at by L remains the same as does the top of the stack which still contains a pointer to L. Note that accumulator 2 has been loaded with 0 and CDR(L) in the left and right halves, respectively. In Section 4.2 we shed some light on how the error was detected.

Further reflection on the encoding in Figure 6 reveals that the only reason for the PUSH and stack adjustment operations at locations REV and PC6 is the external function call to XCONS at PC4. However, XCONS only destroys accumulators 1 and 2. Thus instead of saving L on the stack at REV, we may store it in an accumulator, say 3. This renders the operations at REV and PC6 unnecessary. In fact, we merely need to initialize accumulator 3 with L and iterate with CDR(L) in accumulator 3. Thus we are making use of a different calling sequence for internal recursion since invocation of the function from outside finds L in accumulator 2 while internal invocation finds L in accumulator 3. The new encoding is given in Figure 8. The change in the calling sequence poses no problem for the proof system because whenever loop shortcutting is performed we must make sure that all locations that will be subsequently referenced are set to their proper values. This is one of the tasks of the symbolic interpretation process discussed in Section 4.2.

Observe that the length of the original encoding has been reduced from 11 to 7 instructions. More importantly, the length of the inner loop has decreased from 10 to 4 instructions. The new encoding can be considered optimal for the following reason. Algorithm 2 requires five operations; CAR, CDR, CONS, the testing of the nullness of L, and the iteration step. At times a test may be combined with another nontest operation. We have only one test operation. Therefore, the minimal number of instructions with which we could accomplish our desired computation is four and since we were able to encode the function with four instructions we have achieved the lower bound.

The above examples serve to indicate the type of optimization we wish to be able to prove correct. Although these encodings were a result of a hand optimization procedure, we feel that in the future such optimizations could be achieved by a so-called postoptimizing program. The examples showed that there is a limit to the number of optimizations that can be performed before a change must be made to the algorithm. Specifically, this was seen in the transition from Algorithm 1 to Algorithm 2. We cannot prove the equivalence of the two algorithms by using our techniques. Such work has a greater payoff when done at the source level. In fact, systems such as that reported in [4] are expressly designed to deal with these issues.

4. Representation

The examples of Section 3 serve to demonstrate that any proof system that is chosen must abandon any notion of the existence of a unique relationship between the source code and the object code. We have seen that program translation is a many-to-many-process—that there is no one-to-one relationship between source code and object code. Therefore, there is no reflection of the source-level syntax in the object code and thus we find little use for decompilation [10] techniques—such methods attempt to reconstruct a high level program from the object code. Instead we use an intermediate representation of the program referred to as the *normal form* which reflects all of the computations and decisions that are performed. In addition, this representation reflects an ordering based on the relative times at which the various computations are executed.

The proof system consists of the following phases. The original high level language program is converted to the intermediate representation by use of a set of transformations. Similarly, the low level language program is converted to the intermediate representation by means of a process known as *symbolic interpretation.* This process entails the interpretation of a procedure for each instruction in the object program along each possible execution path. These procedures have the effect of updating a model of the computation which reflects the contents of relevant locations, conditions tested, and computations performed. Next, an attempt is made to prove that the two intermediate forms can be transformed into each other. During the proof procedure inequivalence may be detected and the sources of error can often be pinpointed.

In the remainder of the paper we will give an overview of how the above notions are used in a proof system. However, in order to have some framework for the discussion we must assume the existence of a suitable high level language, a low level language, and an execution-level definition. Our high level language is a subset of Lisp and our object language is Lap.

Briefly, we are dealing with a subset of Lisp that allows side effects and global variables. There are two restrictions. First, a function may only access the values of global variables or the values of its own local variables—it may not access another function's local varia-

Fig. 9. Intermediate representation corresponding to Figure 1.

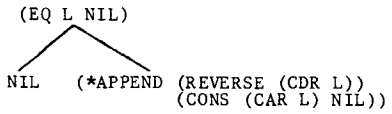
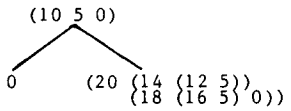


Fig. 10. Numeric intermediate representation corresponding to Figure 1.



bles. Second, the target label of a GO in a PROG must not have occurred physically prior to the occurrence of the GO to the label. For further discussion of this restriction see Section 6.

4.1 Intermediate Representation

The intermediate representation has its root in the work done by McCarthy [19] in showing the existence of a canonical form for the theory of conditional expressions and its use in proving equivalence. This theory corresponds to analysis by cases in mathematics and is basically a generalization of propositional calculus. As an example, see Figure 9 which is a two-dimensional realization of the intermediate representation corresponding to Figure 1.

The basic entity of the intermediate representation is a generalized boolean form (gbf) which can be visualized as a tree, and has the form $(p \rightarrow x, y)$ where p, x, and y are variables or gbf's and are known as the premise, conclusion, and alternative, respectively. p takes the value of T (for true), NIL (for false), or undefined in which case the gbf takes the value x, y, or undefined, respectively. Two gbf's are said to be strongly equivalent (denoted by $=$) if they have the same values for all values of their constituent variables whereas they are weakly equivalent (denoted by $=_w$) if they have the same values only when all of their constituent variables are defined. Thus in the case of weak equivalence we disregard cases where the premises are undefined. Equivalence can be tested by the method of truth tables or by use of the following axioms to transform any gbf into an equivalent one.

