[7] C.B. Weinstock, Dynamic Storage Allocation Techniques, Ph.D. thesis, Department of Computer Science, Carnegie-Mellon University, Pittsburgh, 1976.

[8] W.A. Wulf et al., *The Design of an Optimizing Compiler*, American Elsevier Publishing Company, New York, 1975.

[9] G. Yuval, Is Your Register Really Necessary?, *Software Practice and Experience* (Short Communications), to appear.

Appendix : Run-time system measurements

The five programs which were investigated were a program performing polynomial arithmetic on tree representations, a program performing a Quicksort, a program for the Choleski decomposition and one for the Householder triangularisation, and a program performing the LU decomposition of an array using rational arithmetic. We intend to obtain more complete figures from a wider range of programs later.

	Poly	Quicksort	Choleski	Triang	Rats
Number of run-time support routines called	64,000	8,000	64,000	100,000	150,000
Code sizes in words					
without targeting	2378	1386	2142	2494	3392
with targeting	1979	1084	1730	2084	3087
Number of assignations					
total	1450	170	2600	3170	6820
to variables	870	130	1380	1450	6260
Number of dereferences					
total	3190	660	3430	3660	13840
of variables	1570	400	2400	2100	11940
Assignations to elements of multiples					
pointer count=1 and creation counts match	40	40	1230	1690	310
creation counts do not match					
but no need to copy	0	0	330	190	140
copying required	0	0	80	50	0
Number of multiples dereferenced	60	0	530	600	300

INCREASING THE RELIABILITY OF CODE GENERATION

by

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

Abstract

A method to improve the reliability of code generation is proposed. Particular attention is focused on delegating the code generation and its verification to different individuals and on a two stage compilation process with a capability for enabling and disabling proofs of correctness of translation. Application of such a method to a compiler written in itself results in a proof of the correctness of the bootstrapping process. The highlights of a proof are illustrated by use of an example.

Keywords and Phrases: compilers, code generation, bootstrapping, program testing, program correctness

1. BACKGROUND AND MOTIVATION

In recent years much attention has been focused on the problem of proving the correctness of programs[8]. In [11] a methodology is described for dealing with the problem of proving that a program written in a higher level language, a subset of LISP[9], is correctly translated to LAP, which is a variant of the PDP-10[1] assembly language. In order to demonstrate the feasibility of this methodology a practical system was constructed which performed beyond expectations. Part of this work involved identification of the semantic properties of LISP as well as development of a formalism to describe the assembly language instructions in such a way that the description, which was procedural, could be executed to obtain a representation of the assembly language program[13].

In this paper we are interested in improving the reliability of code generation involving optimized code. We attack this problem with a combination of program testing[5] and program correctness [8] methods. Most of the previous work in the area of correctness has been along the lines of assertions ([3],[6],[15]) about the intent of the program which are then proved to hold. The difficulties with such methods are numerous. Most notable are the problems encountered in specifying the assertions[2] and the actual proof methods. Proofs using such methods reduce to showing that a set of assertions hold. However, when examining such proofs we must allow for the possibility that the assertions are 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. We feel that it is justifiable in proving equivalence between algorithms. Nevertheless, 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. As an example of a problem in which use is made of the purpose of the program, consider two methods of computing the greatest common divisor. In such a case we have defined an input-output pair relationship (i.e. the greatest common divisor) and we wish to determine if the two algorithms actually yield the same results for all possible inputs. The problem of proving the equivalence of two different algorithms is known to be unsolvable in general by use of halting problem-like arguments. We do not deal with such problems here.

Notice that we address ourselves to proving the correctness of the translation. One method of achieving this is to prove that the translator (e.g. a compiler) is correct - i.e. to prove that there does not exist a program which is incorrectly translated by the compiler. In

is case we would revert to the intent characterization of correctness t forth in the previous paragraph. Instead, we use a testing approach prove for each program input to the translation process, that the anslated version is equivalent to the original program. Thus, we are t saying anything about the general correctness of the translation ocess. A proof will have to be generated for each input to the anslation process. However, this has several important advantages, pecially when the translator is a compiler. First, as long as the ompiler does its job for each case input to it, then its correctness is a secondary nature - i.e. we will have bootstrapped ourselves to the tate where we can attribute an effective correctness to the compiler. econd, the proof process is independent of the compiler. The latter eans that if another compiler were used, no difference would result. his implies that programs could be hand compiled or translated. This s quite important and identifies the proof as belonging to the emantics of the high and low level languages in which the input and utput respectively are expressed rather than to the translation rocess. 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 proof would rely on the identification of all the possible input output pairs for code sequences. This is the type of approach taken in the proof of the correctness of LCOM0 and LCOM4 [7].

