A NORMAL FORM FOR COMPILER TESTING*

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

Abstract

A formalism is presented for obtaining a normal form to be used in representing programs for compiler testing. Examples are used to motivate the features that must be considered when developing such a formalism. It is particularly suitable for heuristically optimized code and has been successfully used in a system for proving that programs written in a subset of LISP are correctly translated to assembly language.

1. INTRODUCTION

In [Samet75] work is reported on a system designed to deal with the problem of proving the correctness of translations performed by translators which do a considerable amount of heuristic code optimization. In this paper we present the formalism used in that work to represent the high level source program. This formalism is termed a "normal form."

The paper is organized into several sections. First, we present a summary of related work. This is followed by a brief indication of what constitutes a proof. Next, we give an example of the problem for which our representation has been designed. Once the previous overview has been accomplished, we summarize the normal form that we have chosen. This discussion deals with the motivation for our choice, as well as a presentation of its adaptation to a high level language.

2. RELATED WORK

Compiler testing is a term we use to describe a means of proving that given a compiler (or any program translation procedure) for high level language H and low level language L, a program written in language H is correctly translated to language L. We are especially interested in cases where the translation involves a considerable amount of "low level" optimization [Loveman76]. Some possible approaches to the problem include program proving [London72] and program testing [Huang75].

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

Most of the previous work in correctness from the program proving approach has dealt with specifying assertions ([Floyd67],[King69]) about the intent of the program and then proving that they do indeed hold. The assertions correspond to a detailed formal specification of what constitutes correct program behavior. The process of specifying assertions is a rather difficult one ([Deutsch73], [Suzuki75]), and even when a program is found to satisfy the stated assertions there is no guarantee that the assertions were sufficiently precise to account for all of the contingencies (i.e. there is considerable difficulty in specifying machine dependent details such as overflow, precision, etc.). This difficulty is further compounded when the programs to be proved are of such a complexity that they defy formal analysis (i.e. the exact meaning of the program is not even well understood). Proofs using assertions generally require the aid of a theorem prover and in the case of a compiler they may be characterized as proving that there does not exist a program that is incorrectly translated by the compiler. We feel that such an approach is unworkable for an optimizing compiler although it has been done for a simple LISP [McCarthy60] compiler [London71].

Program testing is a concept which has been gaining an increasing amount of attention in recent years. This is in part due to the realization that formal program proving methods rely on a very powerful theorem proving capability which is unlikely to appear in the near future. Currently, program testing consists of applying a suitable set of test criteria to the program at hand. This is much easier said than done since the formulation of suitable test criteria is not an easy process, and, once done, there still remains the problem of test case generation.

3. COMPILER TESTING

Our notion of compiler testing is a variation on the concept of program testing. We feel that in the case of a compiler there exists a willingness to settle for proofs that specific programs are correctly translated from a high level language to the object language. We embed a proof system in the compiler which proves the correctness of the translation of each program that is compiled as part of the compilation process. This sidesteps the issue of proving that there does not exist a program that is incorrectly compiled; but this issue is now moot since essentially we are only interested in the correctness of translation of programs input to the compiler. Alternatively, we are not interested in the question of the correctness of the translation of programs that have not yet been input to the compiler. In other words, we bootstrap ourselves to a state where an "effective correctness" can be attributed to the compiler by virtue of the correct translation of programs input to it.

Our concept of "test" consists of demonstrating a correspondence or equivalence between a program input to the compiler and the corresponding translated program. The manner in which we proceed is to find an intermediate representation which is common to both the original and translated programs and then demonstrate their equivalence. This intermediate representation is termed a normal form and an example of it is presented, along with the process of obtaining it, in greater detail in section 5. The process of obtaining this representation for the translated program is called "symbolic interpretation" [Samet76]. This process makes use of procedural descriptions of the primitive operations of the object language.