$$(p \rightarrow a, a) = {}_w a \quad (1)$$
$$(\mathrm{T} \rightarrow a, b) = a \quad (2)$$
$$(\mathrm{NIL} \rightarrow a, b) = b \quad (3)$$
$$(p \rightarrow \mathrm{T}, \mathrm{NIL}) = p \quad (4)$$
$$(p \rightarrow (p \rightarrow a, b), c) = (p \rightarrow a, c) \quad (5)$$
$$(p \rightarrow a, (p \rightarrow b, c)) = (p \rightarrow a, c) \quad (6)$$
$$((p \rightarrow q, r) \rightarrow a, b) = (p \rightarrow (q \rightarrow a, b), (r \rightarrow a, b)) \quad (7)$$
$$(p \rightarrow (q \rightarrow a, b), (q \rightarrow c, d)) = (q \rightarrow (p \rightarrow a, c), (p \rightarrow b, d)) \quad (8)$$

The above axioms can be used to transform any gbf into a normal form which is a binary tree whose nonterminal nodes correspond to variables taking on values of T or NIL and whose terminal nodes represent general valued variables. There is a normal form algorithm for both weak and strong equivalence—the difference being that during the provess of obtaining the normal form for strong equivalence axiom (1) can not be used at will. It can only be used when its premise variable is defined.

In order for the above ideas to be useful in proving the correctness of translation of Lisp programs we must show how they are adapted to include the constructs present in Lisp programs. We are primarily interested in proving strong equivalence and in the more general notion of functions rather than variables.

For example, the relation of functions to gbf's is given by the distributive law:

$$f(x_1, \ldots, x_{i-1}, (p \rightarrow q, r), x_{i+1}, \ldots, x_n) = (p \rightarrow f(x_1, \ldots, x_{i-1}, q, x_{i+1}, \ldots, x_n), f(x_1, \ldots, x_{i-1}, r, x_{i+1}, \ldots, x_n))$$

A COND is normally of the form (COND $(p_1\ e_1)$ $(p_2\ e_2)$ ... $(p_n\ e_n)$) and it is mapped into $(p_1 \rightarrow e_1, (p_2 \rightarrow e_2, \ldots (p_n \rightarrow e_n, \mathrm{NIL}) \ldots))$. We generalize this form to enable a sequence of computations to be specified in case a condition is true. This is achieved by the introduction of the function FL, defined to be a function of one or more arguments, which returns, as its result, the value of its final argument. For example, in the above COND if it was desired to perform the computations e_{21} and e_{22} in case p_2 is true, then we would have the following mapping:

$$(\mathrm{COND}\ (p_1\ e_1)(p_2\ e_{21}\ e_{22}) \ldots (p_n\ e_n)) = (p_1 \rightarrow e_1, (p_2 \rightarrow \mathrm{FL}(e_{21}, e_{22}), \ldots (p_n \rightarrow e_n, \mathrm{NIL}) \ldots))$$

We also add a capability for dealing with internal lambdas, PROG's, and global (SPECIAL) variables. These constructs involve a feature absent in the treatment of [19]—the concept of a variable and assignments made to it. In proving equivalence we will want to make certain that SPECIAL (global) variables are assigned their appropriate values; however, local variables and variables associated with internal lambdas (lambda variables) exist only as placeholders for computations. Therefore, for the latter, the act of assignment is only temporary and thus is not a necessary component of the equivalence. That is, in proving equivalence we wish to show that the programs perform the same computations on the Lisp environment which means that identical conditions are tested and identical side effects occur. In the case of local and lambda variables we simply use their bindings and ignore the act of assignment. In the case of SPECIAL variables we use their bindings as well as record the act of assignment.

In the process of obtaining a normal form we will be using the distributive law for functions and conditions. This will mean that certain computations, namely conditions, will be moved so that the physical position will no longer indicate the sequence of computation. For example, in the distributive law for functions given above, the predicate p is specified to be computed after $x_1, x_2, \ldots, x_{i-1}$ and before $q, r, x_{i+1}, x_{i+2}, \ldots, x_n$. However, after application of the distributive law the computation

of p might be misconstrued to take place before all other computations. In order to maintain a record of the original sequence of computation we need a representation of the Lisp program in terms of the order in which computations are performed. What is really desired is a numbering scheme having the characterization that associated with each computation is a number with the property that all of the computation's predecessors have lower numbers and the successors have higher numbers (a partial ordering). For example, see Figure 10, which is a numeric intermediate representation of the function REVERSE given in Figure 1. Notice that the atom NIL is assigned a computation number of zero and the computation numbers associated with atomic variables are smaller in magnitude than those associated with functions since the variables were computed (i.e., their bindings) prior to the execution of any computation in the program. The numbers will be seen to be useful in the proof procedure (see Section 4.3) when we will want to prove the validity of rearranging the order of computing arguments to a function.

4.2 Symbolic Interpretation

In order to obtain the intermediate representation of the object program we require an assembly-language understanding system. Such a system includes a mechanism for describing a computer instruction set and to some degree its basic architecture. Once such a mechanism is defined, we make use of what is termed symbolic interpretation to build the intermediate representation. This is done by activating a set of procedures corresponding to instructions in the object program.

The procedures are expressed in terms of other primitive procedures (e.g. ACFIELD, EFFECTADDRESS, and CONTENTS in Figure 11) in which is embedded, to some extent, the execution-level definition of the high level language (see Section 3). For each instruction there is a procedure that specifies how the instruction affects an entity known as the computation model. This model reflects, by use of an equality database, the contents of the various data structures relevant to the execution of the program (e.g. accumulators, stack, etc.), the values of the conditions that have been tested, and any side effect computations that have taken place. The procedural description must also provide a capability to invoke various parts of the object program as is the case when processing a condition, branch, or a function call.