The proof system employed is not based on an assumption of an existence of a unique relationship between the source code and the object code. Compilation is viewed as a many-to-many process - i.e. there is no one-to-one relationship between source code and object code. Thus there is no reflection of the source level syntax in the object code as is common in decompilation[4] systems which attempt to reconstruct a program from the object code. We make no such attempts at reconstructing the program. Instead use is made of an intermediate representation of the program[12] 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.

2. METHOD

Our goal is to use the results of [11] to improve the reliability of code generation. This topic falls in the domain of software engineering [16] - a rather broad term recently adopted for a field which serves as a bridge between theory and practice. At the present most of the

results have been in the domains of reliability and user interfaces. Our approach to the reliability of compilers consists of providing a double check for code generation. Historically, this has taken on the flavor of a test of some known functions by checking input output pairs. However, we would like to diverge from this path via the following scenario:

Suppose that a compiler for a high level language, say L, is to be constructed for a machine, say M. The traditional approach has beer to decide upon an implementation and then do one of the following:

1. Modify the code generators of an existing compiler for language L to machine M′ to generate code for the target machine M or:

2. Write a new compiler for language L and machine M.

The actual verification stage has usually been left to the user community once the compiler has been deemed relatively correct. Formal methods, with the exception of [7], have not generally been the rule.

We propose a two person programmer team. Both members would initially decide on the implementation for the language L on the target machine M. Results of such decisions generally will include the type of calling sequences to be used, array implementation, location of results from function calls, etc. Once this has been accomplished, one member of the team proceeds to write the compiler and the code generators in a manner outlined in items 1 and 2, while the other member of the team will provide a description of machine M in a manner consistent with language L and the decided implementation. This description procedure is in a form comparable to that presented in [11].

Such a system should result in providing a greater degree of flexibility in debugging code generators. This debugging process will proceed in the following manner:

1. Check the compiler on standard programs.

2. Release the compiler to the user community with the proof procedure enabled. This will result in a slower compilation time, but users will be guaranteed that their programs have been properly compiled.

3. When the compiler has been deemed relatively error-free, the proof feature can be disabled thereby yielding a faster compilation time.

4. If errors are detected in subsequent operation, go back to 2.

The advantages of this method of verification are two-fold. First, we

have a means of providing an independent check on the correctness of the code generation process since the person responsible for the machine description phase of the project need not, in general, know how each construct in the language L will be encoded in the assembly language of machine M. This will eliminate many of the unconscious prejudices which usually enter into proofs of theorems (or correctness of programs) composed by the person performing the proof. Second, the user also benefits from the proof procedure employed since once an error is found and fixed, he may continue to use the compiler (with the proof procedure enabled) with confidence that if any more errors occur, then they will not go undetected. This is vastly preferable to waiting for a proof, by whatever means, that the modifications made to the compiler to fix one error, did not introduce other errors. The previous class of errors is far more common than one would imagine. An even worse fate is when the awaited proof is another complaint taking the form of "why did my program blow up?"

An additional benefit of our method is that it provides a means of proving the correctness of the bootstrapping process. A common compiler construction approach is to write a compiler, C, for high level language L to machine language M using L. This relies on L being available on some machine M′ via a compiler, say C′. The next step is to let C′ compile C to yield C′′, a compiler written in machine language M. Using our methods we can prove the equivalence of C′′ and C. Thus equivalent code will be generated whether the compiler is executed on machine M or on machine M′. This result finds important applications when software is developed on one computer to be eventually used on another computer. As a result we are enabling the debugging of a compiler on the machine where it is first implemented without requiring the presence of the target machine.

In summary, we have presented a two step method:

1. Two man design team.

2. Two stage compiling process.

As a result of using this method we hope to determine the following: First, is reliability indeed improved by use of the two man teams. This can be best assessed by examining the type of errors detected by the proof procedure. Second, the frequency of shuttling between steps 2 and 3 of the debugging procedure should give an indication of the worthiness of the two stage compilation process. A high frequency will indicate an overall saving to the user since he is assured of a more reliable compiler by virtue of not having to worry about the correctness of the translation of his program unless it has indeed been incorrectly translated. This is a problem that plagues all users except possibly the implementors.

3. EXAMPLE

As stated previously a proof system[11] using the techniques proposed in this paper already exists for programs written in a subset of LISP 1.6[10] and LAP[10]. In this section we give a sample LISP program, a LAP encoding for this program, and some highlights of a proof.

