Output Driven Interpretation of Recursive Programs, or Writing Creates and Destroys Data Structures*

Daniel P. Friedman

David S. Wise

Computer Science Department

Indiana University

Bloomington, Indiana 47401

TECHNICAL REPORT NO. 50

OUTPUT DRIVEN INTERPRETATION OF RECURSIVE PROGRAMS, OR
WRITING CREATES AND DESTROYS DATA STRUCTURES

DANIEL P. FRIEDMAN
DAVID S. WISE

JULY, 1976

*Research reported herein was supported (in part) by the National Science Foundation under grants numbered DCR75-06678 and MCS75-08145.

Output Driven Interpretation of Recursive Programs, or Writing Creates and Destroys Data Structures*

Daniel P. Friedman

David S. Wise

Computer Science Department

Indiana University

Bloomington, Indiana 47401

Abstract - We relate the methodologies of recursive and iterative programming by describing an interpretation scheme for recursive programs which in many cases requires run-time space resources comparable to those of the corresponding iterative code. The measure of space resource includes the representation of all environments which are created along the computation path. The side-effect-free language pure LISP is used as a model language for our interpretation scheme. Examples are given of non-terminating programs (the list of natural numbers and Pascal's triangle) which run and provide the desired output within time and space resources of corresponding iterative programs (O(1) and O(n) space, respectively).

Keywords and Phrases - suspended evaluation, reference counts, stack, LISP, cons.

CR Categories - 4.13, 4.22, 4.12.

*Research reported herein was supported (in part) by the National Science Foundation under grants numbered DCR75-06678 and MCS75-08145.

Introduction

Much recent work has been directed at reconciling programming methodologies of recursion and iteration [1,3,8,9,10,11]. Some work has also been directed at transforming recursive code to iterative code [3,5,18,22], and some has emphasized the ability of recursive code to clearly express iterative processes [12,20,21]. Theoretical results pointing the way to an equivalence of recursion and iteration are available as well [4,17,22]. It is our thesis that a stylized form of recursive programming offers an expressive and conceptually simple way to program efficiently. In this paper we argue the issue of efficiency by demonstrating how the semantics of a recursive language like _pure LISP_ can be changed to make the interpreter behave as if it were executing iterative code. This behavior does not result from any transformation on the user's program; rather it depends on a modification of the interpreter.

LISP

We use a variant of _pure LISP_ [16] as a communication language since it is the best known of the purely recursive programming languages. The notation used throughout the paper for form invocation is the S-expression. A list is written using the parenthesis notation; whether the interpreter accesses it as an expression rather than as a value determines whether evaluation will occur. We have made a notational change in the syntax of conditional expressions. McCarthy's _cond_ requires its tail to be structured as a series of lists of two elements which are often called "cond-pairs." Rather than introduce the redundant extra parentheses which make the pairings explicit, we use the commenting keywords *if*, *then*, *else*, and *elseif* to group the expressions and to enhance readability. The user who wishes to interpret an invocation of cond should ignore the commenting keywords.

The language pure LISP is side-effect-free in that bindings may only be effected by lambda and label. Although these facilities do not explicitly exist in our model, bindings are still restricted in such a way that any environment will remain intact for the duration of its existence. The programmer must live with the constraints imposed on his programming style (i.e. no assignment statements and no property lists). LISP is call-by-value so that the user need never worry about how or when actual parameters are evaluated. No cycles can occur in this side-effect-free system. Thus, the data structures, including environments, which can be built at run-time are directed ordered acyclic graphs (doags). As a result the standard LISP garbage collector has been replaced by a reference count scheme [5,23] in our implementation. The reference count in a record indicates the number of structures that share it. If a visit to a node is part of the last traversal of a particular structure, liberation of storage may be built into the traversal. Space requirements may become comparable to those of a good iterative program as a result.

Definitions

Fundamental to our use of reference counts is our reinterpretation of the classic semantics for cons [8]. Let us introduce the term manifest to describe a structure which is actually extant in its ultimate form, occupying as many records as it would under McCarthy's semantics for LISP. A data structure which is not completely manifest is said to be promised. At the implementation level, promised structures are characterized by the presence of some references which are suspended. A suspended reference consists of a pointer to a form (within the program definition) and one to an environment. That information is sufficient to coerce the reference to its ultimate value whenever appropriate, with the help of the system evaluator. A manifest structure has had all internal references coerced.