As an example of the instruction-description facility, consider Figure 11 where the MOVE and HRRZS@ instructions of the PDP-10 are described. Each instruction is described via an MLISP FEXPR (a procedure whose arguments have not yet been evaluated). The argument to each such procedure represents a list containing all but the OPCODE fields of a Lap instruction. For example, symbolic interpretation of the (MOVE 2 1) instruction at label PC8 of Figure 2 will result in the invocation of the MOVE procedure with ARGS being bound to the list (2 1)—i.e., we have the procedure call (MOVE 2 1). This is

Fig. 11. MOVE and HRRZS@ instruction descriptions.

```
FEXPR MOVE(ARGS);
LOADSTORE(ACFIELD(ARGS),CONTENTS(EFFECTADDRESS(ARGS)));

FEXPR HRRZS@(ARGS);
BEGIN
  NEW ADDRESS,CONTNTS;
  ADDRESS←INDIRECT(CONTENTS(EFFECTADDRESS(ARGS)));
  CONTNTS←EXTENDZERO(RIGHTCONTENTS(ADDRESS));
  LOADSTORE(ACFIELD(ARGS),CONTNTS);
  LOADSTORE(ADDRESS,CONTNTS);
END;
```

all made possible by the EVAL mechanism of Lisp which enables the program and data to be indistinguishable.

The instruction descriptions are used in the following manner. In the case of the (MOVE 2 1) instruction at label PC8 in Figure 2, the computation model is updated by LOADSTORE to indicate that accumulator 2 contains the same computation as accumulator 1, which is known by the model to contain CONS(CAR(L), NIL). In the case of the (HRRZS@ 2 0 12) instruction at label PC5 in Figure 7, the computation model is updated by LOADSTORE to indicate that accumulator 2 contains CDR applied to the top of the stack, which is known by the computation model to contain L. In other words, accumulator 2 contains CDR(L). In addition, the computation model is updated by LOADSTORE to indicate that the left half of the location pointed at by L is loaded with 0 (i.e., NIL). However, this is the definition of a RPLACA operation and thus RPLACA(L, NIL) is also added to the set of computations that have been performed. Note that nowhere in the procedural definition of HRRZS@ is there any indication that CDR is being computed. We are able to detect the computation of CDR by virtue of the act of fetching the right half of the contents of a Lisp pointer. This is because the computation model is aware that the contents of the left and right halves of a cell pointed at by a Lisp pointer contain CAR and CDR, respectively, of the pointer. Such computations are recognized by the primitives which are used to describe the instructions (e.g. CONTENTS, EFFECTADDRESS, etc.). Clearly, other instructions can be used to achieve the effect of CDR, yet we do not need to state this in our instruction description.

The MOVE and HRRZS@ instructions have straightforward instruction descriptions since their only effect is the modification of the computation model. Other instructions perform control operations such as conditional branching as well as modify the computation model. In this case we need additional descriptive mechanisms.

For example, when conditional branching instructions are encountered, the symbolic interpretation process attempts to form a description of a test using constructs of the high level language and then determines if its value is known. In the affirmative case the appropriate path is taken and the next instruction along the path is symbolically interpreted. Such situations arise when either the operands of the test do not involve data items of the high level language (e.g. Lisp pointers) or the condition represents a test whose value has been determined earlier in the execution path. The determination of the value of a test is accomplished by interrogating

Fig. 12. Tree representation of a test.

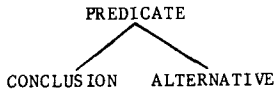


Fig. 13. SKIPN instruction description.

```
FEXPR SKIPN(ARGS);
BEGIN
  NEW MEMG,TST;
  MEMG←CONTENTS(EFFECTADDRESS(ARGS));
  IF ACFIELD(ARGS) NEQ 0 THEN LOADSTORE(ACFIELD(ARGS),MEMG);
  TST←CHECKTEST(MEMG,ZEROCNST);
  IF TST THEN RETURN(
    IF CDR TST THEN NEXTINSTRUCTION()
    ELSE UNCONDITIONALSKIP());
  FALSEPREDICATE();
  CONDITIONALSKIP(ARGS,FUNCTION SKIPNTRUE);
  SKIPALTERNATIVE(ARGS,FUNCTION SKIPNFALSE);
END;

FEXPR SKIPNTRUE(ARGS);
UNCONDITIONALSKIP();

FEXPR SKIPNFALSE(ARGS);
NEXTINSTRUCTION();
```

the computation model which is cognizant of the results of all tests along the execution path. If the condition is a test whose value is unknown, then the two alternate paths are symbolically interpreted in order, and the result returned is a tree as shown in Figure 12.

Prior to the evaluation of each path, the computation model is updated to reflect the assumed value of the test. This includes modification of relevant memory locations as well as propagating equalities and inequalities, as the case may be, through the equality database. This latter step is crucial to having the capability to recognize the occurrence of substitution of equals for equals. For example, in Algorithm 1 of REVERSE, once L is known to be equal to NIL we may use L or NIL interchangeably.