Before proceeding any further, let us 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 [Lee72], that is they have identical execution sequences except for certain valid rearrangements of computations. Such rearrangements include transformations classified as "low level" optimizations. However, more ambitious transformations classified as "source level" ([BurstallDarlington75], [Gerhart75], [Wegbreit76]) 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 the process of demonstrating equivalence no use is made of the purpose of the program. Thus, for example, having the knowledge that a high level program uses insertion sort and a low level program uses quicksort to achieve sorting of the input is of no use in proving equivalence of the two programs. Recall, that sorting is an input-output pair characterization of an algorithm.

4. EXAMPLE

The intermediate representation used in our proofs is dependent to some extent on the high level language for which the translation procedure has been designed. In order to have a framework for the discussion, we must assume the existence of a suitable programming language. Our language is a subset of LISP 1.6 [Quam72] which will be shown to have an intermediate representation. Briefly, this subset 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 local variables - it may not access another function's local variables. 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.

In order to illustrate the type of programs our system is designed to handle we give an example of a high level language program, a low level language program, and their intermediate representations.

Consider the function INTERSECTION whose MLISP [Smith70] (a parentheses free LISP also known as meta-LISP) definition is given in figure 1. The function takes as its arguments two lists U and V and returns as its result a list of all the elements that appear in both lists. Each element is assumed to occur only once in each list. For example, application of the function to the lists (A B C) and (D C B) results in the list (B C).

```
INTERSECTION(U,V) = if NULL(U) then NIL
                    else if MEMBER(CAR(U),V) then
                       CONS(CAR(U),
                            INTERSECTION(CDR(U),V))
                    else INTERSECTION(CDR(U),V)
```

Figure 1 - MLISP Encoding of INTERSECTION

The low level language with which we are dealing is LAP [Quam72] - a variant of the PDP-10 [DEC69] assembly language. A LISP cell is assumed to be represented by a full word where the 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 registers and can be used for indexing), and also 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. 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)) which leave all accumulators unchanged with the exception of 1 and 2.

Figure 2 contains a LAP encoding, obtained by a hand coding process, for the function given in figure 1. 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 num1 num2) appearing in the address field of an instruction denotes the address of a word containing num1 and num2 in its left and right halves respectively (assuming num1 and num2 are less than 15). The meaning of the instructions used in the example LAP encoding should be clear from the adjoining comments.

Figures 3 and 4 give a symbolic intermediate representation of the functions encoded by figures 1 and 2 respectively. The intermediate representations for the two functions are almost identical with the exception of the true case for (EQ U NIL). However, use of equality information resolves this problem since in the former case U and NIL may be used interchangeably. Nevertheless, there remains another problem. The two functions do not have identical execution sequences. Thus there is a need for more than just a symbolic representation. We also need a means of representing the order of execution of various computations. For example, INTERSECTION(CDR(U),V) is only computed once in figure 2 whereas in figure 1 its computation is called for at two separate instances. Moreover, it is computed before MEMBER(CAR(U),V) rather than afterward. In this case we must be able to prove that no side-effect computation (e.g. an operation having the effect of a RPLACA or RPLACD) can occur between the instance of computation of INTERSECTION(CDR(U),V) and the instances of its instantiation. This information is obtained via flow analysis. Conflicts with respect to the order of computing functions are resolved by use of an additional intermediate representation which reflects the instances at which various computations were performed. It is the task of the proof procedure to verify that these variations preserve equivalence.

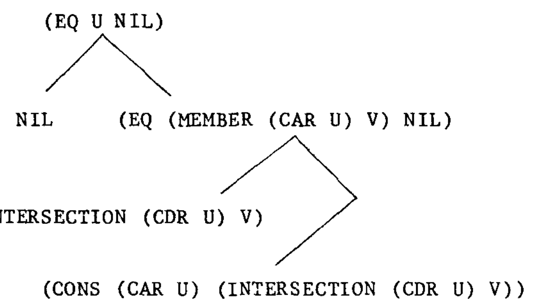
(EQ U NIL)
NIL
(EQ (MEMBER (CAR U) V) NIL)
(INTERSECTION (CDR U) V)
(CONS (CAR U) (INTERSECTION (CDR U) V))

