13. Gat, I., and Saal, H.J. Memoryless execution: A programmer's viewpoint. IBM Tech. Rep. 025, IBM Israeli Scientific Ctr., Haifa, March 1975.
14. Graham, G.S., and Denning, P.J. Protection – principles and practice. Proc. AFIPS 1972 SJCC, Vol. 40, AFIPS Press, Montvale, N.J., pp. 417–429.
15. Gries, D. *Compiler Construction for Digital Computers*, Wiley, New York, 1971.
16. Harrison, M.A., Ruzzo, W.L., and Ullman, J.D. On protection in operating systems. Proc. Fifth Symp. on Operating Systems Principles, Operating Syst. Rev. (ACM SIGOPS Newsletter) *9*, 5 (Nov. 1975), 14–24.
17. IBM. System/360 PL/I (F) Language Reference Manual. Rep. No. GC28-8201-3, IBM Systems Reference Library, 1971.
18. Jones, A.K. Protection in programmed systems. Ph.D. Th., Carnegie-Mellon U., Pittsburgh, Pa., June 1973.
19. Jones, A.K., and Lipton, R.J. The enforcement of security policies for computation. Proc. Fifth Symp. on Operating Systems Principles, Operating Syst. Rev. (ACM SIGOPS Newsletter) *9*, 5 (Nov. 1975), 197–206.
20. Lampson, B.W. A note on the confinement problem. *Comm. ACM 16*, 10 (Oct. 1973), 613–615.
21. Lipner, S.B. A comment on the confinement problem. Proc. Fifth Symp. on Operating Systems Principles, Operating Syst. Rev. (ACM SIGOPS Newsletter) *9*, 5 (Nov. 1975), 192–196.
22. Lowry, E.S., and Medlock, C.W. Object code optimization. *Comm. ACM 12*, 1 (Jan. 1969), 13–22.
23. Millen, J.K. Security Kernel Validation in Practice. *Comm. ACM 19*, 5 (May 1976), 243–250.
24. Minsky, M.L. *Computation: Finite and Infinite Machines*, Prentice-Hall, Englewood Cliffs, N.J., 1967.
25. Moore, C.G. III. Potential capabilities in ALGOL-like programs. TR 74-211, Dep. Comptr. Sci., Cornell U., Ithaca, N.Y., Sept. 1974.
26. Rotenberg, L.J. Making computers keep secrets. Ph.D. Th., MAC-TR-115, Project Mac, M.I.T., Cambridge, Mass., Feb. 1974.
27. Schroeder, M.D. Cooperation of mutually suspicious subsystems in a computer utility. Ph.D. Th., MAC-TR-104, Project MAC, Sept. 1972.
28. Stone, K.S. *Discrete Mathematical Structures and Their Applications*. Science Research Associates, Chicago, 1973.
29. Walter, K.G., et al. Structured specification of a security kernel. Proc. Int. Conf. on Reliable Software, SIGPLAN Notices (ACM) *10*, 6 (June 1975), 285–293.
30. Weissman, C. Security controls in the ADEPT-50 time-sharing system. Proc. AFIPS 1969 FJCC, Vol. 35, AFIPS Press, Montvale, N.J., pp. 119–133.
31. Wirth, N. The programming language Pascal. *Acta Informatica 1*, 1 (1971), 35–63.

Programming Languages
J.J. Horning* Editor

Shifting Garbage Collection Overhead to Compile Time

Jeffrey M. Barth
University of California at Berkeley

This paper discusses techniques which enable automatic storage reclamation overhead to be partially shifted to compile time. The paper assumes a transaction oriented collection scheme, as proposed by Deutsch and Bobrow, the necessary features of which are summarized. Implementing the described optimizations requires global flow analysis to be performed on the source program. It is shown that at compile time certain program actions that affect the reference counts of cells can be deduced. This information is used to find actions that cancel when the code is executed and those that can be grouped to achieve improved efficiency.