The equality database is a set of equivalence classes and pairs of inequivalences which have resulted from the symbolic interpretation of the various conditions along an execution path. Transitivity and functional application are fully propagated. Equality and inequality of two operands is determined by parsing their symbolic representations and checking if they are members of the same equivalence class [25]. If the two operands are members of the same equivalence class, then they are known to be equal. If the parsing process determines that the two operands are in different equivalence classes, then we must determine if these two classes are known to be inequivalent. This is achieved by assuming that the two equivalence classes are equal and correspondingly updating the database to reflect the merging of the two classes. If a contradiction is obtained during the propagation of the equivalence, then the two operands are known to be unequal. Otherwise, the equality of the two operands is unknown.

An example of a conditional branch instruction is SKIPN (see Figure 13) which is used to skip the following instruction if the contents of the effective address is nonzero. In addition, if the accumulator name specified by the accumulator field is nonzero, then the said accumulator is loaded with the contents of the effective

Fig. 14. Result of symbolic interpretation of (SKIPN 21) in Figure 3.

address. The instruction description first tests for the occurrence of a nonzero accumulator name. Once this is done, we see the use of several control primitives in order to provide for the symbolic interpretation of both alternatives of the test. CHECKTEST examines the operands and, if possible, forms a valid test. (TST is a local variable which temporarily records the result of the test.) Next, if the value of the condition is already known, then appropriate action is taken. FALSEPREDICATE marks the sense of the test. (An instruction skipping on equality with zero would use TRUEPREDICATE.) CONDITIONALSKIP and SKIPALTERNATIVE (and similarly CONDITIONALJUMP and JUMPALTERNATIVE in the case of a conditional branch) serve to recursively invoke the symbolic interpretation of the paths corresponding to the true and false cases of the test. One of the parameters to these primitives is the name of another routine which specifies any further processing that might be required prior to executing the path. The actual updating of the computation model occurs in control routines such as CONDITIONALSKIP and SKIPALTERNATIVE. Specifically, the computation model is saved in CONDITIONALSKIP prior to the reinvocation of the symbolic interpretation process for the true case of the condition and restored to its previous value prior to exiting from CONDITIONALSKIP. Note that the computation model needs to be saved only when a conditional branch instruction is encountered. The construction of the tree corresponding to the result of the symbolic interpretation process occurs in SKIPALTERNATIVE.

Whenever the symbolic interpretation process is about to interpret an instruction which has been previously encountered along the path being symbolically interpreted (i.e. loop shortcutting), then recursion is assumed to have taken place (recall the branch to label REV in Figure 8.) In such a case, the symbolic interpretation process will attempt to show that if a branch had indeed been made to the start of the program, then the said instruction would have been reached with the same state of the computation model by virtue of known values for all of the conditions along some path to the instruction in question. If such a path from the start of the program exists, then it is unique since a condition cannot be both true and false. Note that the contents of all locations that are subsequently referenced prior to being overwritten must contain appropriate values. In fact, this is one of the ways the error was detected in the branch from PC6 to REV in Figure 7. In this case accumulator 2 and the top of the stack will both be referenced subsequent to REV prior to being overwritten. Furthermore, when REV is entered via PC2, accumulator 2 and the top of the stack contain identical or equivalent values. Yet, when REV is entered via PC6, accumulator 2 contains CDR(L) while the top of the stack contains L.

Fig. 15. Intermediate representation corresponding to Figure 3.

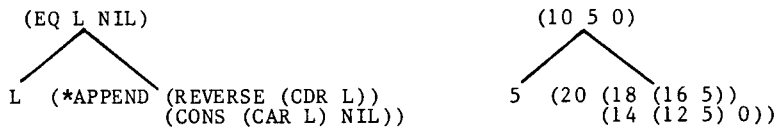


Fig. 16. Partial trace of proof of equivalence of Figure 3 and Figure 1.

```
1.  MANIPULATE NORMAL FORM TO MATCH REDERIVED FORM

***NORMAL***
 (EQ L NIL)                                    (30 5 0)

NIL     (*APPEND (REVERSE (CDR L))          0     (40 (34 (32 5))
                 (CONS (CAR L) NIL))                (38 (36 5) 0))
***REDERIVED***
 (EQ L NIL)                                    (10 5 0)

NIL     (*APPEND (REVERSE (CDR L))          0     (20 (18 (16 5))
                 (CONS (CAR L) NIL))                (14 (12 5) 0))

1.1.  TRYING TO MATCH THE COMPUTATION NUMBERS APPEARING IN THE FUNCTION

(EQ L NIL)                                    (10 5 0)
***BY USING THE FORM***
 (EQ L NIL)                                   (30 5 0)

NIL     (*APPEND (REVERSE (CDR L))          0     (40 (34 (32 5))
                 (CONS (CAR L) NIL))                (38 (36 5) 0))

1.1.1.  COMPUTATION NUMBER 10 IS MATCHED BY COMPUTATION NUMBER 30

1.2.  TRYING TO MATCH THE COMPUTATION NUMBERS APPEARING IN THE FUNCTION

NIL                                           0
***BY USING THE FORM***
NIL                                           0

1.3.  TRYING TO MATCH THE COMPUTATION NUMBERS APPEARING IN THE FUNCTION

(*APPEND (REVERSE (CDR L))                    (20 (18 (16 5))
         (CONS (CAR L) NIL))                      (14 (12 5) 0))
***BY USING THE FORM***
(*APPEND (REVERSE (CDR L))                    (40 (34 (32 5))
         (CONS (CAR L) NIL))                      (38 (36 5) 0))

1.3.1.  COMPUTATION NUMBER 12 IS MATCHED BY COMPUTATION NUMBER 36

1.3.2.  COMPUTATION NUMBER 14 IS MATCHED BY COMPUTATION NUMBER 38

1.3.3.  COMPUTATION NUMBER 16 IS MATCHED BY COMPUTATION NUMBER 32

1.3.4.  COMPUTATION NUMBER 18 IS MATCHED BY COMPUTATION NUMBER 34

1.3.5.  COMPUTATION NUMBER 20 IS MATCHED BY COMPUTATION NUMBER 40
```