Figure 3 - Intermediate Representation of Figure 1

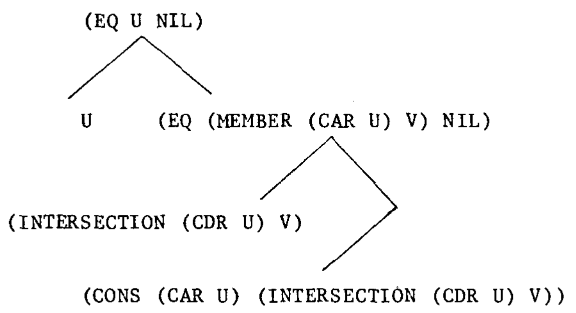
(EQ U NIL)
U
(EQ (MEMBER (CAR U) V) NIL)
(INTERSECTION (CDR U) V)
(CONS (CAR U) (INTERSECTION (CDR U) V))

Figure 4 - Intermediate Representation of Figure 2

5. NORMAL FORM

Our formalism has its root in the work done by McCarthy [McCarthy63] 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.

```
INTERSECTION  (JUMPE 1 TAG1)                jump to TAG1 if U is NIL
              (PUSH 12 1)                   save U on the stack
              (PUSH 12 2)                   save V on the stack
              (HRRZ 1 0 1)                  load accumulator 1 with CDR(U)
              (CALL 2 (E INTERSECTION))     compute INTERSECTION(CDR(U),V)
              (MOVE 2 0 12)                 load accumulator 2 with V
              (MOVEM 1 0 12)                save INTERSECTION(CDR(U),V)
              (HLRZ@ 1 -1 12)               load accumulator 1 with CAR(U)
              (CALL 2 (E MEMBER))           compute MEMBER(CAR(U),V)
              (EXCH 1 0 12)                 save MEMBER(CAR(U),V) and load accumulator
                                            1 with INTERSECTION(CDR(U),V)
              (HLRZ@ 2 -1 12)               load accumulator 2 with CAR(U)
              (SKIPE 0 0 12)                skip if MEMBER(CAR(U),V) is not true
              (CALL 2 (E XCONS))            compute CONS(CAR(U),INTERSECTION(CDR(U),V))
              (SUB 12 (C 0 0 2 2))          undo the first two push operations
TAG1          (POPJ 12)                     return
```

Figure 2 - LAP Encoding of INTERSECTION

The basic entity is a generalized boolean form (gbf) which has the form $(p\to x,y)$ where p,x, and y are variables or gbfs and are known as the premise, conclusion, and alternative respectively. p takes the value of T, NIL, or undefined in which case the gbf takes the value x,y, or undefined respectively. Two gbfs are said to be strongly equivalent 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.

(1) $(p\to a,a) =_w a$

(2) $(T\to a,b) = a$

(3) $(NIL\to a,b) = b$

(4) $(p\to T,NIL) = p$

(5) $(p\to(p\to a,b),c) = (p\to a,c)$

(6) $(p\to a,(p\to b,c)) = (p\to a,c)$

(7) $((p\to q,r)\to a,b) = (p\to(q\to a,b),(r\to a,b))$

(8) $(p\to(q\to a,b),(q\to c,d)) = (q\to(p\to a,c),(p\to b,d))$

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 process 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. Our algorithms are different from those proposed by McCarthy where additional axioms are introduced to cope with obtaining a normal form for strong equivalence. By ordering the variables appearing in premise positions according to some lexicographical scheme the normal form becomes a canonical form and we have the following result.

Theorem: Two gbfs are equivalent (weak or strong) iff they have the same (weak or strong) canonical form.

In this paper we are only interested in obtaining a representation of the function in some normal form with no rearranging of conditions (axiom (8)) or application of axiom (1). This is because the normal form corresponding to a LISP program will be used in a system for proving that LISP programs are correctly compiled. At that point these axioms will be used in an attempt to match the normal form corresponding to the original LISP function definition with the normal form of the object program which has been obtained by use of symbolic interpretation.

