INFORMATION PROCESSING 77, B. GILCHRIST, EDITOR

A NEW APPROACH TO EVALUATING CODE GENERATION IN A STUDENT ENVIRONMENT*

HANAN SAMET
University of Maryland
College Park, Maryland

A new approach to the evaluation of code generation and optimization in a student environment is presented. This approach relies on a validation procedure for demonstrating the correctness of the translation. The motivation for this approach, a brief description of its theoretical foundations, an example, and a discussion of the types of errors that can be detected through the use of such a system are set forth in the context of an existing system.

1. INTRODUCTION

Increasingly, students are expected to be well versed in all aspects of computer science. A particularly popular course is one dealing with compiler construction ([1],[2]). Such a course has traditionally focussed on theory coupled with the construction of a complete compiler for a substantial language. Most often the course [3] covers symbol tables, lexical analysis, parsing, code generation, and code optimization. Since the emphasis is on "doing," care must be exercised in the choice of a high level language and a low level language since on the one hand they must be of a sufficient complexity so that various phases of the compilation process are non-trivial; yet, on the other hand the combination of the languages must not overwhelm the student.

Unfortunately, most compiler construction courses devote most of the time to parsing with the result that little attention is paid to code generation and optimization. These topics are generally resolved, if at all, by generating code for either an idealized stack machine or a one register machine [4]. The result is that the student is not exposed to a realistic code generation situation since the limited nature of the target machine results in standard code sequences for the various constructs of the high level language. Thus the student has virtually no chance in displaying his creativity or to employ optimizations that take advantage of the architecture of the target machine.

One of the main problems that plague the teaching of code generation is the difficulty of evaluating whether the code that has been generated is correct. Unlike parsing, the correctness of code cannot be easily "eyeballed." The student is confronted with two problems. First, he must decide, or learn, how to generate code, and, secondly, he must debug the generated code. These are two distinct problems. The second problem cannot be solved by applying a common testing technique of executing the object program with sample data. This is not sufficient since correct execution of the translated program does not guarantee the correctness of the translation. The previous solution has an additional shortcoming. When applying test data to a program (e.g., pairs of integers to a program that computes the greatest common divisor), the source of the resulting error can not be easily ascertained. The problem is that the error could either be in the program or in the compiler. Thus we see that in order to test a compiler, the test data must consist of source programs. In other words, we wish to demonstrate that the source program has been correctly translated so that all subsequent executions of the resulting object program yield identical results as the source program.

In the following sections, we present a new approach to evaluating code generation in a student environment which relies on proving the correctness of the translation. We give a brief background of the underlying theory followed by an example. Next, we discuss the types of errors that can be detected through the use of such an approach. We conclude our presentation with some suggestions for future work.

2. PROPOSAL

In [5] a formalism is reported which is used to prove that programs written in a high level language are correctly translated to assembly language. The formalism is particularly noteworthy because a significant amount of optimization as well as hand code can be handled. More importantly, the proof system is independent of the actual translation process. In fact, this independence enhances its usefulness as an evaluation tool for code generation and optimization in a student environment. By being able to cope with any translation process, a single system can handle each student's compiler.

The actual validation process consists of proving equivalence between a program input to the compiler and the corresponding translated program. This is accomplished by finding an intermediate representation which is common to both the original and translated programs and then checking for equivalence. Of course, we assume that such a representation exists. By equivalence we mean that the two programs are capable of being proved to be structurally equivalent [6] - i.e., they have identical execution sequences except for certain valid rearrangements of computations. No use is made of the purpose of the program in the process of proving equivalence. Thus, for example, we can not prove that a high level sorting program using insertion sort is equivalent to a low level sorting program using quicksort since the notion of sorting is an input/output pair characterization of an algorithm, and in this case we are confronted with two different algorithms. Nevertheless, in the case of validating code generation our notion of equivalence is adequate.

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

Fig. 1 gives an illustration of the actual validation process. This process consists of three steps. First, the original high level language program is converted via a suitable set of transformations to the intermediate representation. Second, we build the intermediate representation for the assembly language program by using a technique termed "symbolic interpretation." This technique consists of activating a set of procedures corresponding to instructions in the low level program consistent with an execution level definition of the high level language (similar to interpretation). This intermediate representation reflects all of the computations performed on all possible execution paths. Third, a check must be performed of the equivalence of the two intermediate representations. This check takes 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.