Key Words and Phrases: garbage collection, global flow analysis, list processing, optimization, reference counts, storage management
CR Categories: 3.80, 4.12, 4.20, 4.34

Introduction

Heap storage no longer accessible from program variables has traditionally been collected either by garbage collection or by a reference count scheme [5]. Garbage collection involves a periodic disruption of program execution, during which any one of several well-known scan, mark, and collect algorithms can be employed. Reference counting, although less disrup-

* Note. This paper was submitted prior to the time that Horning became editor of the department, and editorial consideration was completed under the former editor, Ben Wegbreit.

Research sponsored by National Science Foundation Grant DCR74-07644-A01. Author's address: Electronic Research Laboratory, College of Engineering, University of California at Berkeley, Berkeley, CA 94720.

tive, normally requires substantial storage overhead. Both schemes are expensive in time. A garbage collection cycle requires time proportional to used storage, and reference counts need to be updated each time a pointer is created or destroyed.

In a recent paper by Deutsch and Bobrow, a combination garbage collection and reference count scheme is proposed [3]. Their scheme maintains reference counts in a way that can be expected to require less space than usual. It has the property that the counts need to be updated far less often than by traditional methods. Moreover, their method is incremental; hence, unlike garbage collection, it is not disruptive of real time computation.

Their scheme, which assumes a LISP environment, works as follows: Unused list entities, called cells, are chained together on a free list. Associated with each program-reachable cell is a count of the number of references to that cell which originate from cells. That is, references that arise directly from program variables are not counted. Counts are maintained in one of three ways:

(1) Addresses of cells with reference count zero are kept in a hash table, the Zero Count Table (ZCT).

(2) Addresses of cells with reference count exceeding one are kept in another hash table with their counts. This is called the Multiple Reference Table (MRT).

(3) Reference counts for cells with count one are not recorded. Statistical evidence indicates that in LISP programs 90–98 percent of the accessible cells fall into the third category [2].

Given these tables, storage reclamation is straightforward. A cell can be freed if its address is in the ZCT and no program variable directly points to it.

The tables are assumed to reside on backing storage. When a cell is grabbed from the free list, or when a pointer in a cell is altered, a transaction is written to a sequential transaction file. There are three possible transactions:

(1) ALLOCATE—Enter a cell address in the ZCT. This is associated with allocating a new cell.

(2) CREATEREF—If the cell address is in the ZCT, then delete it; otherwise if the cell is in the MRT, then increment its count, and if it is not, then add its address to the MRT with count 2.

(3) DELETEREF—Inverse of CREATEREF.

Periodically the tables are brought into core and the transaction file is processed. In the garbage-collection-like phase of the reclamation, the variables are scanned against the ZCT to determine which cells can be freed.

There are two factors that contribute to the overhead of this scheme. First, a program statement which results in a transaction uses time on each execution to write onto the transaction file. Each of these transactions will subsequently require processing by the storage reclamation system. The second source of overhead is space for the tables in which counts differing from 1 are kept.

The remainder of this paper presents methods for generating fewer transactions based on compile time program analysis. These optimizations are accomplished by finding pairs of cancelling transactions, by grouping transactions into larger, more efficient batches, and by keeping reference counts only for certain classes of cells. Storage for the tables can also be lessened in some cases.

Garbage collection typically consumes a substantial percentage of the total computation time for list processing programs [1]. The aggregate effect of these techniques is to partially shift storage reclamation costs away from a program's run time overhead. Example programs in a later section demonstrate that in some cases the improvement is dramatic, eliminating virtually all overhead. In all cases the computation time performance of a program is not degraded.

Assumptions and Syntax Conventions

In order to consider compile time optimization, it is necessary to assume a language with a compilation process distinct from program execution. In all cases we must be able to recognize syntactically when pointers are being manipulated. We assume that the language maintains an area of storage, commonly called the heap, in which all list cells are stored and into which all pointers point.