In order for the previous ideas to be useful in proving the correctness of translation of LISP programs we must show how to adapt them to LISP programs. We are primarily interested in proving strong equivalence and in the more general notion of functions rather than variables.

The relation of functions to gbfs is given by the distributive law:

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

Let FL be a function of one or more arguments which returns as its result the value of its final argument.

A generalized COND is mapped onto the following form:

$$(COND\ (p_1\ e_1)\ (p_2\ e_2\ e_3)\ \ldots\ (p_n\ e_n)) =$$
$$(p_1\to e_1,(p_2\to FL(e_2,e_3),\ldots(p_n\to e_n,NIL)\ldots))$$

We define the base predicates in LISP to be the functions EQ,ATOM, and EQUAL which are known to return T or NIL. All gbfs whose predicate part is not one of the previous, are replaced using the following transformation:

$(predicate\to conclusion,alternative) =$
$\quad (EQ(predicate,NIL)\to alternative,conclusion)$

All occurrences of these predicates in the premise position of the gbf are termed explicit occurrences. All other occurrences are termed implicit occurrences and are replaced by their equivalent via use of axiom (4) - i.e. predicate p is replaced by $(p\to T,NIL)$. This is motivated by the definition of a normal form where the propositional variables have now been replaced by the more general concept of a predicate.

Other forms of predicates such as AND, OR, NOT, etc. are converted to their conditional form representations.

An internal lambda of the form:

((LAMBDA (var$_1$ var$_2$ . . . var$_n$)
 <function body sequence>)
 <function body of var binding$_1$>
 <function body of var binding$_2$>
 .
 .
 .
 <function body of var binding$_n$>)

is represented by the form:

```
FL(SETQ(var ,<function body of var binding>),
           1                       1
   SETQ(var ,<function body of var binding>),
           2                       2
               .
               .
               .
   SETQ(var ,<function body of var binding>),
           n                       n
   FL(<function body sequence>))
```

Note that all lambda variables are given unique names to avoid errors at a later stage when bindings will be used instead of the variable names.

Another feature present in LISP which does not have an analog in [McCarthy63] is 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. The act of assignment is only temporary and thus is not a necessary component of the equivalence - i.e. 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 make use of the distributive law for functions and conditions. This means that certain computations, namely conditions, are moved so that physical position no longer indicates the sequence of computation. 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 computations predecessors have lower numbers and the successors have higher numbers (i.e. a partial ordering). This property can be achieved by numbering a conditional form in the following fashion: each time a number is assigned, it is higher than any number previously assigned.

(1) an atomic symbol is assigned a number.

(2) a function $f(arg_1, arg_2, \ldots, arg_n)$ is numbered in the order arg_1, arg_2, ..., arg_n, followed by assigning a number to f.

(3) a general boolean form $(p \rightarrow q,r)$ is numbered in the order p, q, r.

For example, consider the function UNION whose MLISP definition is given in figure 5. The function takes as its arguments two lists U and V and returns as its result a list of all the elements that appear in either list. Each element is assumed to occur only once in each list. Application of the function to the lists (A B C) and (D C B) results in the list (A D C B). Figure 6 contains the gbf representation corresponding to figure 5. Figure 7 contains the result of applying the above numbering algorithm to figure 6. Figure 8 contains the numeric and symbolic representations corresponding to figure 5 after application of the distributive law for functions. Notice that the test (EQ (MEMBER (CAR U) V) NIL) appears before the computation (CDR U) in figure 8.

```
UNION(U,V) = if NULL(U) then V
          else UNION(CDR(U),
                   if MEMBER(CAR(U),V) then V
                   else CONS(CAR(U),V))
```

Figure 5 - MLISP Encoding of UNION

```
((EQ U NIL)→V,
         (UNION (CDR U)
               ((EQ (MEMBER (CAR U) V) NIL)
                →V,
                 (CONS (CAR U) V))))
```

Figure 6 - Symbolic GBF Representation Corresponding to Figure 5

```
((14 10 12)→16,
         (44 (20 18)
            ((32 (28 (24 22) 26) 30)
             →34,
              (42 (38 36) 40))))
```