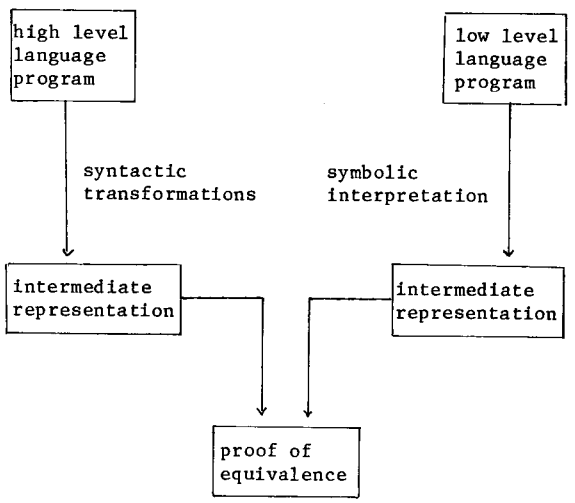

Fig. 1. Compiler testing system diagram.

Symbolic interpretation makes use of an execution level definition for the high level language. Such a definition is concerned primarily with calling sequence conventions, representation of primitive constructs of the high level language in the low level language, and an assumed run-time environment. Further details can be found in ([5],[7]).

The use of procedures to describe the low level language instructions, symbolic interpretation, and an intermediate representation are the distinguishing factors between our approach and one that would use decompilation methods [8]. Such an approach attempts to reconstruct the original program by searching for syntax in the object code. We are able to handle compilers written by different students since, unlike the case when decompilation methods are used, there is no need to know what code sequences each person (or compiler) uses to encode a particular construct of the high level language.

3. EXAMPLE

In order to demonstrate the usefulness of these ideas a proof system has been constructed which employs a subset of LISP 1.6 [9] (a variant of LISP [10]) as the high level language and LAP [9] (a variant of the PDP-10 [11] assembly language) as the low level language. A suitable intermediate representation for LISP is shown to exist in [12]. In this section we present a typical user session.

As an example, consider the function NEXT whose MLISP [13] (a parentheses free LISP also known as meta-LISP) definition is given in fig. 2. 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.

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

Fig. 2. MLISP encoding of NEXT.

Prior to discussing any low level language encodings, we must have an execution level definition for our high level language. We assume that a LISP cell is 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 occurs, the contents of all the accumulators are assumed to have been destroyed unless otherwise known.

Fig. 3 contains an erroneous LAP encoding, obtained by a hand coding process, for the function given in fig. 2. The format of a LAP instruction is (OPCODE AC ADDR INDEX) where INDEX and ADDR are optional. OPCODE is a PDP-10 instruction optionally suffixed by @ which denotes indirect addressing. The AC and INDEX fields contain numbers between 0 and decimal 15. ADDR denotes the address field. Verbal descriptions of the instructions used in the example LAP encodings can be found in Appendix I.

```
NEXT    (JUMPE 1 TAG1)      jump to TAG1 if L is NIL
PC2     (HRRZ 1 0 1)        load accumulator 1 with CDR(L)
PC3     (SKIPN 3 1)         load accumulator 3 with CDR(L) and skip if not NIL
        (POPJ 12)           return NIL
PC5     (HLRZ 4 0 3)        load accumulator 4 with CAR(L)
        (CAME 4 2)          skip if CAR(L) is EQ to X
        (JRST 0 PC2)        compute NEXT(CDR(L),X)
        (HLRZ 1 0 1)        load accumulator 1 with CAR(CDR(L))
TAG1    (POPJ 12)           return
```

Fig. 3. Erroneous LAP encoding of NEXT.

