Towards Code Optimization in LISP

by

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

Abstract

Code optimization is examined from the viewpoint of the needs of a list processing language such as LISP. Examination of the structure of programs written in such a language reveals that traditional code optimization techniques have little benefit. Instead, a collection of low level optimizations is seen to lead to a potential decrease in space and execution time. Code optimization is also examined from the viewpoint of its effect on the frequency of occurrence of garbage collection. Finally, some language extensions are proposed which reduce the amount of storage that needs to be allocated and hence will result in a decrease in the frequency of garbage collection.

Résumé

On examine l'optimisation du code pour un langage de traitement de listes comme LISP. L'examen de la structure des programmes écrits dans un tel langage montre que les techniques classiques sont de peu de secours. Au contraire, plusieurs optimisations de bas-niveau conduisent à un gain en temps et en espace. L'optimisation est aussi étudiée pour ses conséquences sur la fréquence d'appel du ramasse-miettes. Finalement, on propose des extensions linguistiques qui réduisent l'espace mémoire devant être alloué et donc réduisent la fréquence d'appel du ramasse-miettes.

1. INTRODUCTION

With the growing interest in the field of artificial intelligence has come an increase in computing involving symbolic expressions. This interest has been coupled with the development of a variety of programming languages [BobrowRaphael74]. Many of these languages are interpreter-driven and are characterized by a small set of basic primitives. The use of an interpreter has been tolerated by most users by virtue of the smallness of their application. However, the increasing use of these languages in knowledge-based systems (e.g. MYCIN [Shortliffe74]) has led to the resurfacing of the efficiency problem. The most obvious solution to this problem is to use a compiler. However, the nature of the programs written in such languages has not traditionally lent itself to very efficient code.

In this paper we examine some of the consideration that must be taken into account when attempting to obtain a code optimizer for a list processing language. Our presentation consists of three parts. First, we discuss some of the low level optimization techniques which are seen to have a direct effect on the amount of space occupied by the program and on the execution time of the program. Next, we explore the types of optimizations whose effect is not directly discernible. These optimizations deal with reducing the frequency of garbage collection. Finally, we discuss the feasibility and problems associated with some of the proposed optimizations.

In order to illustrate our ideas we use LISP1.6 [Quam72] (a variant of LISP [McCarthy60]) as the high level list processing language and LAP [Quam72] (a variant of the PDP-10 [DEC69] assembly language) as the object language. 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. The implementation at hand 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 to find its parameters in accumulators 1-n. Accumulator 1 is used to return the result of the function. Accumulator 12 contains a stack pointer.

2. LOW LEVEL OPTIMIZATION TECHNIQUES

Traditional code optimization work [CockeSchwartz70] is primarily oriented towards making use of flow analysis to yield common subexpression elimination and reduction in strength of operators. In most list processing systems such considerations are not as important. Examination of the structure of typical programs reveals that they consist of a large number of small, often recursive, modules. Analysis of the actual programs shows that most of the execution time is spent in setting up the linkages between functions as well as for recursion. Thus it seems that a substantial payoff can result from optimizing the linkages and making the recursive step as fast as possible. The latter means that the execution path which corresponds to an occurrence of a recursive call is optimized at the possible expense of other execution paths.

Significant reductions in execution time can be obtained by making use of an adaptive calling sequence. Recall that we mentioned that we are dealing with a LISP system where all the arguments to a function are found in the accumulators. Clearly, this convention must be adhered to when there is communication between two different functions. However, in the case of recursion, this is unnecessary. Observation of a large number of programs reveals that whle preparing for a function call, a stack is used often for saving values of arguments that have already been computed. Once all of the arguments have been computed, the accumulators are loaded with the appropriate values and the function call occurs. However, in most instances, the first task performed by the function is to save its arguments on the stack. The amount of extraneous data shuffling should be obvious.