Promised structures are implemented by changing the definition of three of the five LISP primitives. Cons allocates a fresh record and fills it with suspensions of its arguments instead of with evaluated arguments. The probing functions car and cdr distinguish (using a "suspended" bit in every record) between suspensions and coerced values within the record which is their argument. (Figure 1 gives PASCALesque [13] type declarations and the code of car for our implementation.) If a suspension is uncovered, a value is required, and the evaluator is invoked upon the designated form within the referenced environment. The value which is returned as a result of the probing function is planted back in the record in place of the suspension. This scheme provides at most one reference to a suspension since suspensions are destroyed as they are coerced. The references to environments preserved within suspensions are likewise released; the structure representing an environment other than the global environment _will be destroyed_ with the last suspension referring to it.

The ability to promise structures provides LISP with new semantics. For instance, the user can specify an infinite vector [9] to be printed, which is presented to the output routine in promised form. The output traversal uses the functions _car_ and _cdr_ to traverse the vector, printing as it goes. As a result the prefix of the vector, that part which can be successfully coerced, can be displayed on the output device, whereas under the classic semantics for LISP nothing would be displayed but an error message indicating that the structure could not be completely manifested before printing.

Other authors have made suspensions available at the explicit request of the programmer. Landin [15] and Burge [1] provide a constructor which suspends one argument. POP-2 [2] allows a generating function to be converted into a suspended list. Henderson and Morris [10] describe an interpreter for LISP with behavior similar to ours based on elementary code transformations.

Burning your bridges behind you

The system we have described computes answers in suspended form, maintains brief and disjoint environments with suspensions, and depends only on reference counts for memory management of the space used to store both environment and user structure. In this section we shall demonstrate how the traversal algorithm of the output driver not only manifests the structure of the answer but also destroys it. The effect is a drastic reduction in the space required to interpret the recursive code.

Fischer [7] gives a powerful result for the programmer's data structure, pointing out that the only structure which need be manifested is that which is printed. However, his analysis does not allow for the cost of manifesting environments within the interpreter. Our space utilization analysis includes the cost of that structure as well. In terms of LISP our space analysis includes the recursion stack and all current environments (association lists).

It is clear how a traversal manifests a structure using the new interpretations for the functions car and cdr. To see that destruction is also possible requires another look at the global environment of the interpreter. This is composed of permanent bindings for the system functions and system constants. The interpreter allows the user to add his own functions and constants to those globally bound, but not to alter them. This is accomplished through a define facility. Before and after any function evaluation these are the only bindings extant in the entire system. Unless the user is dumping the value of one of these global bindings, the structure which he is printing will not survive to the next top-level evaluation except on the paper in his teletype.

Destruction of the data structure can occur as part of the print traversal if there is only one reference to the (perhaps promised) structure before printing. That single reference is held by the top-level of the system as the answer which exists solely to be printed; after the answer has been delivered to the output device (even though it is not yet manifested or printed) it is lost to the system.

Under a reference-count storage manager [5] the system admits a recursive traversal algorithm which adjusts the reference counts as it goes, and liberates all records and atoms whose reference count drops to zero as the printing traversal passes. In LISP terminology, we can destroy much of a (singly referenced) structure as soon as traversal passes in the

CDR direction, but passage in the CAR direction implies a parenthesized subexpression for which some memory is required just until the closing parenthesis is on the output file. The algorithm in Figure 2 traverses and coerces a data structure in double order (Problem 2.3.1-21 [14]) releasing an unshared node as soon as the traversal passes except for the first node on a list, which is preserved until the sublist is completely printed. No extra space is needed for a recursion stack because links from preserved nodes are inverted between the two visits of the traversal. This trick was used in [19] and is at the root of Knuth's problem; it requires an extra bit, in this case the "suspended" flag, to indicate which link is inverted.