As soon as the user starts the proof system, he is asked some questions with respect to the execution level definition that he has assumed in his LAP encoding. These questions serve in part to inform the system of certain assumptions that might have been made with respect to optimization of function calls. Specifically, when functions invoke other functions than themselves, then they must generally go through the CALL or JCALL mechanism [9] (in case tracing is possible). Antisymmetry refers to the relationship exemplified by A<B is equivalent to B>A. The question dealing with destruction of accumulators pertains to overriding the initial assumption that upon recursion or an external function call all accumulators are assumed to no longer have the same contents as they had prior to the call. A typical user system dialogue for the programs shown in fig. 2 and fig. 3 is shown below.

```
ENTER FILENAME CONTAINING LAP PROGRAM TO BE
VALIDATED.
**(NEXT.LAP)

IF THE STACK POINTER IS NOT IN ACCUMULATOR 12
THEN ENTER AN ACCUMULATOR NUMBER.  OTHERWISE
ENTER NIL.
**NIL

ENTER NAME OF FUNCTION TO BE VALIDATED.
**NEXT

ENTER FILENAME CONTAINING LISP ENCODING OF THE
FUNCTION TO BE VALIDATED AND THE FUNCTIONS
INVOKED BY IT.
**NEXT

THE ONLY KNOWN ANTISYMMETRIC FUNCTIONS ARE THE
PAIRS CONS XCONS AND *LESS *GREAT.  ENTER A LIST
OF ANY OTHER PAIRS OR NIL.
**NIL

THE ONLY KNOWN COMMUTATIVE FUNCTIONS ARE EQ,
EQUAL, *PLUS, AND *TIMES.  ENTER A LIST OF OTHERS
OR NIL.
**NIL

THE ONLY FUNCTIONS KNOWN NOT TO DESTROY ALL
ACCUMULATORS ARE CONS, XCONS, NCONS, AND ATOM.
ENTER ANY OTHERS AND THE NUMBER OF THE HIGHEST
ACCUMULATOR THAT THEY DESTROY IN A DOTTED PAIR
FORMAT.  OTHERWISE ENTER NIL.
**NIL

ENTER A LIST OF FUNCTIONS WHICH CAN BE ACTIVATED
WITHOUT THE CALL MECHANISM.  BE SURE TO ENTER
THE NAME OF THE FUNCTION BEING VALIDATED, IF
APPLICABLE.  OTHERWISE ENTER NIL.
**NIL

ENTER A LIST OF FUNCTIONS WHICH MUST BE
ACTIVATED BY THE CALL MECHANISM.  DON'T INCLUDE
CONS, XCONS, NCONS, ATOM, EQ, NOT, NULL, CAR, AND
CDR WHICH ARE ALREADY KNOWN.  OTHERWISE ENTER NIL.
**NIL
```

As stated earlier, the encoding given in fig. 3 is erroneous. The validation process returns with an error message of the form shown below.

```
Computation (CAR (CDR L)) can not be matched.
Computed at instruction PC5
along path NEXT,PC2,PC3,PC5.
```

The message identifies a computation which can not be shown to be computed in both programs (i.e., the LISP and LAP programs), and the location and execution path along which the erroneous function is computed. This can be seen by examining fig. 4 and fig. 5 which show the intermediate representations corresponding to fig. 2 and fig. 3 respectively. The latter has been obtained by symbolically interpreting the program in fig. 3. These intermediate representations are in the form of trees with a predicate at the root and the left and right subtrees correspond to the true and false values respectively of the predicate. In fact, these forms are given as part of the error message that the validation procedure emits when encountering an error. Hence, users of the system should be capable of deducing the error with this information at hand.

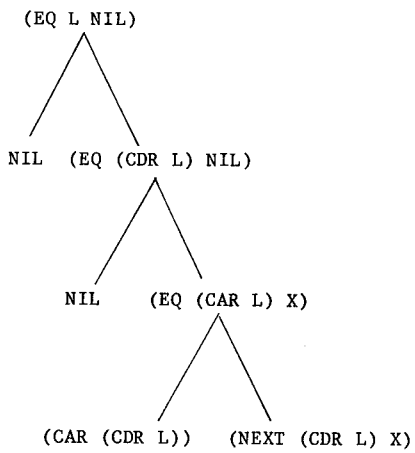

Fig. 4. Intermediate representation of fig. 2.