Figure 7 - Numeric GBF Representation Corresponding to Figure 5

```
((EQ U NIL)→V,
         ((EQ (MEMBER (CAR U) V) NIL)
          →(UNION (CDR U) V),
           (UNION (CDR U) (CONS (CAR U) V))))
```

```
((14 10 12)→16,
         ((32 (28 (24 22) 26) 30)
          →(44 (20 18) 34)
           (44 (20 18) (42 (38 36) 40))))
```

Figure 8 - Numeric and Symbolic Representations Corresponding to Figures 6 and 7 After Application of the Distributive Law for Functions

The algorithm for obtaining the normal form is only briefly presented. It has two phases each of which processes the symbolic and numeric representations of the program in parallel. The first phase 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 second phase corresponds to making use of axioms (2),(3),(5), and (6) to get rid of duplicate occurrences of predicates as well as redundant computations. The latter is the case when a computation such as (CAR U) in figures 3 and 5 is computed more than once along a computation path with no intervening computations that might cause the two instances to have different values (i.e. no computations having side-effects). Figure 9 contains the intermediate representation corresponding to figure 8. Note that all occurrences of the same local variable have the same computation number. This computation number is less than the computation numbers associated with any function. Also, by convention we assign NIL the computation number 0.

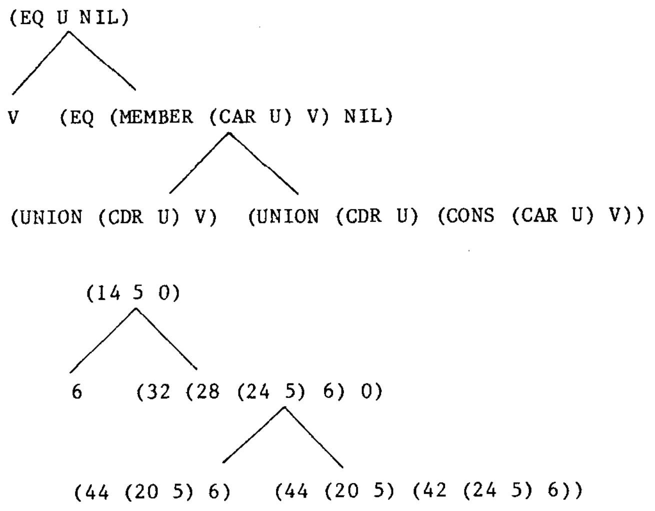

Figure 9 - Normal Form Corresponding to Figure 8

Briefly, some other steps are necessary to properly deal with assignments to SPECIAL variables and side-effect computations such as RPLACA and RPLACD. We use a mechanism that views these operations as two-parted. One corresponds to the act of assignment and the other to the process of returning a value. We decouple these two components. This enables us to be able to deal with cases where acts of assignment are redundant as is the case when a value is assigned to a SPECIAL variable without any access of the said variable prior to a subsequent assignment. In this case, the act of computing the new value being assigned in the first case is not redundant while the actual act of assignment is redundant.

We have seen that the numeric representation of a LISP function has the property that associated with each constituent computation is a number which is greater than the numbers associated with the computation's predecessors and at the same time less than the numbers associated with the computation's successors. This approach, henceforth referred to as breadth first, was necessary in order to properly execute the binding and condition expansion part of the normal form algorithm (also the only possible numeric representation prior to its execution). The duplicate computation removal phase, and more importantly the matching phase (i.e. the proof of the equivalence of the representations corresponding to the source and object programs), requires an even stronger criterion. We wish the numeric representation to have the afore-mentioned properties plus the property that all computations with the same computation number have been computed simultaneously. By simultaneously, we do not necessarily require computation at the same location. This criterion is not strong enough. Basically, what we are after is the following: If two identical computation numbers appear in two different subtrees, then the functions associated with them must have been computed with the same input conditions and equivalent arguments. For example, recall the following fragment from figure 6:

```
(UNION (CDR U)
       ((EQ (MEMBER (CAR U) V) NIL)
        →V,
         (CONS (CAR U) V)))
```