The language used in Figure 2 is based on the language PASCAL [13] with enhancements to the control structure described in [24]. The *while-until* loop is a sequential control structure which evaluates its contents in order (iteratively) until the predicate following the *while* (or the *until*) evaluates to *FALSE* (respectively, *TRUE*); that Boolean value is the value of the loop itself. We use related notation from [24] as well. The function true (or false) takes statements as arguments; it evaluates all statements and returns the value *TRUE* (respectively, *FALSE*). The conditional expression is taken from ALGOL 60 and has the form *if...then...else...* ; a conditional statement has the form *if...then...else...fi* or *if...then...fi* . The Boolean operator & is conjunction as in LISP; if its first operand is *FALSE* then its second is not evaluated.

The procedure dispose belongs to PASCAL; it releases only the referenced node. The procedure print?dispose is introduced to print the pname of an atom and,in traversing the atomic structure while printing, to dispose of it if its reference count is zero.

Let us consider a simple example:

```
(numbersupto n) ≡ (cond
    if (zerop n) then []
    else (cons n (numbersupto (subl n))) ).
```

Under a classic LISP interpreter this code would require time O(n) and space O(n) for the linear recursion. Under the interpreter using suspending cons and the space-recovering output traversal algorithm proposed here, it would still require O(n) time, but only O(1) or constant space. These are the resources required by the "best" iterative code for similar output. Only constant space is required because at most one suspension at a time is coerced and each environment is used for only two coercions. A new environment is created at the second and final use of the old one so that as much space is released as is required for creation of the new one. The list of n numbers is only manifested one record at a time during the interpretation of (numbersupto n) at the top level, so only constant space is required (including all global bindings).

Observation 1: Pure LISP code can be interpreted so that some constructing functions with apparent recursion depth n (where n depends on input) require only constant space.

More simply, pure LISP code can be interpreted so that some functions with recursion depth apparently dependent on input require space proportional to that required by refined iterative code. This result can be extended to infinite structures, yielding (potentially) infinite output from constant memory allocation:

```
(successors n) ≡ (cons (add1 n) (successors (add1 n))).
```

The interpreter proposed would evaluate (successors 0) printing all the natural numbers until the program was interrupted. It would not stop before printing with the recursion stack exhausted, as does any classic LISP interpreter, nor would it stop with that stack exhausted after printing a prefix of the list, as does the interpreter described in our earlier paper [8]. In fact, the amount of memory allocated at any time during interpretation would remain uniformly small (bounded by a constant) just as in a tight iterative loop doing the same thing:

for n := 1 *step* 1 *while TRUE do* write(n) .

From the operating system's perspective the interpreted LISP code is indistinguishable from (rather slow) good iterative code; the resources consumed do not reflect the fact that the code being interpreted is recursive.

Let us now turn to a problem which requires O(n) space even under an efficient iterative program. The problem is an algorithm for printing Pascal's triangle of binomial coefficients. A good iterative program uses a vector of length n (perhaps n/2 if row symmetries are used) which is initialized to the vector of length 1 with contents 1. On each iteration the current row is printed and the next row is computed. Here is a version of the generative code for Pascal's triangle [9] which prints the entire (infinite) structure:

```
(pascal row) ≡ (cons row (pascal (cons (car row) (pairsum row))));
(pairsum row) ≡ (cond
    if (null (cdr row)) then row
    else (cons (sum (car row) (car (cdr row))) (pairsum (cdr row))).
```

The outermost call for printing the infinite structure is

```
        (pascal (cons 1 () ))
```

where the O(n) space claim is based on resources required for printing the first n rows of the structure:

```
        (prefixofsize n (pascal (cons 1 () ))).
```

Although this generation of Pascal's triangle was not designed for any particular run-time behavior it is interesting that it, too, requires only O(n) space and O(n^2) time to give the desired output. The occurences of <u>cons</u> act very much like basis conditions, stopping the evaluator until the traversal within the print routine coerces suspensions while manifesting the rows of the infinite array returned by the function pascal. Because the printing traversal follows the order in which the rows are generated, and because the generation of a new row depends only on the values in the previous row manifested by the print traversal before the new row is coerced, references into at most two rows are the most ever bound in any extant environment. Moreover, the recursion pattern requires at most two environments for any of these functions at once. The space bound follows; the time bound for both algorithms is trivially dependent on output size.

