COMPILER TESTING VIA SYMBOLIC INTERPRETATION*

by

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

Abstract

A method for compiler testing using symbolic interpretation is presented. This method is a cross between program proving and program testing. It is useful in demonstrating that programs are correctly translated from a high level language to a low level language thereby improving the reliability of the compiler. The term symbolic interpretation is used to describe the process of obtaining an intermediate form of the low level language program that is suitable for further processing by a proof system. Symbolic interpretation is the heart of the system and enables the recording of a transcript of all computations in the program. This process interprets a set of procedures which describe the effects of machine language instructions corresponding to the target machine on a suitable computation model. The highlights and limitations of the process as well as future work are discussed in a framework of a specific LISP implementation on a PDP-10 computer.

Keywords and Phrases: compilers, program proving, program testing, correctness, symbolic interpretation, machine description languages, program verification, LISP

1. INTRODUCTION

Given a compiler, or for that matter any program translation procedure, a question that often comes to mind is how accurate are the resulting translations. We are interested in a testing tool for proving the correctness of translations performed by translators which do a considerable amount of code optimization. Some possible approaches to this problem include program proving[10], program testing[6], and decompilation[5].

Most of the work in correctness from the program proving approach has dealt with specifying assertions[4] about the intent of a 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 ([3],[16]), and even if a program is found to satisfy the stated assertions there is no guarantee that the assertions were sufficiently precise to account for all contingencies (i.e. there is considerable difficulty in specifying machine dependent details such as overflow, precision, etc.). This difficulty is compounded when the programs are of such a complexity that they defy formal analysis (e.g. 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[11] compiler[9].

Program testing is a concept which has been gaining an increasing amount of attention in recent years. This is in part due to a realization that formal program proving methods rely on a very powerful theorem proving capability which is unlikely to appear in the near future. Program testing in its current state can be used to discover the presence of errors but not their absence. Nevertheless, program testing has several advantages over program proving. These include a capability for scrutinizing machine dependent details, and greater ease in specifying program correctness - at least on a conceptual level. Namely, we need only consider matching input output pairs. However, this may lead to having to test the program a very large number of times. This leads us to a need to develop test criteria. The first criterion that comes to mind is that every code unit should be executed at least once. This test is inadequate for two reasons. First, programs with inaccessible code segments fail to pass this test. Secondly, and more importantly, a program may be constructed in such a way that a test may exercise every code unit, yet not every path in the program will be tested. This leads to a stronger criterion which states that every branch in the flowchart must be traversed at least once. Given that a suitable test criteria has been found we are still faced with a realization that test case generation is a considerable problem in its own right.

Decompilation methods could conceivably be used to verify the equivalence of a source program and an object program. Such methods are designed to return a representation of the object program in a format identical to the source program. These methods operate by searching for a syntax in the assembly language program. This is much akin to pattern recognition. The trouble with such methods is that they imply that the decompiling program must know how the various constructs of the high level language are encoded in the low level language. This sets a limit on the variation in the object code that can be presented to such a system. A more serious flaw is the fact that compilation is a many to many process. Namely, the object program corresponding to a program written in a high level language can be encoded in many different equivalent ways. Similarly, to an object program there corresponds more than one equivalent source program. This is because most high level programming languages have built in redundancies that allow duplication or non-unique program specification (e.g. internal lambda construct in LISP).

2. PROGRAM TESTING WITH RESPECT TO COMPILERS

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. This willingness enables us to make use of the notion of program testing to achieve our goal. Note that we will be testing the compiler and not the translations. Thus there is no need to test the translated

*This work was supported in part by a General Research Board Faculty Award of the University of Maryland.

program by applying all possible inputs. Instead, the test case generation problem reduces to testing the compiler by applying all possible inputs to it (i.e. all possible programs in the high level language of issue). At first this seems an insurmountable task. However, closer scrutiny reveals that the proof procedure can be embedded in the compiler as part of the translation process. In such a case the issue of whether or not all possible inputs have been tested disappears since in fact we are only interested in the correctness of the translation process with respect to the set of programs being translated. In brief we have bootstrapped 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.