As an example of the symbolic interpretation process, consider the encoding of Algorithm 1 of REVERSE given in Figure 3. The first instruction that we encounter is SKIPN which is used to skip to label PC3 if accumulator 1 is nonzero. The result is shown in Figure 14. Recalling that NIL is represented by zero, we observe that the test corresponds to checking if the list L is NIL—i.e. (EQ L NIL). Since neither of the paths corresponding to the true and false cases of the test have yet been symbolically interpreted, we denote the two subtrees as UNKNOWN-CONCLUSION and UNKNOWN-ALTERNATIVE. This procedure is continued in this manner until the symbolic and numeric intermediate representations shown in Figure 15 are obtained. The numbers are assigned to the computations as they are symbolically interpreted. Again, only the relative ordering is significant.

4.3 Proof

The proof procedure makes no use of a theorem prover. Instead, it relies on the intermediate representation of the original function being placed in a normal form. This form is obtained by a two-part algorithm. The first part corresponds to application of axioms (2), (3), and (7) along with the distributive law for functions while simultaneously binding variables to their proper values. The latter is necessary when SETQ's (Lisp's assignment operator) and internal lambdas are being used. The second part corresponds to making use of axioms

(2), (3), (5), and (6) to remove predicates whose values are known and to replace duplicate occurrences of computations by their first occurrence. As an example of the former consider the gbf $(p \leftarrow (q \leftarrow (p \leftarrow a, b), c), d)$. The second instance of p is redundant and is replaced by its equivalent—i.e. T. Subsequent application of axiom (2) results in the form $(p \leftarrow (q \leftarrow a, c), d)$.

Note that when duplicate occurrences of computations are replaced by their first occurrences we must make sure that they are indeed duplicate. For example, when a computation such as CDR(L) is computed more than once along an execution path, then the second occurrence is a duplicate only if no computations took place between the two instances which might cause them to have different values (e.g. a RPLACD operation which has as its side effect the modification of a right half of some element in the List Structure). The detection of redundancies and the determination that no intervening side effect computations have occurred is aided by a flow analysis and by the numeric representation where the latter indicates a relative order for the instances of computation of all computations along an execution path.

At this point we have a normal form representation for the original Lisp program. As mentioned earlier, the symbolic interpretation procedure returns a similar representation for the object program. To this representation we also apply the second part of the above normal form algorithm to replace duplicate occurrences of computations by their first occurrence. In cases where no computation rearrangement or loop shortcutting has taken place, equivalence will now hold. For many compilers such situations are not uncommon. Otherwise we must prove that using our axioms, either one of the forms can be transformed into the other. This procedure, termed matching, begins by attempting to prove that each computation appearing in one of the forms appears in the other and vice versa. This matching process is accomplished by uniformly assigning the computation numbers in one form, say B, to be higher than all of the numbers in the other form, say A, and then, in increasing order, search form B for matching instances of computations appearing in form A. Whenever a computation, say C, in A is matched by a computation, say D, in B, then D is replaced by C, which has a lower computation number and the proof continues. During the proof liberal use is made of axioms (1), (2), (3), (5), and (6) as well as substitution of equals for equals, and possibly functional expansion in cases of loop shortcutting where conditions are precomputed (e.g. in the proof of the equivalence of Figures 4 and 8).

As an example in which a computation appears in one form and not in the other consider a slightly modified version of Figure 6 where PC7 is (JUMPN 2 PC1) instead of (JUMPN 2 REV) and PC5 has changed from (HRRZ@ 2 0 12) to (HRRZS@ 2 0 12). In this case, the only problem is that an (RPLACA L NIL) results from the HRRZS@ operation at PC5 in addition to loading accumulator 2 with (CDR L). Unfortunately, the original program in Figure 4 does not call for the computation of (RPLACA L NIL). Thus we see why the proof system must prove that each computation that appears in one form must appear in the other form and vice versa.

For our examples we must prove the equivalence of Figures 9, 10, and 15. Symbolically, the two intermediate representations are almost identical. The only difference is that when (EQ L NIL) is true, the intermediate representation corresponding to the original program indicates that a value of NIL is to be returned while the intermediate representation corresponding to the Lap program in Figure 3 indicates that L ought to be returned. In fact, this discrepancy is resolved by the component of the proof procedure which replaces duplicate occurrences of computations by their first occurrence. Thus, in reality, L is replaced by NIL since NIL has a lower computation number by virtue of the numbering scheme which assigns a computation number of 0 to the atoms T and NIL.

The numeric intermediate representations of the arguments to *APPEND also differ. Specifically, (REVERSE (CDR L)) is computed before (CONS (CAR L) NIL) in the original program while the Lap program in Figure 3 reverses this order. The proof must show that this variation preserves equivalence. Since no global variables are being referenced here, the only possible side effects are RPLACA and RPLACD operations whose occurrence would affect the rearrangement of the order of computing functions involving the computation of CAR or CDR, respectively. However, neither CONS, REVERSE, nor *APPEND involve such operations. Thus the rearranging is valid.