Examples are illustrated in Pascal [4]. Syntactically, variables like P, P1, and SOMEPOINTER should be considered to have been declared as pointer variables. The pointer dereference symbol is ↑. Record fields follow dots, and we use F and CAR for the examples. Thus P↑.F is a field of a cell pointed to by P. The Pascal procedure NEW assigns to its argument the address of a freshly allocated cell of the type pointed to by that variable. All uninitialized pointers have been set to NIL. Comments are delimited by * and *.

A transaction may be considered to be generated by inline code at a point in the program text. Sometimes for clarity we explicitly write the transaction code with the source text. There are default transactions associated with some program statements. In particular, NEW(P) becomes

```
NEW(P)
ALLOCATE P↑,
```

and for assignments P↑.CAR := P2 becomes

```
DELETEREF to P↑.CAR↑
P↑.CAR := P2
CREATEREF to P↑.CAR↑
```

When additional information is available at compile time, the inline code will be varied or moved accordingly. Suppose, in the above sequence, P↑.CAR is known to be NIL before the assignment. The code can then be altered to:

```
P↑.CAR := P2
CREATEREF to P↑.CAR↑
```

This saves generation and processing of one transaction.

Regarding this process as a transformation on the program text augmented by the underlying transaction code, we specify two criteria necessary to preserve the program's semantics. A transformation is *safe* if it does not allow a cell which is still accessible to be put on the free list. A transformation is *effective* if it does not delay reclamation of a cell beyond the point in the dynamic execution of the program at which the cell becomes inaccessible.

Cancelling Transactions – Discussion

There are four ways in which the effect of a transaction may be cancelled by events which follow it sequentially. They are:

(1) ALLOCATE followed by CREATEREF – Unoptimized execution would entail entering the cell address in the ZCT for the ALLOCATE and removing it for the CREATEREF. The code sequence may be transformed so that no transactions are generated.

(2) ALLOCATE followed by a loss of all variable references to the cell – The default execution is to make an entry in the ZCT, deleting it in the process of placing the cell on the free list. An optimization of this sequence is to suppress the ALLOCATE and provide inline code in the program to return the cell to the free list at the point where the last variable reference is lost.

(3) CREATEREF followed by DELETEREF – This has no total effect.

(4) DELETEREF followed by CREATEREF – They cancel.

Clark and Green indicate that two frequent occurrences in a program are that a cell is created and "nailed down" shortly thereafter and that a cell is created and quickly discarded, which correspond to the first two kinds of cancellation [2]. Deutsch and Bobrow suggest that the former case be treated by maintaining a buffer of the transaction file in core, hash addressed, so that the CREATEREF transaction may delete the ALLOCATE transaction. Unfortunately this causes the CREATEREF transaction to be nontrivial computationally. This combination is almost always easy to detect at compile time:

```
NEW(P1↑.CAR)

or

NEW(P)          (* would generate an ALLOCATE transaction*)
  .             (*nothing bad intervening*)
  .
P1↑.CAR := P    (*would generate a CREATEREF*)
```

An additional savings is reaped since it is now doubtful that the more expensive CREATEREF transaction proposed by Deutsch and Bobrow is still justified. The second kind of cancellation is typified by the variable which points to the cells being assigned or going out of scope.

The cancellation of CREATEREF by DELETEREF is illustrated in the following example:

```
P↑.CAR := P2
  .             (*nothing bad intervening*)
  .
P↑.CAR := P3
```

Example programs. Two example list processing programs are presented which illustrate the optimizing transformations discussed in the previous sections. The transformations are pointed out in comments at the appropriate points in the source text.