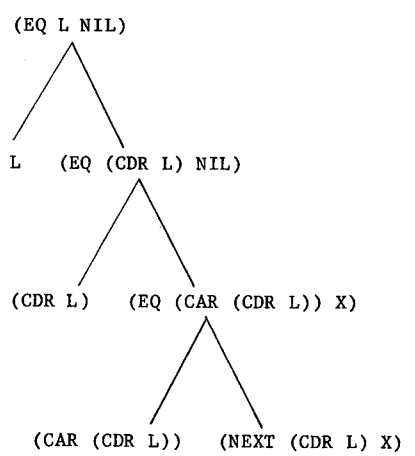

Fig. 5. Intermediate representation of fig. 3.

The problem is in the computation of the function (CAR (CDR L)) which is later compared with X. Closer examination of fig. 4 and fig. 5 reveals that the predicate should have been (EQ (CAR L) X) and in fact (CAR (CDR L)) should only be computed in case the latter predicate is found to be true. The problem is that we have destroyed the location containing L before encountering the last instruction at which L is needed. However, by inserting a temporary storage operation between locations NEXT and PC2 in fig. 3 we will have the desired result. At this point, the conditional skip and load operation at location PC3 is no longer necessary - i.e., a test is sufficient and we replace the skip by a conditional jump. The resulting correct encoding is shown in fig. 6. Note that the comment field in fig. 3 corresponding to location PC5 contains what the programmer thought was being computed - i.e., the comment is also in error!

```
NEXT    (JUMPE 1 TAG1)      jump to TAG1 if L is NIL
PC2     (MOVE 3 1)          load accumulator 3 with L
        (HRRZ 1 0 1)        load accumulator 1 with CDR(L)
        (JUMPE 1 TAG1)      jump to TAG1 if CDR(L) is NIL
        (HLRZ 4 0 3)        load accumulator 4 with CAR(L)
        (CAME 4 2)          skip if CAR(L) is EQ to X
        (JRST 0 PC2)        compute NEXT(CDR(L),X)
        (HLRZ 1 0 1)        load accumulator 1 with CAR(CDR(L))
TAG1    (POPJ 12)           return
```

Fig. 6. Modified LAP encoding of NEXT.

4. ERRORS

As noted in the previous section, errors in the translated program that are caused by the translation process can be detected. These errors can generally be classified into two categories depending on the instance of their detection. The first class of errors pertains to the well-formedness of the program. The second class of errors consists of computations occurring in the source program and not in the object program and vice versa.

Errors pertaining to the well-formedness of the program are detected during the symbolic interpretation of the translated program. Typical errors include improper calling sequences, illegal stack pointer formats, illegal operations on certain high level language data structures (e.g., arithmetic on LISP pointers), etc. Such errors are not uncommon and generally are the result of a state of confusion on the part of the compiler writer. These errors are of the type that would be detected when attempting to execute the translated program. However, the error message that will be obtained is nowhere as revealing as the message provided by our system since we can pinpoint the instruction which will cause the problem to occur at some time in the future. Thus the errors might be better characterized as warnings.

The second class of errors is much harder to detect without a proof system. In such a case there is nothing seemingly wrong with the translated program. Execution of the translated program may not yield any problems that might cause the program to blow up. These are the kind of errors that occur when senses of tests are confused, a function is applied to the wrong set of arguments, or the wrong function is being applied. The last two errors often result from improper use of common subexpression elimination, rearranging the order of computing arguments to functions, etc. In large programming systems, such errors may go undetected for a considerably long period of time.

The errors that have been detected during experimental use of the system include the following: Use of CAR instead of CDR. This was due to the use of a HLRZ instruction instead of a HRRZ - i.e., possibly due to misspelling. Computation of CONS(A,B) instead of CONS(B,A). Misconception about the sense of a test - e.g., computing A>B as the opposite test of B>A rather than B greater than or equal to A. (CDR (CDR L)) instead of (CDR (CAR L)). Errors were also detected with respect to illegal stack pointer formats.

5. CONCLUSION

We have seen how a validation system can be used to make code generation more tractable. We now have a means of evaluating code generation from both an efficiency and a correctness viewpoint. Hence this aspect of the compilation process need not be shunted in favor of parsing on the grounds that it cannot be evaluated in a systematic manner.