At this point it is appropriate to define our notion of program testing for a compiler or, more generally, any translation process be it mechanical or manual. The 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 relies on the existence of such a representation. 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[8], that is they have identical execution sequences except for certain valid rearrangements of computations. Note that this is a more stringent requirement than that posed by the more conventional definition which holds that two programs are equivalent if they have a common domain and range and both produce the same output for any given input in their common domain. 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 the notion of sorting is an input output pair characterization of an algorithm.

The actual testing procedure consists of three steps. First, the high level language program must be converted to the intermediate representation. Second, the low level language program must be converted to the intermediate representation. Third, a check must be performed of the equivalence of the two representations. This check may take the form of a procedure which applies valid equivalence preserving transformations to the results of the first two steps in attempting to reduce them to a common representation.

In the remaining discussion we expand on the second step of the testing procedure. This step is termed "symbolic interpretation" and denotes a technique for obtaining a symbolic representation of the low level program which reflects all of the computations performed on all possible program execution paths rather than just one as in symbolic execution[7]. The technique differs from decompilation methods since their use in establishing equivalence yields a syntactic equivalence between the decompiled version of the low level program and the original version of the high level program. However, symbolic interpretation is based on the run-time equivalence of computation sequences. The representation obtained by symbolic interpretation is compatible with the result of the first step of the testing procedure and combines with it to form an input to the third step. Thus we see that the test criteria problem alluded to in the discussion of program testing is solved by the use of the notion of symbolic interpretation coupled with our definition of equivalence. In addition, the absence of errors will mean that the program has been correctly translated thereby removing the objection raised earlier to program testing that it is only good for detecting errors.

3. SYMBOLIC INTERPRETATION

The symbolic interpretation process builds an intermediate representation for the low level program by activating a set of procedures corresponding to the instructions in the low level program in a manner consistent with the execution level definition of the high level language (quite similar to interpretation). These procedures specify how each instruction effects an entity known as the computation model (e.g. procedural embedding[17]). This model reflects the contents of the various data structures relevant to the execution of the program as well as the values of the conditions tested. Thus there is a need for a capability to describe a computer instruction set. This description must provide for data types as well as a control structure for the symbolic interpretation process. By control structure we mean the ability to invoke various parts of the assembly language program, as is the case when processing a condition, a branch, or a function call.

In the following three sections we describe the type of information that comprises the computation model, a control structure for the symbolic interpretation process, and an example. However, in order to have some framework for the discussion, we must assume the existence of a suitable programming language and an execution level definition for the language. Our high level language is a subset of LISP[12] which has been shown to have a suitable intermediate representation[14] in the form of a tree. The low level language is LAP[12] (a variant of the PDP-10[2] assembly language). An actual proof system employing the ideas discussed here is described in [13].

Briefly, we are dealing with a subset of LISP which 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 functions 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.

3.1 COMPUTATION MODEL

In order to be able to symbolically interpret a low level language program we need both an execution level definition of the high level language as well as a set of data structures to record the effects of the various operations. As the symbolic interpretation process proceeds along a given execution path, it must maintain a record of the various assumptions it has made as to the results of condition testing instructions so that subsequent tests of related conditions can be recognized. This is accomplished with the aid of an equality data base. We must also maintain an uptodate model of the contents of all directly accessible memory locations as well as a record of all computations that have been performed so that true equivalence can be determined.

The equality data base is a set of equivalence classes for all values computed in the program (e.g. for LISP this includes functions as well as atoms) where transitivity and functional application are fully propogated. In addition, built into this data base is knowledge of the full implication of basic constructs of the high level language. For example, in the case of LISP this includes the basic predicates EQ, EQUAL, ATOM and their interrelationships; commutativity of arithmetic operations such as PLUS and TIMES; antisymmetry of operations such as CONS and XCONS and LESS and GREAT (e.g. A<B is equivalent to B>A). The actual test for equality or inequality of two operands consists of parsing them and checking if they are members of the same equivalence class; if not, then the two operands are assumed to be equal and if a contradiction is obtained during the process of propogating the equality through the data base, then the two operands are known to be unequal.