The first example is a complete program that insertion sorts an arbitrary number of integers into a singly linked list. The list is initialized to have two cells. The first contains −MAXINT, the largest negative integer representable in the system, and the second contains MAXINT, the largest positive integer. The addition of these two cells removes the special case checks for adding cells to the front of the list and for reaching the end of the list.

```
PROGRAM INSERTIONSORT(INPUT,OUTPUT);
TYPE LIST = ↑RECORD
        CAR:INTEGER;
        CDR:LIST;
      END;
VAR  LISTOFNUMBERS:LIST;
     I:INTEGER;
PROCEDURE INSERT(VAL:INTEGER;L:LIST);
  (* parameters by value *)
VAR P:LIST;
BEGIN
  WHILE L↑.CDR↑.CAR < VAL DO
    L := L↑.CDR;          (* Assignment to variable, no transac-
  NEW(P);                    tion *)
  P↑.CAR := VAL;
  P↑.CDR := L↑.CDR; (* P↑.CDR is known to be NIL; do
                       not generate a DELETEREF on
                       P↑.CDR before this statement *)
  L↑.CDR := P;        (* there is a CREATEREF/DELE-
                       TEREF cancellation on L ↑ .CDR
                       from the previous statement *)
                      (* there is ALLOCATE/CREATEREF
                       cancellation on P from the NEW
                       statement to here *)
END;
BEGIN (* MAIN *)
  NEW(LISTOFNUMBERS);
  LISTOFNUMBERS↑.CAR := −MAXINT;
  NEW(LISTOFNUMBERS↑.CDR); (* ALLOCATE/CRE-
                              ATEREF cancellation *)
                           (* LISTOFNUMBERS ↑ .
                              CDR known to be NIL
                              before execution so no
                              DELETEREF *)
  LISTOFNUMBERS↑.CDR↑.CAR := MAXINT;
  WHILE NOT EOF(INPUT) DO
    BEGIN
      READ(I);
      INSERT(I,LISTOFNUMBERS);
    END;
END. (* MAIN *)
```

If the optimizations mentioned in the comments are performed at compile time, the execution of the program will result in the generation of exactly one trans-

action. This is the ALLOCATE that arises from the first NEW statement in the main program. If ALLOCATE followed by loss of all variable references to a cell is checked, then it is a simple special case to check for cells which are referenced by variables and never lost. If this optimization is included, the program generates no transactions at run time.

The second example is a subroutine that iteratively copies a list. It maintains a pointer to the last cell of the copy being created.

```
FUNCTION COPY(L:LIST); (* parameter by value *)
VAR P,LAST:LIST;
BEGIN
  IF L = NIL THEN COPY := NIL
  ELSE
    BEGIN
      NEW(LAST);
      LAST↑.CAR := L↑.CAR;
      COPY := LAST;
      WHILE L↑.CDR ≠ NIL DO
        BEGIN
          L↑ := L↑.CDR;
          NEW(P);
          P↑.CAR := L↑.CAR;
          LAST↑.CDR := P; (* ALLOCATE/CREATEREF
                             cancellation *)
          LAST := P;
        END;
    END;
END;
```

Assuming that the optimizer is able to deduce that LAST ↑ .CDR is always NIL before it is assigned P, the above function generates only one transaction, namely that for the NEW statement at the beginning of the ELSE clause.

Cancelling Transactions – A Sample Algorithm

This section presents an algorithm for finding cancellations between ALLOCATE and CREATEREF. This algorithm also detects ALLOCATE followed by loss of access to a cell. It is a sample of how one would computationally implement the transformations suggested in this paper.

The presentation is informal. It assumes that the program is represented by a directed graph (flow graph) with $|E|$ edges and $|N|$ nodes. Program statements reside at the nodes. In particular, there are $|P|$ nodes that contain NEW statements. We call the set $\{u \mid (u, v)$ an edge in the graph$\}$ the *parents* of v.

The term "variable" is used to refer to an assignable entity which, by convention, is not in the heap. This reflects the intention not to count references that arise from variables. By the generic term "assignment statement" we capture all program actions with the semantic effect of changing a value that can be interrogated by the program. Assignments that are implied, such as a variable going out of scope being assigned UNDEFINED and a parameter being initialized, are assumed to be explicitly present in the flow graph. The uniform convention is adopted that the semantic effect of an assignment statement is that something called LHS is altered to agree with a value RHS.

As a simplification, the language is assumed to have no calls to user subroutines. Techniques commonly used in optimization to relax this restriction apply to this problem as well [6].