A system for proving the correctness of code generation for LISP and the PDP-10 currently exists. It can be used with several dialects of LISP - LISP 1.6 and UCI LISP [14]. A special version of the system has been constructed for use in conjunction with a course on LISP and data structures. In this course one of the options for a term project is the construction of an optimizing LISP compiler (in fact such a course provided the seed for the system reported here). It is hoped that use of such a system will reveal the type of errors that are commonly made by students. Results of an analysis of these errors can be used by the instructor to guide his exposition of the code generation process.

Future work includes an extension to handle object code for other machines besides the PDP-10 (e.g., PDP-11 [15]). An equally interesting extension, although more difficult, would be the creation of a proof procedure for a high level language other than LISP. Granted, LISP is a rather simple language in terms of its constructs. In particular, its control structure (i.e., case analysis) is very similar to our intermediate representation. Nevertheless, we feel that other well structured high level languages that are richer in data structures could also be handled. For example, in order to handle a language such as PASCAL [16], we would need to be able to cope with array descriptors, record structure descriptors, etc. There is also a more general procedure calling and return mechanism due to the various types of parameters that can be used.

Another possible direction for future work is expanding the type of information that can be supplied to the system by the user. We have seen a limited amount of such information in the sample dialogue. Performance of our system could be improved by declaring additional identities or equivalences. For example, associativity of functions, equalities, constancy of global variable bindings across function calls, etc.

ACKNOWLEDGMENT

Special thanks go to Robert E. Noonan for his helpful comments on the presentation of this paper.

APPENDIX I

The following are verbal definitions of LAP instructions encountered in fig. 3 and fig. 6. AC denotes the accumulator specified by the accumulator field of a LAP instruction.

CAME: The contents of AC is compared with the contents of the effective address and the next instruction is skipped if equality holds.

HLRZ: Load the right half of AC with the left half of the contents of the effective address; clear the left half of AC.

HRRZ: Load the right half of AC with the right half of the contents of the effective address; clear the left half of AC.

JRST: Unconditional jump to the effective address.

JUMPE: Jump to the effective address if the contents of AC is zero; otherwise continue execution at the next instruction.

MOVE: Load AC with the contents of the effective address.

POPJ: Return from a recursive function call. Formally, subtract octal 1 000 001 from AC to decrement both halves by one and place the result back in AC. If subtraction causes the count in the left half of AC to reach -1, then set the Pushdown Overflow flag. The next instruction is taken from the location addressed by the right half of the location that was addressed by the right half of AC prior to decrementing.

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

REFERENCES

[1] D. Gries, Compiler construction for digital computers, John Wiley and Sons, Inc., New York, 1971.

[2] H.D. Shapiro and M.D. Mickunas, A new approach to teaching a first course in compiler construction, Proceedings of ACM SIGCSE-SIGCUE Joint Symposium, February 1976, 158-166.

[3] Curriculum 68, Communications of the ACM, November 1968, 151-197.

[4] F.R.A. Hopgood, Compiling techniques, American Elsevier, New York, 1969.

[5] H. Samet, 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.

[6] J.A.N. Lee, Computer semantics, Van Nostrand Reinhold, New York, 1972, 346-347.

[7] H. Samet, Compiler testing via symbolic interpretation, Proceedings of the ACM 29th Annual Conference, 1976, 492-497.

[8] C.R. Hollander, Decompilation of object programs, Ph.D. Thesis, Digital Systems Laboratory Technical Report No. 54, Department of Electrical Engineering, Stanford University, 1973.

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

[10] J. McCarthy, Recursive functions of symbolic expressions and their computation by machine, Communications of the ACM, April 1960, 184-195.

[11] PDP-10 system reference manual, Digital Equipment Corporation, Maynard, Massachusetts, 1969.

[12] H. Samet, A normal form for LISP programs, TR-443, Computer Science Department, University of Maryland, College Park, Maryland, 1976.

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

[14] R.J. Bobrow, R.R. Burton, and D. Lewis, UCI LISP manual, University of California at Irvine Technical Report No. 21, University of California at Irvine, October 1972.

[15] PDP-11 reference manual, Digital Equipment Corporation, Maynard, Massachusetts, 1973.

[16] N. Wirth, The programming language PASCAL, Acta Informatica, Vol. 1, No. 1, 1971, 35-63.