In the example LISP execution level definition, memory consists of all of the accumulators, a stack, the consecutive block of words containing the object code corresponding to the function being symbolically interpreted, and cells containing the values of the global variables. Each cell in memory has two halves - left and right. A LISP cell is said to occupy one full word where the left

half contains CAR and the right half contains CDR. Thus it should be clear that the act of accessing the left half of a LISP cell corresponds to computing CAR and similarly for the right half of a LISP cell and CDR. We use a wide set of data types to describe the contents of memory cells. These types include LISP pointers (all of the locations containing the parameters to the function being symbolically interpreted are initialized to the symbolic names of their corresponding parameters), stack pointers, data (non-LISP numbers and symbolic addresses), and others.

In order to be able to demonstrate equivalence between the high and low level programs we must be able to show that all computations performed in the high level program have also been performed in the low level program. Thus all computations in the low level program that involve LISP constructs are recorded. This is relatively straightforward since the LISP computations can be distinguished from overhead computations (e.g. stack pointer manipulation, etc.). The only possible stumbling block is in distinguishing between calculations of addresses and calculation of data. However, numbers have a distinct representation in LISP which is not the raw number (i.e. an atom since otherwise it would be difficult to differentiate between numbers and pointers) and appear in the program as (QUOTE number). Thus there is a separation between program and data which differs from the Von Neumann concept[1] of indistinguishability between the two. The task of recording the LISP computations is performed by the memory as well as by a list known as UNREFERENCED. As a program path is symbolically interpreted, UNREFERENCED contains a record of all computations that have been performed but do not occur as a subexpression of the contents of at least one memory location (i.e. their result or functions of their result are no longer accessible to future operations along the path). This is primarily for recording computations performed solely for their side-effect.

3.2 CONTROL STRUCTURES

The symbolic interpretation process must be able to cope with the basic control structures of a language. Some of the effects of these control structures are described explicitly in the procedures corresponding to the instructions in the low level program and others are implicit in the sense that when certain events are recognized by the symbolic interpretation process as having occurred, then corresponding actions are effected on the computation model. In this section we discuss what happens in case of a condition test, a branch, a function call, and encountering an instruction which has occurred previously on the path being symbolically interpreted.

As mentioned earlier, the symbolic interpretation process corresponds to interpreting a procedure for each instruction. For most instructions this consists of simply updating the computation model to reflect the interpretation of the instruction. For example, a HLRZ instruction is defined to load the right half of an accumulator with the left half of the contents of the effective address, and to clear the left half. This instruction is described in fig. 1 using MLISP[15], a variant of LISP, as the procedural language.

```
FEXPR HLRZ;
LOADSTORE(ACFIELD(ARGS),
          EXTENDZERO(
              LEFTCONTENTS(
                  EFFECTADDRESS(ARGS))));
```

Fig. 1 - HLRZ instruction

Some instructions may require more than one statement to describe their effect. For example, the POPJ instruction which is used to encode a return from a recursive call has a considerably longer description (see fig. 2). In brief, this instruction deallocates the stack entry which was used to store the return address, decrements the stack pointer, and returns control to the address stored in the address pointed at by the stack pointer prior to decrementing it.

```
FEXPR POPJ(ARGS);
BEGIN
  NEW LAB;
  LAB←RIGHTCONTENTS(RIGHTCONTENTS(ACFIELD(ARGS)));
  DEALLOCATESTACKENTRY(ACFIELD(ARGS));
  SUBX(<ACFIELD(ARGS),X11>);
  UNCONDITIONALJUMP(LAB);
END;
```