The ALLOCATE/CANCELLATION algorithm is based on the observation that a cell freshly allocated at a NEW statement, NEWCELL, can be kept track of at compile time until first pointed to from the heap. The algorithm deduces the variable names which are definitely known to point to NEWCELL on an edge E of the flow graph. It remembers them in a bit vector set E.DS (definitely set). Similarly names which may point to NEWCELL, that is are not known not to point to NEWCELL, are recorded in set E.MS (maybe set). For any edge, DS $\subseteq$ MS. Along a flow path originating at the NEW statement, one of three cases will arise:

(1) A point is reached where it is determined that a reference to NEWCELL is created in the heap. This is when ALLOCATE is cancelled at compile time by CREATEREF.

(2) A point is reached where it is determined that no variable still points to NEWCELL, and case (1) has not occurred. This is ALLOCATE cancelled by the actual loss of access to the cell.

(3) Inconclusive information at compile time.

The algorithm takes a program flow graph and inserts safe and effective transactions for all the cell allocations. The program is organized as follows:

MAIN iterates over each NEW statement in the program.

NUMBERNODES computes a modified depth-first ordering on some of the nodes of the flow graph, starting at a particular NEW statement [7]. The numbering has the property that no node, except the NEW statement, is numbered until all its parents have been numbered. For example, see Figure 1.

COMPUTEMSDS is outlined. It uses the numbering to find the set of variable names which definitely point to the last cell allocated, as well as the set of those which may point to it.

ENTERTRANSACTIONS is outlined. It walks flow paths originating at the NEW statement. As it proceeds down these paths it looks for cancellations.

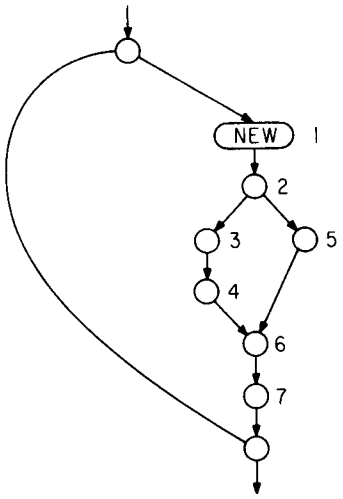
ALLOCATE/CANCELLATION Algorithm

```
(* Enters all transactions for allocation needed in program *)
BEGIN (* MAIN *)
  FOR PICKEDNEW := each NEW statement in the program DO
    IF argument to PICKEDNEW is a variable THEN
      BEGIN
        NUMBERNODES(PICKEDNEW,2);
        (* Assume PICKEDNEW is numbered 1 *)
        COMPUTEMSDS;
        ENTERTRANSACTIONS;
      END;
END; (* MAIN *)
```

```
PROCEDURE NUMBERNODES(U:NODE; VAR I:INTEGER);
BEGIN
  FOR each edge (U,V) DO
    IF V unnumbered and all parents of V are numbered THEN
      BEGIN
        Assign number I to V;
        I := I+1;
        NUMBERNODES(V,I);
      END;
END; (* NUMBERNODES *)
PROCEDURE COMPUTEMSDS;
BEGIN
  Label the edge directed out of PICKEDNEW with DS and MS sets
  initialized to the variable which is its argument. Visit nodes in the
  ordering computed above. At a node, compute the DS and MS sets
  for the edges directed out of the node by: intersection of the
  incoming DS sets (already computed); union of the incoming MS
  sets (already computed); and observing whether the node is some
  kind of assignment statement which modifies the information
  about which names reference the last allocated cell. For example, if
  the node is an assignment to an MS variable and the RHS is not an
  MS name, then the LHS can be removed from the MS and DS sets
  on the outwardly directed edges. In general, the membership of
  LHS in the sets associated with edges directed out of the node is
  made to match that of RHS in these sets.
END; (* COMPUTEMSDS *)
PROCEDURE ENTERTRANSACTIONS;
BEGIN
  (* The presence of a marker on an edge means "an ALLOCATE
  transaction is owed when this edge is traversed in the execution of
  the program" *)
  Place a marker on the edge directed out of PICKEDNEW;
  Visit nodes in the ordering computed above, starting at 2.
  At each node DO
    IF all edges directed into the node are marked THEN
      BEGIN
        Check to see if cancellation occurs at this node. This might
        occur in one of two ways:
        (1) The node is an assignment statement with RHS in all
        the incoming DS sets and LHS a heap reference. This is
        ALLOCATE/CREATEREF cancellation. Erase markers
        on all incoming edges and do not mark the outgoing edges.
        (2) The node is an assignment statement which assigns to a
        variable that previously referenced the last allocated cell. In
        particular, if the MS sets on the leaving edges are empty,
        this must be the last reference and it is an instance of
        ALLOCATE cancelled by loss of access to the cell. Erase
        markers on all incoming edges. Do not mark the leaving
        edges, but produce inline code to free the cell being re-
        leased.
        If no cancellation occurs, check whether the ALLOCATE
        can be delayed effectively beyond this node. If the DS sets
        on the outgoing edges are nonempty, the cell must still be
        accessible and delaying the ALLOCATE is effective. Also,
        check that the node is not an assignment to a heap reference
        from an MS variable which is not also a DS variable. This
        would reflect compile time uncertainty about cancellation.
        If these checks allow the ALLOCATE to be delayed, erase
        the markers on all the incoming edges and mark all the
        edges leaving this node. Otherwise, leave the markers as
        they are.
      END;
  Insert an ALLOCATE transaction on each marked edge. This
  catches the cases in which there is insufficient compile time infor-
  mation to detect cancellation.
END; (* ENTERTRANSACTIONS *)
```