A calling sequence which uses the stack exclusively has its own share of problems. There is a need for more memory to hold the stack and extra memory operations are necessary when items are accessed from the stack. Data shuffling remains a problem when functions call other functions with many of the same variables in the same argument positions. The difficulty is that room must be made for the return address. A solution is to have a separate parameter stack and control stack. However, this has the disadvantage that much space must be used as well as there is a constant need to have the parameters on the stack, whereas in a calling sequence which makes use of accumulators, only the parameters needed for future reference are saved on the stack. The reason a calling sequence which makes use of accumulators appears so poor is that only rarely is there compliance with the previous criterion. Most compilers fail to make the distinction between what should and should not be saved on the stack.

```
START (PUSH 12 1)                     START    (PUSH 12 4)
      (PUSH 12 2)                              (PUSH 12 3)
      (PUSH 12 3)                              (PUSH 12 2)
      (PUSH 12 4)                     NEWSTRT  (PUSH 12 1)
      = save arguments on stack                 = save arguments on stack
         .                                         .
         .                                         .
         .                                         .
      Compute argument for acc.1               (PUSH 12 (C 0 0 RESUME))
      (PUSH 12 1)                               = Save the return address
      Compute argument for acc.2               Compute argument for acc.4
      (PUSH 12 1)                              (PUSH 12 1)
      Compute argument for acc.3               Compute argument for acc.3
      (PUSH 12 1)                              (PUSH 12 1)
      Compute argument for acc.4               Compute argument for acc.2
      (MOVE 4 1)                               (PUSH 12 1)
      (MOVE 1 -2 12)                           Compute argument for acc.1
      (MOVE 2 -1 12)                           (JRST 0 NEWSTRT)
      (MOVE 3 0 12)                             = jump to NEWSTRT
      (SUB 12 (C 0 0 3 3))            RESUME   Continue with rest of
       = set up calling sequence                  computation
         by restoring arguments                    .
         from the stack                            .
      (PUSHJ 12 START)                             .
       = perform the recursion        END      (SUB 12 (C 0 0 4 4))
      Continue with rest of                    (POPJ 12)
         computation
         .
         .
         .
END   (SUB 12 (C 0 0 4 4))
      (POPJ 12)
```

Fig. 1 - Comparison of Accumulator and Mixed Calling Sequences

We propose that for an internal recursive call, a mixed calling sequence might be appropriate. In this case some of the parameters are found on the stack and others are found in the accumulators. In such a case if there is more computing to be done within the function after the recursive call, then it is necessary to place the return address on the stack prior to the placement of the arguments on the stack. For example, consider the function START of four arguments in fig. 1. On the left is given the normal LAP encoding while on the right is given an encoding which makes use of a mixed calling sequence. Note that the order of computing arguments was rearranged. Such rearranging must be capable of being proved to yield equivalent results with the original encoding. Also observe a shift in the location of the parameters to the function at function entry. In the case of entry from outside of the function, accumulators 1-4 will contain the parameters; whereas, if the entry was from within the function, then only accumulator 1 contains a parameter. This, again, requires a proof that accumulators 2-4 are never referenced past the label NEWSTRT with the assumption that they contain the parameters to the function.

Another variation of calling sequence rearrangement can be seen in fig. 2 (note the use of MLISP [Smith70] - an ALGOL-like [Naur60] version of LISP) where a function, REVERSE, to reverse the links of a list is encoded with the aid of an auxiliary function. However, instead of the customary formulation of the auxiliary function, we have interchanged the first and second arguments. Thus the accumulating variable is the first argument rather than the second. The LAP encoding, obtained by a hand coding process, demonstrates an internal calling sequence where L is in accumulator 3 and RL is in accumulator 1. This is useful because XCONS, like CONS with the arguments reversed - i.e. XCONS(B,A)=CONS(A,B), is known to leave all accumulators but 1 and 2 unchanged. Thus there is no need to save L or CDR(L) while computing CONS(CAR(L),RL). Note that to all external functions, REVERSE1A still appears to require two arguments in accumulators 1 and 2.

```
REVERSE(L) = REVERSE1A(NIL,L)
REVERSE1A(RL,L) = if NULL(L) then RL
                  else REVERSE1A(CONS(CAR(L),RL),CDR(L))
```

Fig. 2 - MLISP Definition of REVERSE

```
(LAP REVERSE1A SUBR)
        (SKIPN 3 2)              load accumulator 3 with L and
                                 skip if not NIL
        (POPJ 12)                return NIL
REV     (HLRZ 2 0 3)             load accumulator 2 with CAR(L)
        (CALL 2 (E XCONS))       compute CONS(CAR(L),RL)
        (HRRZ 3 0 3)             load accumulator 3 with CDR(L)
        (JUMPN 3 REV)            if CDR(L) is not NIL then compute
                                 REVERSE1A(CONS(CAR(L),RL),CDR(L))
TAG1    (POPJ 12)                return
```

Fig. 3 - LAP Encoding Corresponding to Fig. 2

The LAP encoding in fig. 3 serves to illustrate the notions expressed earlier with respect to optimizing the execution path corresponding to recursion.