Fig. 2 - POPJ instruction

Until now the instructions that we have encountered describe explicitly how the computation model is to be updated. There are also instructions whose effects on the computation model are invisible insofar as their procedural definition. These are operations that result in function calls. In such a case the effect on the computation model is determined by the function being called and also by the assumed execution level definition of the language. In our case, upon a function call all accumulators but those that are known to have their contents unchanged by the execution of the said function are considered to have been overwritten with some unknown value. This causes their contents (if not previously referenced) to be added to a list of computations known as UNREFERENCED. Recall that this is how we represent computations executed primarily for their side-effect rather than their resulting value. When a function call occurs, the function being called is not symbolically interpreted (hence our finite tree representation); instead, the location which has been defined by the execution level definition of the language to contain the result is updated to indicate that it now contains the result of the said function applied to its arguments which are to be found in the a set of locations defined by the execution level definition of the language. The computation model is also updated to reflect any possible changes in the bindings of global variables and also to include any new equalities or inequalities that are implied by the execution of the function. For example, a (RPLACA A B) operation in LISP implies that subsequent to the instance of performance of the operation, B and (CAR A) point to the same list structure.

Some instructions perform control operations such as conditional branching as well as modify the computation model. Prior to explaining the role of symbolic interpretation in evaluating conditional branch instructions we must digress for a moment and define a predicate and the notion of a valid test. The basic type of non-arithmetic test that can be performed by a computer is a check for equality. All other non-arithmetic tests are modifications of this primitive using certain data structures. This equality test is a comparison against another value or zero. In LISP such tests translate into the predicate EQ having two operands. By valid test we mean that the two operands of the EQ predicate represent valid data items of the high level language. If not, then the value of the test must be known - e.g. address computation, etc. This means that there must be a suitable mechanism for converting the data structures used in the test to a corresponding meaningful test in the sense of the high level language. For example, instructions, that manipulate bits (e.g. TLNN) by checking if any bits (denoted by a suitable mask) in a word are one, must be capable of being converted, with the aid of knowledge about the execution level definition of the language, to a suitable test in the high level language. In this example, in an execution level definition of LISP which represents NIL by zero, the test would correspond to a check against NIL.

When conditional branching instructions are encountered, the symbolic interpretation process attempts to form a valid test 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, or the condition represents a test whose value has been determined earlier in the computation path. The latter is aided by the equality data base component of the computation model. If the condition is a valid test whose value is unknown, then the two alternate paths are evaluated in order and the result returned is a tree as shown in fig. 3.

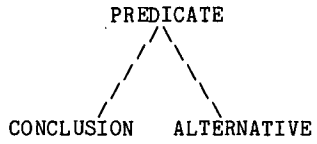

Fig. 3 - tree representation of a test

Prior to the evaluation of each path, the computation model is updated to reflect the assumed value of the condition. This includes modification of relevant memory locations as well as propogating equalities and inequalities, as the case may be, through the equality data base. This latter step is crucial to having the capability to recognize the occurrence of substitution of equals for equals.

An example of a conditional branch instruction is JUMPE (see fig. 4) which is used to branch to a specified location if the value of a specified accumulator is equal to zero. The description makes use of several control functions. CHECKTEST examines the operands and forms a valid test if possible. Next, if the value of the condition is already known, then appropriate action is taken. TRUEPREDICATE marks the sense of the test (an instruction branching on inequality with zero in this case would use FALSEPREDICATE). CONDITIONALJUMP and JUMPALTERNATIVE simply serve to recursively invoke the symbolic interpretation of the paths corresponding to the true and false cases of the condition. One of the parameters to these routines is the name of another routine which specifies any further processing that might be required prior to executing the path. Note that the actual construction of the tree corresponding to the result of the symbolic interpretation process occurs in JUMPALTERNATIVE.

```
FEXPR JUMPE;
BEGIN
  NEW TST;
  TST←CHECKTEST(CONTENTS(ACFIELD(ARGS)),ZEROCNST);
  IF TST THEN RETURN(
    IF CDR TST THEN
      UNCONDITIONALJUMP(EFFECTADDRESS(ARGS))
    ELSE NEXTINSTRUCTION());
  TRUEPREDICATE();
  CONDITIONALJUMP(ARGS,FUNCTION JUMPETRUE);
  JUMPALTERNATIVE(ARGS,FUNCTION JUMPEFALSE);
END;

FEXPR JUMPETRUE(ARGS);
UNCONDITIONALJUMP(EFFECTADDRESS(ARGS));

FEXPR JUMPEFALSE(ARGS);
NEXTINSTRUCTION();
```