Observation 2: The Pascal triangle problem is solved recursively within the same resource bounds (up to a constant factor) as the "best" iterative solution.

How little was required of the recursive programmer in designing his algorithm is significant. In particular, an explicit data structure was not required before the design of the algorithm. Nor were flow of control or conflicting variable bindings of any concern. All these problems were handed to the interpreter of the recursive code which suddenly exhibits a run-time behavior very much like iterative code.

Conclusion

We have proposed a scheme in which the top-level control of a recursion interpreter is driven by the appetite of the output device. Restriction to a pure form of bindings guarantees the integrity of environments for a constructor function which suspends evaluation of its arguments. This ensures the independence of environments so that their lifetime is well controlled, and leads to a doag data structure, thus admitting reference counts for storage management. Then the output driver can manifest and recover all structures. An iterative output driver which uses constant space, such as out, then replaces the recursion stack when the recurring operation is the construction of a structure.

The results recorded here are part of a project which originally had as its goal the compilation of stylized recursive code, in which the constructor played an important role, into stackless iterative code. It now appears that the interpretation scheme proposed here solves the problem of stackless interpretation of code keyed on such constructors, when the constructors can be suspended and treated very similarly to basis conditions. Under this scheme such stylized code can be interpreted within time and space resources that differ from the requirements of iterative code by at most a constant factor. The goal of efficient execution of recursive source code is now closer, surprisingly because of a result on literal interpretation of such code rather than because of an insightful transformation.

Acknowledgements: We thank Cynthia A. Brown for her careful reading and suggestions.

References

1) W.H. Burge. Recursive Programming Techniques, Addison-Wesley, Reading, MA (1975).

2) R.M. Burstall, J.S. Collins & R.J. Popplestone. Programming in POP-2, Edinburgh Univ. Press, Edinburgh (1971).

3) R.M. Burstall & U. Darlington. Some transformations for deve oping recursive programs. Proc. of Intl. Conf. on Reliable Software, ACM SIGPLAN Notices 10, 6 (June, 1975), 465-472.

4) A. Chandra. Efficient compilation of linear recursive programs. STAN-CS-72-282, Stanford Artificial Intelligence Project.

5) G.E. Collins. A method for overlapping and erasure of lists. Comm. ACM 3, 12 (December, 1960), 655-657.

6) J. Darlington & R.M. Burstall. A system which automatically improves programs. Acta Informat. 6, 1 (March, 1976), 41-60.

7) M.J. Fischer. Lambda calculus schemata. Proc. ACM Conf. on Proving Assertions about Programs, ACM SIGPLAN Notices 7, 1 (January, 1972), 104-109.

8) D.P. Friedman and D.S. Wise. CONS should not evaluate its arguments. In S. Michaelson and R. Milner (eds.), Automata, Languages and Programming, Edinburgh University Press, Edinburgh (1976), 257-284.

9) D.P. Friedman, D.S. Wise and M. Wand. Recursive Programming through table look-up. Proc. ACM Symposium on Symbolic and Algebraic Computation (1976), 85-89.

10) P. Henderson & J. Morris, Jr. A lazy evaluator. Proc. 3rd ACM Symp. on Principles of Programming Languages (January, 1976), 95-103.

11) C. Hewitt, P. Bishop, R. Steiger, I. Grief, B. Smith, T. Matson, & R. Hale. Behavioral semantics of nonrecursive control structures. Proc. Colloq. sur la Programmation, Springer-Verlag, Berlin (1974), 385-407.

12) C.E. Hewitt & B. Smith. Towards a programming apprentice, IEEE Trans. Software Engineering SE-1, 1 (March, 1975), 26-45.

13) K. Jensen & N. Wirth. PASCAL User Manual and Report (2nd Ed.), Springer-Verlag, New York (1975).

14) D.E. Knuth. Fundamental Algorithms (2nd Ed., 2nd Printing), Addison-Wesley, Reading, MA (1975), 330-331, 412, 562.

15) P.J. Landin. A correspondence between ALGOL 60 and Church's lambda notation, part I. Comm. ACM 8, 2 (February, 1965), 89-101.