The running time of the ALLOCATE/CANCELLATION algorithm can be held to $O(|P|\,|E|)$ bit vectors steps. Bit vector operations are used to manipulate the sets of variable names. The timing analysis is simple. MAIN iterates $|P|$ times. The recursive calls on NUMBERNODES can be executed in $O(|E|)$ time by implementing the test "all parents of V are numbered" with a counter. All other computation is constant with respect to edges directed out of numbered nodes. Similarly COMPUTEMSDS and ENTERTRANSACTIONS use at most constant time for each graph edge.

Fig. 1.



The transformations generated by the ALLOCATE/CANCELLATION algorithm are safe. The only way that a cell can be collected is if its address is in the ZCT. Addresses get entered into this table by ALLOCATE and DELETEREF transactions. By delaying the ALLOCATE transaction, nothing additional is entered into the ZCT. No DELETEREF can be issued for a cell with a pending, but unexecuted ALLOCATE transaction since no ALLOCATE transaction is left owed beyond the point where a heap pointer to the cell might exist. An additional way that a cell can be put on the free list is through added inline code to free the cell. This is only done when no heap pointer is known to exist and no variable could still point to the cell; hence this too is safe.

The transformations generated by the ALLOCATE/CANCELLATION algorithm are effective since the only time that a cell can be collected is when no variable points to it. By ensuring that DS is nonempty on every edge of the flow graph for which a transaction is owed, effectiveness is ensured.

Note that the ALLOCATE/CANCELLATION algorithm involves a time-space tradeoff. The ALLOCATE transaction may be generated at several places rather than just one. Only one instance of these will be executed, and in many cases not even that. Also, inline code to free cells may appear repeatedly. One may opt to save the program space if the solution to the flow problem indicates that more than some number of code insertions would need to be done.

Batching

Another class of transformations involves creating larger, more efficient transactions. In its simplest form, batching is accomplished by enhancing the transaction repertoire to include (1) create N references to a cell (CREATENREFS) and (2) delete N references to a cell (DELETENREFS). The motivation for these extensions is that at user run time only one transaction needs to be written to the transaction file, saving time. In addition, the collection time processing for these extended transactions appears to be no greater than for simple ones. Since there will be fewer extended transactions, this is a collection time economy.

Batching can be implemented by a variation of the global flow technique presented in the last section. Ensuring that the transformations are safe and effective is somewhat more intricate but can be accomplished.