Fig. 4 - JUMPE instruction

Whenever the symbolic interpretation process is about to interpret another instruction which has been previously encountered along the path being symbolically interpreted, then recursion is assumed to have taken place. 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. This means that the condition values along the path need not be tested since their values are known. If such a path from the start of the program exists, then it is unique since a condition cannot be both true and false.

3.3 EXAMPLE

The previous two sections served to highlight various aspects of the symbolic interpretation process. At this point it is appropriate to show how the symbolic interpretation process builds an intermediate representation.

Consider the function NEXT whose LISP1.6[12] and MLISP[15] definitions are given in fig. 5. The function takes as its arguments a list L and an element X. It searches L for an occurrence of X. If such an occurrence is found, and if it is not the last element of the list, then the next element in the list is returned as the result of the function. Otherwise, NIL is returned. For example, application of the function to the list (A B C D E) in search of D would result in E, while a search for E or F would result in NIL. Fig. 6 contains the LAP encoding for the function which was obtained by hand coding. Notice that the encoding is extremely compact - the inner loop is only four instructions long. This is minimal when we consider the fact that the inner loop consists of four operations - CAR, CDR, EQ test, and recursion.

```
(DEFPROP NEXT (LAMBDA (L X)
        (COND ((NULL L) NIL)
              ((EQ (CAR L) X)
               (COND ((NULL (CDR L)) NIL)
                     (T (CADR L))))
              (T (NEXT (CDR L) X)))) EXPR)

NEXT(L,X) = if NULL(X) then NIL
            else if CAR(L) EQ X then
                 if NULL(CDR(L)) then NIL
                 else CADR(L)
            else NEXT(CDR(L),X)
```

Fig. 5 - LISP and MLISP encodings of NEXT

When symbolically interpreting the example program, the first instruction that we encounter is JUMPE which is used to jump to label DONE if accumulator 1 contains a zero. The result is shown in fig. 7. Notice 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.

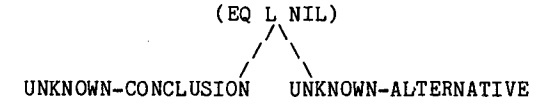

Fig. 7 - result of symbolic interpretation of (JUMPE 1 DONE)

The definition of JUMPE in fig. 4 calls for the path corresponding to the true case of the condition to be symbolically interpreted. This corresponds to updating the equality data base to reflect the equality of L and NIL followed by a branch to the instruction POPJ. At this point the current execution path is considered to be terminated since there is no return address on the stack corresponding to the current invocation of the recursive call. Thus in this case the control structure implicit in the symbolic interpretation process results in a mopup operation. This includes returning a value - L or NIL (they are equivalent at this point and the proof procedure which processes the intermediate forms corresponding to the high and low level programs will recognize this fact since built into it is the

LABEL	PROGRAM COUNTER	INSTRUCTION	COMMENT
NEXT	1	(JUMPE 1 DONE)	jump to DONE if L is NIL
LOOP	2	(HLRZ 3 0 1)	load register 3 with CAR(L)
	3	(HRRZ 1 0 1)	load register 1 with CDR(L)
	4	(CAIE 3 0 2)	skip if CAR(L) is EQ to X
	5	(JUMPN 1 LOOP)	if CDR(L) is not NIL then compute NEXT(CDR(L),X)
	6	(JUMPE 1 DONE)	jump to DONE if CDR(L) is NIL
	7	(HLRZ 1 0 1)	load register 1 with CAR(CDR(L))
DONE	8	(POPJ 12)	return

Fig. 6 - LAP encoding corresponding to NEXT

same equality data base mechanism as in the symbolic interpretation process). In addition, we must return a list of all of the computations that were performed but not referenced (i.e. UNREFERENCED). However, no such computations were performed. Fig. 8 shows the state of the intermediate representation after symbolic interpretation of POPJ. Note that only the true case of the test (EQ L NIL) has been resolved so far.