16) J. McCarthy, P.W. Abrahams, D.J. Edwards, T.P. Hart, & M. E. Levin. LISP 1.5 Programmer's Manual, M.I.T. Press, Cambridge, MA (1962), Chapter 1.

17) A.R. Meyer & D.M. Ritchie. The complexity of loop programs. Proc. ACM Natl. Conf., Thompson, Washington (1967), 465-469.

18) T. Risch. REMREC--a program for automatic recursion removal in LISP. Dept. of Comp. Sci., Uppsala Univ., Sweden (1973).

19) H. Schorr & W.M. Waite. An efficient machine-independent procedure for garbage collection in various list structures. Comm. ACM 10, 8 (August, 1967), 501-506.

20) G.L. Steele, Jr. & G.J. Sussman. LAMBDA the ultimate imperative. A.I. Memo 353, M.I.T. Artificial Intelligence Lab (1976).

21) G.J. Sussman & G.L. Steele, Jr. SCHEME: an interpreter for extended lambda calculus. A.I. Memo 349, M.I.T. Artificial Intelligence Lab (1976).

22) S.A. Walker & H.R. Strong. Characterizations of flowchartable recursions. J. Comput. Systems Sci. 7, 4 (August, 1973), 404-407.

23) J. Weizenbaum. Symmetric list processor. Comm. ACM 6, 9 (September, 1963), 524-544.

24) D.S. Wise, D.P. Friedman, S.C. Shapiro & M. Wand. Boolean-valued loops. BIT 15, 4 (December, 1975), 431-451.

```
TYPE pointer = ↑node;
     node = PACKED RECORD
                 suspended: BOOLEAN;
                 refct: 0..memsize;
                 CASE atom: BOOLEAN OF
                      TRUE( pname: chars);
                      FALSE( ar, dr: pointer )
                 END
            END

FUNCTION car(Q: pointer): pointer;
VAR P: pointer;
IF NOT Q↑.ar↑.suspended THEN car := borrow(Q↑.ar)
ELSE P := Q↑.ar;
     car := Q↑.ar := borrow(eval(P↑.ar,P↑.dr));
     erase(P)
FI
```

Figure 1. Node structure and the LISP function car (cdr is similar). The functions borrow and erase are from [5]. The function eval is from [16] with changes from [8].

```
PROCEDURE out (Q: pointer);
VAR STACK, P: pointer;
IF Q = NIL THEN write( '()' )
ELSE
   Q↑.refct := Q↑.refct - 1;
   IF Q↑.atom THEN print?dispose(Q)
   ELSE
      write( '(' );
      STACK := NIL;
      REPEAT
         IF REPEAT
                 P := Q;
                 Q := car(P);
               WHILE Q ≠ NIL;
                 IF P↑.refct = 0 THEN Q↑.refct := Q↑.refct - 1 FI;
               UNTIL Q↑.atom;
                 write( '(' );
                 P↑.ar := STACK;
                 STACK := P
               TAEPER
         THEN print?dispose(Q)
         ELSE write( '()' )
         FI;
      UNTIL REPEAT
                 Q := cdr(P);
               WHILE IF Q = NIL THEN true(Q := P)
                     ELSE true(IF P↑.refct = 0
                               THEN Q↑.refct := Q↑.refct - 1;
                                    dispose(P)
                               ELSE P↑.suspended := TRUE;
                                    P↑.dr := STACK
                               FI
                              ) & IF Q↑.atom
                                  THEN true(write( ' . ' );
                                            print?dispose(Q)
                                           )
                                  ELSE false(write( ' ' ));
                 write( ')' );
               UNTIL REPEAT
                     UNTIL STACK = NIL;
                        P := STACK;
                     WHILE P↑.suspended;
                        STACK := P↑.dr;
                        P↑.suspended := FALSE;
                        P↑.dr := Q;
                        Q := P
                     TAEPER
                 STACK := P↑.ar;
                 IF P↑.refct ≠ 0 THEN P↑.ar := Q FI
               TAEPER
      TAEPER
   FI
FI
```

Figure 2. The printer/coercer/storage manager which drives the interpreter.