One half of the actual proof for our example, a proof that each computation in the intermediate representation corresponding to Figure 3 (termed the rederived form) appears in the intermediate representation corresponding to Figure 1 (termed the normal form) is given in Figure 16. There are several items of note. First, when (EQ L NIL) is true, NIL, rather than L, is being used since duplicate computation removal has already been applied. Second, all computation numbers of functions in the normal form are higher than all of the computation numbers in the rederived form. Of course, the occurrences of the atoms NIL and L have the same computation numbers in both forms since they can be thought to be computed (i.e., bound in the case of L) prior to the invocation of the function being processed. For example, in proving that (CAR L) in the rederived form is matched by (CAR L) in the normal form, we must prove that between computation number 12 and 36 no operation is performed in the normal form whose result is the modification of the left half of a Lisp cell. In other words, computation numbers 32 and 34 which correspond to (REVERSE (CDR L)) do not involve RPLACA operations. We are not concerned with computation number 30 which corresponds to the predicate EQ, since it has already been matched by computation 10 and hence has been replaced in the normal form by 10.

The examples we have seen illustrate that the two

intermediate representations are often very similar. In fact, if no loop shortcutting or rearrangement of computations (this includes common subexpression elimination) take place, then the intermediate representations will be identical. Nevertheless, such cases should not detract from our techniques; the point to note is that in these cases the symbolic interpretation process often results in the detection of errors as well as in the verification of the extraction of several redundant computations. For example, recall the errors in Figure 7 pertaining to the erroneous calling sequence.

5. Implementation Status

A system has been implemented to prove the correctness of translation of Lisp programs to Lap and is currently running on a PDP-10. Both Lisp 1.6 and UCI LISP [3] can be handled. The system is written in Mlisp and consists of two components which may be run separately. One component, DERIV, corresponds to the symbolic interpretation procedure and returns as its result a suitable intermediate representation for a Lap program. The second component, CANON, proves the equivalence of the original Lisp program and the output of DERIV.

The proof system is interactive in the sense of asking the user a set of questions to aid in the proof process. These questions deal with calling sequence conventions and properties of functions which may be needed in a proof. For example, the system needs to know if the optimization process makes use of the commutativity of functions other than PLUS and TIMES. Similarly, the system must be made aware of the antisymmetry of pairs of operations other than CONS-XCONS and LESS-GREAT (e.g. $A < B$ is equivalent to $B > A$).

When given a pair of programs to try to prove equivalent, the system will always terminate with a yes or no answer. A "yes" answer means that the programs are equivalent. A "no" answer results when either an error has been found or certain components of the intermediate representation cannot be matched due to insufficient equality information. There are two types of errors. Errors that are detected during symbolic interpretation generally correspond to an object program that does not obey calling sequence conventions—i.e. the program is not well formed. For example, recall the erroneous encoding in Figure 7. Errors detected during the proof procedure generally correspond to computations present in the object program and absent in the source program or vice versa. The actual location of many errors can be pinpointed by virtue of the numeric representation of the function that was processed by the symbolic interpretation procedure. Specifically, a dictionary is kept containing all of the computation numbers with the instruction number and execution path along which they have been computed. When the proof procedure detects certain errors, it outputs the dictionary entry associated with the offending computation.

The system has been used to prove the correctness or incorrectness of a large number of programs. For example, one program named Hier [24] was compiled by the Lisp 1.6 compiler to yield an encoding of 145 instructions. Using the types of optimizations discussed in Section 3, the object program was hand optimized to yield an encoding consisting of 105 instructions which, in addition, was 40 percent faster and required 50 percent less stack space. The most interesting aspect of the hand coding was that during the coding process a number of errors were made. However, the system was able to detect all of these errors and emit error messages that pinpointed the locations of the errors. The actual process of correcting the errors took several iterations through the proof system since only one error at a time can be detected for each execution path.

When using compiled code, the two components DERIV and CANON occupy 19K and 14K 36 bit words on a PDP-10. Of course there is a need for additional space for the basic Lisp system (25K) and the list structures. The latter is primarily dependent on the size and complexity of the program being processed.

The amount of time necessary to prove the correctness of translation is dependent on the size of the function and the type of optimization performed. We are primarily concerned with the number of conditions tested in each function since the symbolic interpretation process and the proof procedure must explore all possible execution paths. This implies a possible exponential contribution by the function size (in term of conditions) to the amount of time required to perform a proof (but see the note about COND in Section 6). Optimized encodings exhibiting loop shortcutting where conditions are precomputed, as in Figure 8, require slightly longer proofs since the symbolic interpretation process must prove that values of conditions whose computation has been bypassed are known, as well as demonstrate that all locations referenced subsequent to the target label of the instance of loop shortcutting have appropriate values. For example, on a PDP-10 (KL10-AA with 384K memory) using core images of 55K resulted in a proof of the equivalence of Figures 1 and 3 taking 8 seconds, Figures 4 and 8 taking 10 seconds, while 50 seconds were necessary for the hand-optimized version of Hier mentioned earlier.

6. Conclusion

An overview has been presented of a formalism for proving the correctness of translations involving a heuristic code optimization process. The formalism has been demonstrated by means of a system which proves the correctness of translations involving Lisp and Lap. Clearly, instruction sets of other computers could also be handled. Extending our ideas to other computers would serve to highlight any deficiencies in our machine-description process. A more ambitious direction is to at-

tempt to apply our techniques to other high level languages. Admittedly, Lisp is a rather simple language in terms of its constructs. In particular, its control structure (i.e. case analysis) is quite similar to our intermediate representation. Nevertheless, we feel that other well-structured high level languages could also be handled.