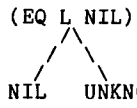

Fig. 8 - result of symbolic interpretation of (POPJ 12)

When symbolic interpretation is resumed we are in the false case of the condition (EQ L NIL) and the computation model is updated to reflect the fact that L is not NIL. The next two instructions, HLRZ and HRRZ, result in the updating of the contents of accumulators 3 and 1 to contain (CAR L) and (CDR L) respectively. In this example HLRZ loads the right half of accumulator 3 with the left half of the contents of the effective address (indexing via accumulator 1) and clears the right half of accumulator 3. HRRZ is similar to HLRZ except that the right half of the contents of the effective address is fetched instead of the left half. Note that nowhere in the procedural definition of HLRZ is there any indication that CAR is being computed. We are able to detect the computation of CAR by virtue of the act of fetching the left half of the contents of a LISP pointer.

CAIE is a condition testing instruction which compares the right half of the specified accumulator with the effective address and skips the next instruction if the condition is satisfied. It is described procedurally in a manner similar to JUMPE except for the addition of suitable primitives for effecting a skip rather than a jump. In our case this test corresponds to checking if (CDR L) is NIL and returning values of NIL and (CAR (CDR L)) for the true and false cases respectively. The intermediate representation prior to symbolically interpreting the false case of the (EQ (CAR L) X) condition is shown in fig. 9.

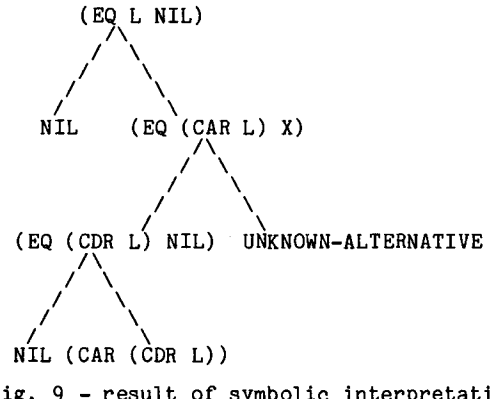

Fig. 9 - result of symbolic interpretation of true case of (CAIE 3 0 2)

The false case of the CAIE condition is interesting in several respects. The immediately following instruction is a conditional jump which in the true case proceeds to branch to an instruction that has been previously encountered, while in the false case we exit from the function. However, this exit takes advantage of the structure of the program to enable a tight encoding. This is accomplished by recognizing that the next instruction performs a test which is a no operation for the said execution path (i.e. a test of register 1 containing a 0). The no operation is easily detected by virtue of the equality data base mechanism which we recall keeps track of the values of the various tests encountered along the execution path. The branch to LOOP, a label previously encountered along the execution path, is interpreted as recursion in accordance with our earlier explanation of this concept.

The resulting intermediate representation is shown in fig. 10. Actually, there is an additional intermediate representation which indicates a relative ordering of computing the various functions. This is shown in fig. 11. Notice that the number corresponding to the CDR function in (CDR L) is less than that of EQ in (EQ (CAR L) NIL) despite its appearance in the tree below the said predicate. This ordering is necessary for the proof procedure to be able to adequately handle cases where the rearranging of the order of computing functions might lead to errors due to side-effect considerations. The relative magnitudes of the numbers only serve to indicate a partial ordering. The actual values of the numbers are used as indices into a table which indicates at what instruction and along which execution path each function was computed. This proves to be very handy in detecting where in an object program certain classes of errors occur.