Invariants

The final group of optimizations attempts to capitalize on information that is kown about entire classes of cells. In particular, it would be extremely worthwhile to know at compile time that no member of a class of cells will ever be released at run time. If such information were available, all transactions that operate on cells of this class could be suppressed. This saves execution time as well as space in the storage management tables.

There are several variations on the general idea of knowing that a class of cells is never released. Unfortunately there appears to be no simple automatic way to collect the global information needed for this optimization. Programmers are usually aware of macro properties of data structures such as the fact that a structure is built and never released. It is thus reasonable to allow the user to input this information into the compiler.

Conclusion

Automatic storage reclamation, when viewed in the Deutsch-Bobrow transaction model, can benefit from compile time optimization. Some amount of this can be done automatically by using global flow analysis.

Several sample programs were inspected in an attempt to evaluate the effectiveness of the transformations in reducing run time storage reclamation overhead. Eight Pascal programs with diverse list structure applications were obtained from cooperating students. It appeared that detecting pointers that are NIL before assignment would have resulted in the saving of many transactions. Practically every ALLOCATE transaction observed could have been cancelled at compile time. The one exception which appeared was code that looked like NEW(A[I]).

These transformations are easy to detect automatically in practically all cases in which they appear.

CREATEREF/DELETEREF cancellation and batching appear in many contexts that are impossible to detect automatically. Consider the code sequence:

```
P↑.CAR := SOMEPOINTER
P1↑.CAR := P2
P↑.CAR := ANYPOINTERORNIL
```

Should P = P1 on any execution of this sequence, the cancellation transformation would produce wrong results. Batching has similar limitations arising from the existence of a cell already referenced from the heap which is difficult to keep track of at compile time. The instances in which batches are found seem to correspond to data structures with multiple links and are usually in inner loops.

It would be impossible to state a percentage of expected improvement that could be achieved by using these transformations. As the examples illustrate, there are certain programs for which the execution time savings is dramatic. It is also clear that additional hints from the programmer can be put to good use in cases that the information gathered by global flow analysis is not sufficient. This might take the form of supplying assertions about invariants or by allowing the programmer some means of hand supplying the transactions for critical sections of his program.

The kinds of issues addressed in this paper are just beyond the range of what is considered practical in running language systems. A problem which is normally thought of totally in the context of run time has been viewed in a different perspective. Eventually it will pay to routinely design compiler systems that have built in machinery to handle the necessary bookkeeping for global flow analysis. At that time the services of compiler systems can be made far more extensive than what is currently available and the optimizations suggested here should become practical.

Acknowledgments. I would like to thank Susan L. Graham of the University of California, Berkeley, and Mark Wegman of IBM Research, Yorktown Heights, for their valuable assistance at different stages in the preparation of this paper.

Received June 1975; revised April 1976

References

1. Boom, H. Experience with the use of ALGOL W as SIL. *Algol Bulletin. 37* (July 1974), 63-67.
2. Clark, D., and Green, C.C. An empirical study of list structure in LISP. *Comm. ACM 20*, 2 (Feb. 1977), 78-87.
3. Deutsch, L.P., and Bobrow, D.G. An efficient incremental automatic garbage collector. Tech. Rep., Xerox Palo Alto Res. Ctr., Palo Alto, Calif., Jan. 1975; also, *Comm. ACM 19*, 9 (Sept. 1976), 522-526.
4. Jenson, K., and Wirth, N. *Pascal User Manual and Report.* Springer-Verlag, New York (1974).
5. Knuth, D.E. *The Art of Computer Programming, Vol. 1: Fundamental Algorithms.* Addison-Wesley, Reading, Mass., 1969.
6. Rosen, B.K. Data flow analysis for recursive PL/1 programs. IBM Res. Rep., IBM, Yorktown Heights, N.Y., Jan. 1975.
7. Tarjan, R.E. Depth first search and linear graph algorithms. *SIAM J. Comptng. 1*, 2 (1972), 146-160.