(1) Use is made of known values of predicates in order to enable bypassing the start of the program when recursion occurs.

(2) Instructions are used which accomplish two tasks at once. (SKIPN 3 2) results in a test of the nullness of L as well as a load of accumulator 3. (JUMPN 3 REV) results in the test of the nullness of L as well as recursion. Note that in this case the execution path corresponding to recursion is optimized in the sense that (JUMPN 3 REV) is used instead of the sequence (JUMPE 3 TAG1) followed by an unconditional branch to REV.

(3) Flow analysis is seen to play an important role in the placement of parameters. In the program at hand, knowledge that XCONS leaves accumulators 1 and 2 unchanged enables a shift of a parameter to accumulator 3 thereby avoiding the data shuffling which would have been inevitable had accumulator 2 been used.

In fig. 1 the return address is pushed on the stack prior to the computation of the parameters to the function call. The same technique can be used in the following situation. Suppose that a function tests a number of conditions, and that based on these conditions, other functions are executed (similar to a CASE statement in ALGOL). Upon termination, all of these functions must return to a common point and execute a segment of code. For example, see fig. 4 where a function epilogue is illustrated. This can be implemented by executing a branch to the desired location of the common code sequence after each of the function calls. However, a more efficient approach is to push the address of the common code sequence on the stack prior to testing the first condition, and then to invoke the functions in the various cases via simple branch (non stack) instructions. When the invoked functions terminate, they will return via the stack. Thus the size of the program has been reduced by a number of instructions equal to one less than the number of conditions, with virtually no increase in execution time. Actually, the difference in execution times is the difference between executing a PUSH plus an unconditional jump and a recursive jump (PUSHJ) plus an unconditional jump.

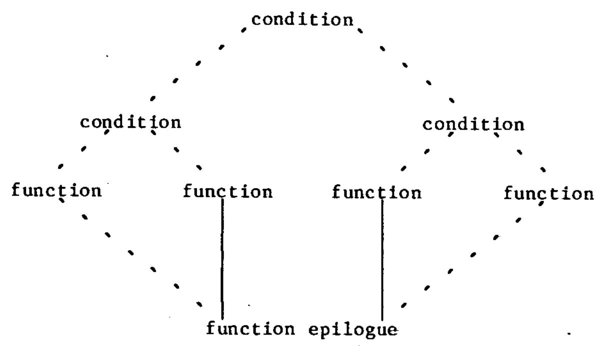

Fig. 4 - Function Epilogue

3. GARBAGE COLLECTION-RELATED OPTIMIZATIONS

Unlike most conventional programming languages, LISP does not have an explicit storage allocation command. Storage is allocated whenever a CONS operation is performed, in which case a cell is allocated from a heap (i.e. the free storage list). More importantly, there is no mechanism for releasing storage. Thus once the free storage list is exhausted, we have two alternatives. One choice is to quit and emit a message to the effect that storage is exhausted. The second choice entails determining which of the cells that have been previously allocated are no longer accessible. This procedure is known as garbage collection. Such a process is rather time consuming. This is especially true if most of the cells are in use. In such a case, very little storage is reclaimed and the garbage collection procedure will have to be reinvoked shortly. Such a situation tends to negate any effect of optimizations with respect to execution time (savings in the space occupied by the program are still valid).

The need to do garbage collection is one of the primary deficiencies of LISP and other similar list processing languages. This is because when garbage collection takes place, execution of the program is suspended until a sufficient amount of storage is reclaimed. The suspension of execution is extremely undesirable in real-time applications. The only alternative is to possibly have a second processor whose sole task is to perform garbage collection. Thus it is seen that in the ideal situation, garbage collection is always taking place in parallel with the main process (for an interesting exposition on the problems associated with parallel garbage collection see [Steele75]).

In section 2 we discussed a number of low level optimizations. Individually, they are not earth-shattering. However, when viewed collectively, significant reductions in total space and time can be observed. Some of these optimizations have additional benefits. Specifically, those optimizations which result in the reduction of the size of the stack should reduce the number of active links that will need to be pursued during the marking phase of garbage collection. Furthermore, such a decrease will most probably be accompanied by an increase in the number of cells that are reclaimed. We have no data with respect to the effect of our optimizations on garbage collection. However, one rather complex program which was optimized using the techniques presented here occupied 72% of the space needed by the original function, was 40% faster in execution time, and required 50% of the stack space needed by the original function. The last reduction has an immediate benefit in the sense that the amount of stack space necessary can be reduced thereby decreasing the likelihood of stack overflow.