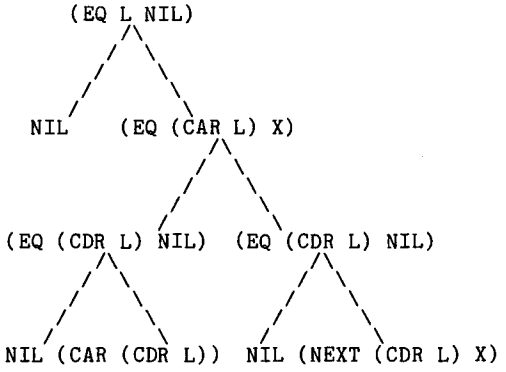

Fig. 10 - symbolic representation

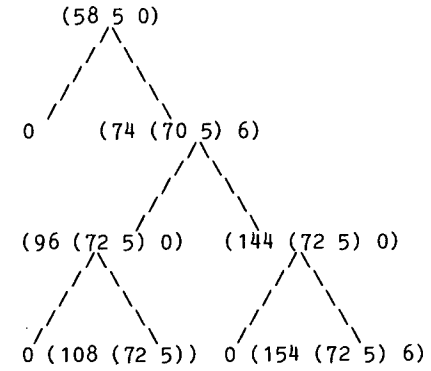

Fig. 11 - numeric representation

4. CONCLUSIONS

The use of symbolic interpretation as a means of obtaining the intermediate representation in the second step of the program testing procedure is the distinguishing factor between our system and decompilation[5] methods. We have seen that in our system, there was no need to specify how a particular construct is encoded since the internal representation is simply a record of computations performed. In other words, the system is built on the semantics of the various assembly language instructions in terms of their effect on a computation model (recall how the CAR and CDR operations were recognized).

A system[13] has been implemented which uses the ideas reported here to prove the correctness of translation of programs written in LISP1.6 to LAP. It was successfully used in proving the correctness of translation of a large number of programs many of which were hand coded for efficiency. This was possible because the system is independent of the actual translator. It only relies upon the execution level definition of the source high level language. In particular, the system was able to

locate errors in the translations as well as pinpoint in the object code the location where the error was made. This was accomplished with the aid of a numeric representation of the symbolic intermediate representation which recorded the value of the program counter and the path for each computation. These results suggest that the system would be particularly useful as a compiler debugger which is a resident part of the compiler. Proofs would be enabled when there is a reasonable belief that erroneous code is being generated. During this time compilation would proceed at a slower pace due to the additional burden of generating a proof; however, this is a small price to pay for the correctness assurance.

Some future extensions to the symbolic interpretation process include the following. Incorporate a more complete equality checking mechanism which would be able to cope with associativity as well as equalities in the arithmetic domain - i.e. at the present we can not detect the equivalence of x=1 and x-1=0. Currently, the system tries all possible paths. There is no way for the user to control the paths to be symbolically interpreted. Such a feature would be useful in a situation where certain execution paths are known to be erroneous and therefore are to be ignored. This is in contrast with the method of [7] which gives the user full control over the selection of paths to be explored. Another useful addition is a state save restore capability under the control of the user. This would mean that when errors occur, the symbolic interpretation process need not be started all over again.

ACKNOWLEDGEMENT

Special thanks go to Robert E. Noonan for his help in improving the presentation of the ideas set forth in this paper.

REFERENCES

[1] - Burks, A.W., Goldstine, H.H., and von Neumann, J., "Preliminary Discussion of the Logical Design of an Electronic Computing Instrument" in Computer Structures: Readings and Examples by Gordon Bell and Allen Newell, McGraw Hill, New York, 1971, pp. 92-119.

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

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

[4] - 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.

[5] - Hollander, C.R., "Decompilation of Object Programs," Ph.D. Thesis, Digital Systems Laboratory Technical Report No. 54, Department of Electrical Engineering, Stanford University, 1973.

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

[7] - King, J., "A New Approach to Program Testing," IBM Research Report RC 5037, Yorktown Heights, New York, September 1974.

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

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

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

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

[12] - 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.

[13] - 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.

[14] - Samet, H., "A Normal Form for LISP Programs," TR-443, Computer Science Department, University of Maryland, College Park, Maryland, February 1976.

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

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

[17] - Winograd, T., "Procedures as a Representation for Data in a Computer Program for Understanding Natural Language," MAC TR-84, Massachusetts Institute of Technology, February 1971.