The numeric representation assigned to this form was:

```
(44 (20 18)
    ((32 (28 (24 22) 26) 30)
     →34,
      (42 (38 36) 40)))
```

After applying the first two phases of the normal form algorithm we have:

```
((EQ (MEMBER (CAR U) V) NIL)
 →(UNION (CDR U) V),
  (UNION (CDR U) (CONS (CAR U) V)))
```

with the numeric representation:

```
((32 (28 (24 5) 6) 0)
 →(44 (20 5) 6),
  (44 (20 5) (42 (24 5) 6)))
```

Note that the same computation number, 44, is associated with the two instances of UNION. However, the function UNION yields different results for the two instances since in one case the second argument is V and in the other case the second argument is (CONS (CAR U) V). Thus we wish to have different computation numbers for the two instances of UNION. Of course, if V and (CONS (CAR U) V) were equivalent (impossible in this case), then the two instances of UNION could have the same computation number; the proof procedure deals with such questions.

Recalling our characterization of the normal form as a tree, we see that the numbering scheme that we require, known as depth first, has the property that all computations performed solely in the right subtree have a higher computation number associated with them than the numbers associated with computations performed solely in the left subtree. In fact, this is the basis of the algorithm used to convert a breadth first numeric representation to a depth first numeric representation.

Thus for the above example we would want the following numeric representation:

```
((32 (28 (24 5) 6) 0)
 →(44 (20 5) 6),
  (48 (20 5) (46 (24 5) 6)))
```

and figure 10 is the numeric representation corresponding to figure 9.

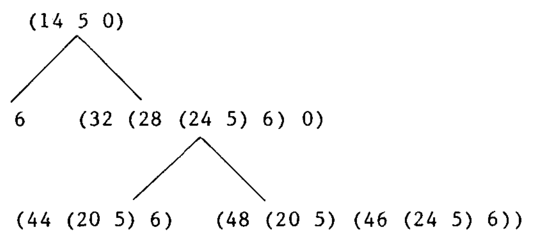

Figure 10 - Depth First Renumbering of Figure 9

The duplicate computation removal algorithm processes the symbolic and numeric representations of the program in order of increasing computation numbers. The main objective of the algorithm is to remove predicates whose values are known and to replace duplicate occurrences of computations by their first occurrence. The tree-like nature of our representations coupled with the property that all computations performed solely in the left subtree have lower computation numbers associated with them than with those performed solely in the right subtree greatly facilitates our work. We also make use of the fact that application of axiom (7) coupled with the manner in which the distributive law for functions was applied preserved the order in which conditions were tested - i.e. each predicate has a lower computation number associated with it than is associated with the predicates computed in its subtrees.

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. All that remains is to prove that the two representations are identical or that either one can be transformed using our axioms into the other. This procedure proceeds by attempting to prove that each computation appearing in one of the representations appears in the other and vice versa. This is accomplished by uniformly assigning the computation numbers in one representation, say B, to be higher than all of the numbers in the other representation, say A, and then, in increasing order, search B for matching instances of computations appearing in A. In the proof liberal use is made of axioms (1),(2),(3),(5), and (6) as well as substitution of equals for equals.

For example, figures 11 and 12 contain the numeric intermediate forms corresponding to figures 3 and 4 respectively. We assume that duplicate computation removal has already been applied and thus (CAR U) has the same computation number in both instances in figures 11 and 12. Note also that we have chosen to make all computation numbers in figure 12 higher than those in figure 11. Despite the apparent similarity of the two representations, there is a problem. The proof system must prove that (INTERSECTION (CDR U) V) can be computed simultaneously and before the test (MEMBER (CAR U) V). In other words, we must be able to prove that the act of computing (MEMBER (CAR U) V) can be postponed to a point after computing (INTERSECTION (CDR U) V). Thus we are proving the correctness of a factoring-like optimization. Once it is shown that figure 12 can be transformed to yield figure 11, the process is repeated by assigning to the computations in figure 11 higher numbers than those in figure 12 and proving that figure 11 can be transformed to yield figure 12.