In Section 4 we mentioned a restriction on backward jumps. Specifically, we said that, at present, in a PROG we can only handle the GO construct whose target label has not occurred physically prior to the GO statement. This restriction is related to the question of loops and backward jumps. Part of the problem is our insistence on interpreting recursion to have occurred whenever the symbolic interpretation process is about to interpret an instruction which has been previously encountered along the path (i.e. loop shortcutting). A solution is to break up the original high level language program into modules of intervals [1, 2] having one entry point and several exit points. Branches which jump back anywhere within the interval other than to the entry node of the interval are assumed to be cases of loop shortcutting. Branches to points other than entry nodes in other intervals are also assumed to be cases of loop shortcutting. Proofs would be necessary for each of these intervals.

Symbolic interpretation, as well as the proof process, must exercise all possible execution paths. This is different from symbolic execution [12] where various cases of a high level language program are tested by use of symbolic values for the parameters. Thus our system has a potential drawback in that for a program with a large number of IF THEN ELSE statements the intermediate representation (i.e. the tree) might grow to be rather large. Specifically, for N such statements we might have to process $2^{**}N$ execution paths. Fortunately, conditions in recursive programs of the nature with which we are dealing are more of the form of a COND. In such a case N conditions only represent $N + 1$ execution paths. Nevertheless, the problem associated with the $2^{**}N$ execution paths could be alleviated by use of the notion of intervals presented above. In fact, scrutiny of many Lisp programs reveals them to consist of a large set of small recursive functions very much akin to the notion of intervals. Alternatively, a facility could be provided for the user to select which execution paths are to be tested.

The system as presented here finds usefulness as a postoptimization component of a compiler. It is also well-suited to an interactive optimization process where a user sits at his terminal and interactively applies transformations to his program. During this process, mistakes may be made, and if possible they are detected, and the user is informed of his errant ways. This is quite similar to a system for achieving the type of results proposed in [13].

Another potential application of a proof system for the correctness of translation is in the domain of bootstrapping. Suppose we only have a Lisp interpreter available and we desire to have a compiler. The solution is to write such a compiler in Lisp, say C, and let the compiler translate itself to yield C' written in assembly language. A proof system such as the one described herein can be used to prove that C and C' are equivalent and hence that they generate equivalent code. This is accomplished by proving that C has correctly translated C to yield C'. In fact, such ideas are potentially useful in dealing with crosscompilers for minicomputers. However, we might add as a note of caution, that different machine architectures may cause problems with respect to different word sizes, character formats, input-output primitives, and other machine-dependent factors.

8. Appendix

PDP-10 Operations

CALL	A special Lap instruction which is analogous to a PUSHJ. The difference is that it is used to invoke Lisp functions via the property list. This is useful when a trace of the arguments to a function is desired, or when the actual binding of a function changes. (CALL num (E fname)) denotes a CALL to fname where num is the number of arguments.
HLRZ	Load the right half of accumulator AC with the left half of the contents of the effective address and clear the left half of AC.
HRLZ@	Same as HLRZ with indirect addressing.
HRRZ	Load the right half of accumulator AC with the right half of the contents of the effective address and clear the left half of AC.
HRRZ@	Same as HRRZ with indirect addressing.
HRRZS	Load the right half of accumulator AC with the right half of the contents of the effective address and clear the left half of AC. The same value is also stored in the effective address—i.e., the left half of the effective address is set to zero.
HRRZS@	Same as HRRZS with indirect addressing.
JCALL	A special Lap instruction which is analogous to a JRST. The difference is that it is used to invoke Lisp functions via the property list. This is useful when a trace of the arguments to a function is desired, or when the actual binding of a function changes.
JRST	Unconditional jump to the effective address.
JUMPE	Jump to the effective address if the contents of accumulator AC is zero; otherwise continue execution at the next instruction.
JUMPN	Jump to the effective address if the contents of accumulator AC is unequal to zero; otherwise continue execution at the next instruction.
MOVE	Load accumulator AC with the contents of the effective address.
MOVEI	Load the right half of accumulator AC with the effective address, and clear the left half.
POP	Move the contents of the location addressed by the right half of accumulator AC to the effective address and then subtract octal 1 000 001 from AC to decrement both halves by one. If the subtraction causes the count in the left half of AC to reach −1, then the Pushdown Overflow flag is set.
POPJ	Subtract octal 1 000 001 from accumulator AC to decrement both halves by one. If subtraction causes the count in the left half of AC to reach −1, then set the Pushdown Overflow flag. The next instruction is taken from the location addressed by the right half of the location that was addressed by AC prior to decrementing.
PUSH	Add octal 1 000 001 to accumulator AC to increment both halves by one and then move the contents of the

effective address to the location now addressed by the right half of AC. If the addition causes the count in the left half of AC to reach zero, then set the Pushdown Overflow flag.

PUSHJ — Add octal 1 000 001 to accumulator AC to increment both halves by one. If addition causes the count in the left half of AC to reach zero, then set the Pushdown Overflow flag. Store the contents of the program counter and the processor flags in the right and left halves, respectively, of the location now addressed by the right half of AC, and continue execution at the effective address.

SKIPN — Skip the next instruction if the contents of the effective address is not equal to zero. If the AC field specification is nonzero, then load accumulator AC with the contents of the effective address.