One approach to reducing the need to do garbage collection at run-time is termed "compile-time garbage collection" [DarlingtonBurstall73]. In this method, algorithms which use CONS cells are converted by the compiler to "equivalent" algorithms which do not perform CONS operations. This is achieved by changing links in the list structure rather than working on a new copy of the data structure. For example, the algorithm given in fig. 2 to reverse the links of a list can be encoded using compile time garbage collection in the following manner:

```
REVERSE(L) = REVERSE2(L,NIL);
REVERSE2(L,RL) = if NULL L then RL
                 else REVERSE2(CDR L,RPLACD(L,RL));
```

However, such techniques do not preserve equivalence since if node-sharing is likely to occur, then all lists in which the argument to REVERSE appears, will be effected -- an undesirable result.

Until now we have seen how the frequency of garbage collection can be effected by code optimization. An alternative method of reducing this frequency is to change the mechanism which is used to obtain storage. A CONS operation is often performed unnecessarily - i.e. a cell containing the appropriate elements of the pair has already been allocated. For example, suppose CONS(A,B) has been performed. Subsequently, it is determined that A is EQ to C. Now, if it is desired to perform CONS(C,B), then there is in fact no need to do so since such a cell already exists. This can be recognized by using a hashing [Knuth73] scheme for CONS where the hashing function has as its arguments the addresses corresponding to the arguments of the CONS. Of course, such an implementation means that prior to the allocation of a cell from the free storage list, we must determine if a cell has already been allocated with the same components. Also note that when CONS is hashed, the determination of whether a CONS cell with the same components has already been used is done at run-time and is unrelated to common subexpression elimination - a process performed at compile-time.

There are several factors which must be taken into account when evaluating the hashed CONS method of storage allocation.

(1) If side-effect operations such as RPLACA and RPLACD are allowed, then there is a potential for disastrous results due to node-sharing.

(2) There is a constant need to check if a cell has already been allocated with the said components. This may prove to be time consuming. However, as mentioned earlier, in a real-time application we are more concerned with predictability of length (i.e. in time) of operations. Nevertheless, the fact that the amount of time spent to check if a cell has already been allocated greatly exceeds the average time per cell spent in a garbage collection process must be taken into consideration.

(3) There is a need for hash tables and pointers to keep track of the various cells in use. Therefore, unless there is a high factor of duplication (i.e. there are many instances of CONS cells with the same components) we may find that our available storage has been cut down significantly. However, these tables need not be part of the free storage list. This has a significant meaning in an environment with a limited address space. For example, on certain versions of a minicomputer such as the PDP-11 [DEC73] which support segments, the tables may be kept in a separate segment. In such a case there is a premium on non-redundant use of the free storage area and the use of a hashed CONS aids in maximizing the available space.

Often a CONS operation is performed in order to circumvent the inability to return more than one result from a LISP function. In such a case a list is formed whose elements are the results of the function rather than by use of a special construct as is available in the POP2 [Burstall71] language. In a LISP system which uses an accumulator to return the value of the function and a stack for the purpose of program control, a return of more than one result can be implemented as follows. Return the first result in an accumulator, and return the remaining results in a contiguous block of storage immediately above the top of the stack (which contains the return address). The only remaining task is to indicate how a multiple result is to be specified in LISP. We feel that the most natural way is to add the special form RLAMBDA which we define to be identical to an internal LAMBDA of as many arguments as there are results being returned and only one binding - i.e. the function which has more than one result.