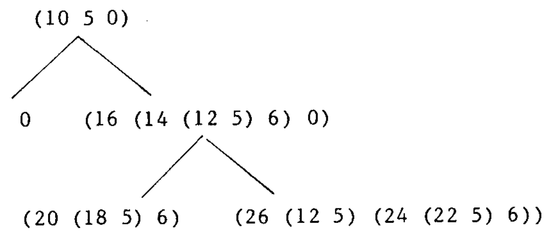

Figure 11 - Numeric Representation of Figure 3

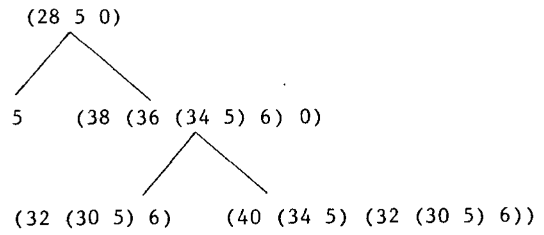

Figure 12 - Numeric Representation of Figure 4

6. CONCLUSION

We have seen a formalism for representing a high level language program so that a proof procedure could be used for proving correctness of translation. Our technique has been illustrated by use of a sample language LISP. Indeed many of the steps have been heavily influenced by LISP and its structure. Nevertheless, we feel that many of these techniques will be useful in attempting such proofs for other languages. Future work should be focussed on identifying inadequacies in our formalism so that more complex languages can be handled. In [Samet75] modifications are proposed for handling more complicated LISP constructs such as PROG, GO, and RETURN.

7. REFERENCES

[BurstallDarlington75] - Burstall, R.M., and Darlington, J., "Some Transformations for Developing Recursive Programs," Proceedings of the 1975 International Conference on Reliable Software, April 1975, pp. 465-472.

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

[Deutsch73] - Deutsch, L.P., "An Interactive Program Verifier," Ph.D. Thesis, Department of Computer Science, University of California at Berkeley, May 1973.

[Floyd67] - Floyd, R.W., "Assigning Meanings to Programs," Proceedings of a Symposium in Applied Mathematics, Volume 19, Mathematical Aspects of Science, (Schwartz, J.T. ed.), American Math Society, 1967, pp. 19-32.

[Gerhart75] - Gerhart, S.L., "Correctness Preserving Program Transformations," Second ACM Symposium on Principles of Programming Languages, January 1975, pp. 54-66.

[Huang75] - Huang, J.C., "An Approach to Program Testing," ACM Computing Surveys, September 1975, pp. 113-128.

[King69] - King, J., "A Program Verifier," Ph.D. Thesis, Department of Computer Science, Carnegie-Mellon University, Pittsburgh, Pennsylvania, 1969.

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

[London71] - London, R.L., "Correctness of Two Compilers for à LISP Subset," Stanford Artificial Intelligence Project Memo AIM-151, Computer Science Department, Stanford University, October 1971.

[London72] - London, R.L., "The Current State of Proving Programs Correct," in Proceedings of the ACM 25th Annual Conference, 1972, pp. 39-46.

[Loveman76] - Loveman, D.B., "Program Improvement by Source to Source Transformation," Third ACM Symposium on Principles of Programming Languages, January 1976, pp.140-152.

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

[McCarthy63] - McCarthy, J., "A Basis for a Mathematical Theory of Computation," in Computer Programming and Formal Systems (Eds. Braffort and Hirshberg), North Holland, Amsterdam, 1963.

[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," in Proceedings of the ACM 29th Annual Conference, 1976, pp. 492-497.

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

[Suzuki75] - Suzuki, N, "Verifying Programs by Algebraic and Logical Reductions," Proceedings of the 1975 International Conference on Reliable Software, April 1975, pp. 473-481.

[Wegbreit76] - Wegbreit, B., "Goal-Directed Program Transformation," Third ACM Symposium on Principles of Programming Languages, January 1976, pp. 153-170.