Consider the function NEXT whose LISP1.6[10] and MLISP[14] definitions are given in fig. 1. 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.

```
(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. 1 - LISP and MLISP encodings of NEXT

Fig. 2 represents one possible LAP encoding corresponding to the LISP program of fig. 1. The format of a LAP instruction is (OPCODE AC ADDR INDEX) where INDEX and ADDR are optional. OPCODE is a PDP-10 instruction. The AC and INDEX fields contain numbers between 0 and 15 and denote accumulators. ADDR denotes the address field. Our implementation assumes that NIL is represented by zero and that a LISP cell occupies one full word where the left half contains CAR and the right half contains CDR. The stack is used for passing control between functions. A function of n variables expects its parameters in accumulators 1-n. Accumulator 1 is used to return the result of the function.

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. 2 - LAP encoding corresponding to NEXT

Our proof system uses an intermediate representation which is a tree (see fig. 3). The root denotes a predicate. The left and right subtrees correspond to the true and false cases respectively of the predicate. The same intermediate representation is used for both the assembly language program and the high level language program.

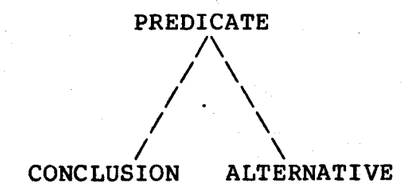

Fig. 3 - tree representation of a test

The intermediate representation for the assembly language program is obtained by use of a process termed "symbolic interpretation." This process symbolically interprets procedural descriptions of instructions to yield the resulting intermediate form. Fig. 4 contains a few sample instruction descriptions using MLISP. HLRZ is used 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. POPJ is used to encode a return from a recursive call. Its action includes the deallocation of the stack entry containing the return address, the decrementing of a stack pointer, and the return of control to the calling function. JUMPE is a branch on a zero in the designated accumulator. This instruction also performs a control operation. First, its description provides a capability for symbolically interpreting both the true and false parts of the condition - provided its value is not already known (e.g. instruction 6 when entered via instruction 5 in fig. 2). Second, upon

termination its description provides for an implicit construction of a tree (the primitive JUMPALTERNATIVE).

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


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


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 - example instruction descriptions

Fig. 5 shows the result of the symbolic interpretation of the true case of instructions 1 and 8.

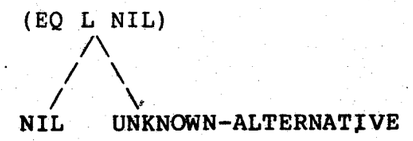

Fig. 5 - result of symbolic interpretation of (POPJ 12)

The final intermediate representation of the LAP encoding of fig. 2 is shown in fig. 6. Notice that we used the occurrence of recursion to halt the symbolic interpretation process. In general recursion is detected when an instruction is interpreted along an execution path which has been previously encountered along the path (e.g. instruction 2 via instruction 5 in fig. 2). In such a case the proof system must prove that the same result would have been obtained had the instruction been reached via the start of the program. The actual proof consists of applying transformations to fig. 6 and the intermediate form of fig. 1 to make them identical.

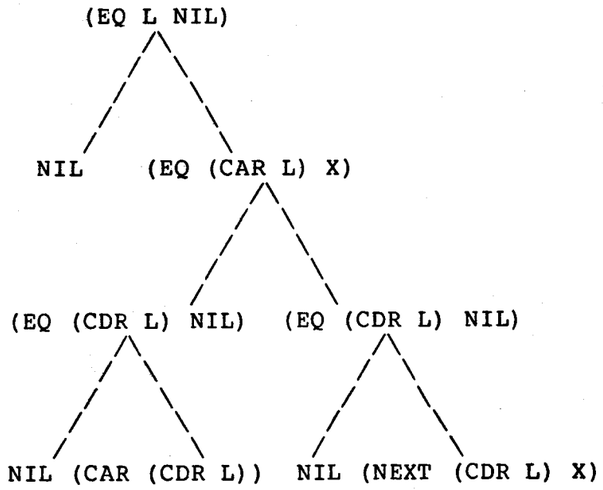

Fig. 6 - intermediate representation

4. REFERENCES

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

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

[3] - 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.

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

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

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

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

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

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

[10] - 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.

[11] - 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.

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

[13] - Samet, H., "Compiler Testing via Symbolic Interpretation," to appear in Proceedings of the ACM 29th Annual Conference, 1976.

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

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

[16] - Yeh, Raymond T., "Editor's Note," IEEE Transactions on Software Engineering, March 1975, pp. 1-6.