For example, see fig. 5 where the function G calls the function H which returns as its results three values. In this case, the variables B, C, and D in function G are bound to (H1 A), (H2 A), and (H3 A) respectively.

```
(DEFPROP G (LAMBDA (A)
                   ((RLAMBDA (B C D)
                             (M B C D))
                    (H A)))
 EXPR 1)
(DEFPROP H (LAMBDA (A)
                   (H1 A)
                   (H2 A)
                   (H3 A))
 EXPR 3)
(DEFPROP SUMDIV (LAMBDA (DIVIDEND DIVISOR)
                        ((RLAMBDA (QUOTIENT REMAINDER)
                                  (*PLUS QUOTIENT REMAINDER))
                         (DIVISION DIVIDEND DIVISOR)))
 EXPR 1)
```

Fig. 5 - Examples of Functions that Return More than One Result

The only distinction with LAMBDA is that the values of all but the first argument are found on top of the stack. Thus if any other functions are to be called, then the stack must be adjusted to save these values below the stack pointer which points to the top of the stack. A typical solution is to store the value that was returned in the result accumulator (i.e. accumulator 1) in the location pointed at by the stack pointer (i.e. the stack location which contained the previous return address), and then increase the stack pointer by a number equal to the number of results that were returned.

We must also provide a syntactic mechanism for denoting that more than one result is to be returned. The easiest way to achieve this is to have, in addition to the property associated with each function denoting its type (e.g. EXPR when the arguments have been evaluated prior to the function call and FEXPR if not), a property that indicates the number of results returned by the function. For example, the function H in fig. 5 is an EXPR which returns 3 results. The actual act of returning more than one result, say n, is accomplished by returning the last n values that have been computed which are not subexpressions of other computations. We also make the stipulation that a function returns the same number of results in all cases.

As an additional example, consider the function DIVISION which returns as its result the QUOTIENT and the REMAINDER when integer division is performed on its two arguments - i.e. the first argument is integer-divided by the second argument. Fig. 5 shows the use of DIVISION in the definition of the function SUMDIV of two arguments which returns the sum of the quotient and the remainder when DIVIDEND is integer-divided by DIVISOR.

The above examples lead us to conclude that LISP should provide a capability for control over the deallocation of cells. The example of how multiple results are currently handled demonstrates a need for deallocation to be performed by the function to which multiple results are returned. There is also a potential need for deallocation at function exit as is done in ALGOL 60. This could be accomplished by use of specialized CONS operations which leave messages as to the lifetime of the cell that has been allocated. We feel that the determination of cells that can be deallocated in such a manner would be best achieved by a comprehensive flow analysis package.

4. CONCLUSION

We have presented two views of optimization for a LISP system. The main thrust of the presentation has been towards obtaining an efficient compiler-based system. However, several of the optimizations proposed in section 3 could also be used in an interpreter-based system.

A majority of the optimizations proposed in section 2 have a heuristic flavor associated with them. Many are a result of a trial and error code generation procedure. In such a case, there may be several attempts at obtaining an optimal encoding; some of which might be incorrect. The correctness of the translation can be demonstrated by use of a proof system such as [Samet75] which has as its input a high level language encoding of an algorithm and a low level language encoding of the same algorithm. Such a proof system is embedded in the translator and is intended to be the final step in the code generation procedure.

The optimizations of section 3 are more of a language design nature. They must be evaluated in light of their effects on programming style. Clearly, a hashed CONS mechanism implies node-sharing and therefore its use deprives the programmer of the ability to use RPLACA and RPLACD. However, the introduction of a multiple result feature does not seem to have any drawbacks. Another factor to consider is the size of the available directly addressible memory. This is a factor that might lead to the adoption of a hashed CONS mechanism.

5. REFERENCES

[BobrowRaphael74] - Bobrow, D.G., and Raphael, B., "New Programming Languages for Artificial Intelligence," ACM Computing Surveys, September 1974, pp. 153-174.
[Burstall71] - Burstall, R.M., Collins, J.S., and Popplestone, R.J., Programming in POP2, University Press, Edinburgh, Scotland, 1971.
[CockeSchwartz70] - Cocke, J., and Schwartz, J.T., Programming Languages and their Compilers, New York University Courant Institute, April 1970.
[DarlingtonBurstall73] - Darlington, J., and Burstall, R.M., "A System which Automatically Improves Programs," in Proceedings of the Third International Joint Conference on Artificial Intelligence, 1973, pp. 479-485.
[DEC69] - "PDP-10 System Reference Manual," Digital Equipment Corporation, Maynard, Massachusetts, 1969.
[DEC73] - "PDP-11 Reference Manual," Digital Equipment Corporation, Maynard, Massachusetts, 1973.
[Knuth73] - Knuth, D.E., Sorting and Searching, Addison-Wesley, Reading, Massachusetts, 1973.
[McCarthy60] - McCarthy, J., "Recursive Functions of Symbolic Expressions and