SUB — The contents of the effective address is subtracted from the contents of accumulator AC, and the result is left in AC.

Acknowledgments. Special thanks go to Professor Vint Cerf for his constant advice and encouragement during a period in which some of this research was pursued. Thanks also go to the referees and the editor, Ben Wegbreit, for asking the right questions.

Received February 1976; revised August 1977

References

1. Aho, A., and Ullman, J.D. *The Theory of Parsing, Translation, and Compiling*, Vol. 2. Prentice-Hall, Englewood Cliffs, N.J., 1973, p. 938.
2. Allen, F.E. Control flow analysis. SIGPLAN Notices (ACM), *5*, 7 (July 1970), 1–19, 239–307.
3. Bobrow, R.J., Burton, R.R., and Lewis, D. UCI LISP Manual. Inform. and Comptr. Sci. Tech. Rep. No. 21, U. of California at Irvine, Oct. 1972.
4. Boyer, R.S., and Moore, J.S. Proving theorems about LISP functions. *J. ACM 22*, 1 (Jan. 1975), 129–144.
5. Burstall, R.M., and Darlington, J. Some transformations for developing recursive programs. Proc. 1975 Int. Conf. on Reliable Software, April 1975, pp. 465–472.
6. Digital Equipment Corp. PDP-10 System Reference Manual. Digital Equipment Corporation, Maynard, Mass., 1969.
7. Deutsch, L.P. An interactive program verifier. Ph.D. Th., Dept. Comptr. Sci., U. of California at Berkeley, May 1973.
8. Floyd, R.W. Assigning meanings to programs. *Proceedings of a Symposium in Applied Mathematics*, Vol. 19, *Mathematical Aspects of Science*, J.T. Schwartz, Ed., Amer. Math. Soc., Providence, R.I. 1967, pp. 19–32.
9. Gerhart, S.L. Correctness preserving program transformations. Proc. Second ACM Symp. on Principles of Programming Languages, Jan. 1975, pp. 54–66.
10. Hollander, C.R. Decompilation of object programs. Ph.D. Th., Tech. Rep. No. 54, Digital Syst. Lab., Dept. of Electr. Eng., Stanford U., Stanford, Calif., 1973.
11. King, J.C. A program verifier. Ph.D. Th., Dept. of Comptr. Sci., Carnegie-Mellon U., Pittsburgh, Pa., 1969.
12. King, J.C. Symbolic execution and program testing. *Comm. ACM 19*, 7 (July 1976), 385–394.
13. Knuth, D. Structured programming with GO TO statements. Rep. STAN-CS-74-416, Comptr. Sci. Dept., Stanford U., Stanford, Calif., May 1974.
14. Lee, J.A.N. *Computer Semantics.* Van Nostrand Reinhold, New York, 1972, 346–347.
15. London, R.L. Correctness of two compilers for a LISP subset. Stanford Artif. Intell. Proj. Memo AIM-151, Comptr. Sci. Dept., Stanford U., Stanford, Calif., Oct. 1971.
16. London, R.L. The current state of proving programs correct. Proc. ACM 25th Annual Conf., 1972, pp. 39–46.
17. Loveman, D.B. Program improvement by source to source transformation. Proc. Third ACM Symp. on Principles of Programming Languages, Jan. 1976, p. 145.
18. McCarthy, J. Recursive functions of symbolic expressions and their computation by machine. *Comm. ACM 3*, 4 (April 1960), 184–195.
19. McCarthy, J. A basis for a mathematical theory of computation. In *Computer Programming and Formal Systems*, P. Braffort and D. Hirshberg, Eds., North Holland, Amsterdam, 1963.
20. Newell, A. Artificial intelligence and the concept of mind. In *Computer Models of Thought and Language*, R.C. Schank and K.M. Colby Eds., W.H. Freeman, San Francisco, 1973, pp. 1–60.
21. Quam, L.H., and Diffie, W. Stanford LISP 1.6 Manual. Stanford Artif. Intell. Proj. Operating Note 28.7, Comptr. Sci. Dept., Stanford U., Stanford, Calif., 1972.
22. Reiser, J.F., Ed. SAIL User Manual. Stanford Artif. Intell. Proj. Memo AIM-289, Comptr. Sci. Dept., Stanford U., Stanford, Calif., Aug. 1976.
23. Samet, H. Automatically proving the correctness of translations involving optimized code. Ph.D. Th., Stanford Artif. Intell. Proj. Memo AIM-259, Comptr. Sci. Dept., Stanford U., Stanford, Calif., 1975.
24. Samet, H. A study in automatic debugging of compilers. TR-545, Comptr. Sci. Dept., U. of Maryland, College Park, Md., 1977.
25. Samet, H. Equivalence and inequivalence of instances of formulas. TR-553, Comptr. Sci. Dept., U. of Maryland, College Park, Md., 1977.
26. Smith, D.C. MLISP. Stanford Artif. Intell. Proj. Memo AIM-135, Comptr. Sci. Dept., Stanford U., Stanford, Calif., Oct. 1970.
27. Suzuki, N. Verifying programs by algebraic and logical reductions. Proc. 1975 Int. Conf. on Reliable Software, April 1975, pp. 473–481.
28. Wegbreit, B. Property extraction in well-founded property sets. *IEEE Tran. Software Eng.* SE-1,31 (Sept. 1975), pp. 270–285.
29. Wegbreit, B. Goal-directed program transformation. Proc. Third ACM Symp. on Principles of Programming Languages, Jan. 1976, pp. 153–170.