SHIFTING GARBAGE COLLECTION OVERHEAD TO COMPILE TIME

by

Jeffrey M. Barth

Memorandum No. ERL-M524

17 June 1975

ELECTRONICS RESEARCH LABORATORY

College of Engineering
University of California, Berkeley
94720

SHIFTING GARBAGE COLLECTION OVERHEAD TO COMPILE TIME[†]

Jeffrey M. Barth

Computer Science Division
Department of Electrical Engineering and Computer Sciences
and the Electronics Research Laboratory
University of California at Berkeley

June 1975

Abstract

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 optimations will require global flow analysis to be performed on the source program. Also discussed is a way of integrating these techniques into the source language to give the programmer extensible garbage collection.

Keywords and Phrases: garbage collection, global flow analysis, list processing, optimization, reference counts, storage management

CR Categories: 3.80, 4.12, 4.20, 4.34

[†]Research sponsored by National Science Foundation Grant DRC74-07644-A01.

Introduction

The process of collecting heap storage no longer accessible from program variables has traditionally been done either by garbage collection or by reference counts [K]. 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 disruptive, normally requires substantial storage overhead. Both schemes are expensive in time, garbage collection requiring time proportional to used storage and reference counts which need to be updated each time a pointer is created or destroyed.

In a recent paper by Deustch and Bobrow a combination garbage collection and reference count scheme is proposed [DB]. 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 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 used 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. The counts are maintained in one of three ways:

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

2) Addresses of cells with reference count exceeding one are kept in another hashtable along with their counts. This table is called the

Multiple Reference Table (MRT).

3) Cells with count one are ignored.

Statistical evidence indicates that in LISP programs a vast percentage of the accessible cells fall into the third category [CG].

Given these tables, storage reclamation is straight-forward. 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.

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, otherwise add its address to the MRT with count two.

3) DELETEREF -- the 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.

The principal costs of this scheme fall into two general categories. Firstly, a program statement which causes a transaction uses time on each execution to write onto the transaction file. Secondly, each transaction will subsequently require processing. Methods for generating fewer transactions, as well as for decreasing the amount of necessary processing at collection time, are discussed in the remainder of this paper. Various kinds of canceling transactions and ways to find them at compile time are explored. Two levels of generating fewer, larger transactions are investigated. Machinery is outlined for determining which variables need not be scanned against the ZCT and for finding classes of cells for which counts are totally unnecessary. Finally, a mixed automatic and programmer controlled storage reclamation scheme is presented which is a natural extension of the previous sections.

Garbage collection typically uses a substantial percentage of the total computation time for list processing programs. The aggregate effect of these techniques is to partially shift storage reclamation costs away from a program's run time overhead. In some cases the improvement will be dramatic, eliminating virtually all overhead, but in all cases the computation time performance of the program will not be degraded.

Assumptions and Syntax Conventions

In order to talk about compile time considerations, it is necessary to assume a language with a compilation process distinct from program execution. Moreover, since the intent is to examine the program text statically and deduce useful information, a language with static naming structure is assumed. It will be possible to say more about programs written in a language with typed variables, but in all cases it is necessary to be able to recognize syntactically when pointers are being manipulated. We assume that the language maintains an area of the address space, commonly called the heap, in which all list cells are stored and into which all pointers point. ALGOL 68 is one of the few languages that allows other variations.

Examples will be illustrated in PASCAL which meets all the criterion mentioned above [WJ]. The noninteractive nature of PASCAL is not essential to anything that follows.

Using the Deutsch-Bobrow method in a language with static naming structure introduces no added complications. Note merely that all cells pointed to by variables on the run time stack are potentially accessible, not only those statically addressable. Generalizing to cells of variable size and internal structure will entail having additional information available at collection time, but this will not be pursued further here.

Deutsch and Bobrow cite Clark and Green's study of LISP list structure which indicates the overwhelming percentage of singly referenced cells. We have no way of knowing whether this remains true in the kind of language environment being considered here.

Syntactically, variables like P, P1, and SOMEPOINTER should be considered to have been declared as pointer variables. The pointer dereference is ↑. Record fields follow dots and we will use F and CAR for the examples. 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 (* and *). In presenting algorithms, the COND statement selects the first TRUE boolean expression and executes the statement following it.

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 will be default transactions associated with some program statements. In particular,

```
        NEW(P)
```

becomes

```
          NEW(P)
          ALLOCATE address pointed to by 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 in-line 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↑
```

saving a transaction generation and processing.

Regarding this process as a transformation on the program text augmented by the underlying transaction code, we specify two criterion necessary to preserve the program's semantics. A transformation is <u>safe</u> if it does not allow a cell to be put on the free list which is still accessible. A transformation is <u>effective</u> if everything which could have been on the free list before optimization at any given dynamic instant of the program can be on it after optimization. This insures that storage will not be exhausted prematurely because freeing cells is delayed.

Canceling Transactions -- Discussion

There are four ways in which the effect of a transaction may be canceled 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 -- which has no total effect.

4) DELETEREF followed by CREATEREF -- which 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. 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 implies that the CREATEREF transaction is nontrivial computationally. This combination is often 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 is still justified. The second kind of cancellation can also frequently be detected in the same process needed to detect the first. This is typified by the variable which points to the cell 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
```

It is doubtful that enough such occurrences could be found to make the optimization worthwhile. An algorithm is discussed to detect these cancellations because its limitations are informative.

The fourth class of cancellation is included only for completeness. The reader can construct an example to see how contorted code would have to be to give rise to the possibility of detecting a CREATEREF after a DELETEREF on a particular cell.

Canceling Transactions -- Algorithms

This section will present algorithms for finding canceling transactions. The first is presented in detail, including correctness proofs and running time bounds which appear in Appendix A. It illustrates computationally how one would go about actually doing the transformations suggested in this paper. Since subsequent algorithms are patterned on this one, and since the main intent of this paper is to raise ideas on what might be done at compile time rather than to present exactly how to do it, the remaining programs will be sketched only. In practice the running time is one order of magnitude smaller than the worst case, since in program flow graphs nodes tend to be sparsely connected.

We use a variation of the global flow techniques studied by Ullman, Hecht, and Kam [HU][KU]. Nodes in the flow graph are ordered in some fixed ordering, FIXEDORDER, with the program entry node appearing first. The ordering used by Hecht and Ullman will work best in practice, but identically in the worst case to any other ordering. Sets are maintained in bit vectors, and computation steps are row operations on those vectors. The flow graph has $|E|$ edges, $|N|$ nodes, and $|V|$ variables.

We use the term "assignment statement" to 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. Finally, the generic term "variable" refers to an assignable entity which by convention is not in the heap.

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 [R].

The Allocate/Cancellation Algorithm is based on the observation that a cell freshly allocated at a NEW statement, NEWCELL, is easy to keep track of until first pointed to from the heap. The algorithm deduces the variable names which are known definitely to point to NEWCELL on an edge, E, of the flow graph. It remembers them in a 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). Along a flow path originating at the NEW statement one of three things will happen:

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

3) Inconclusive information at compile time.

The algorithm is going to perform best when DS and MS are identical everywhere, which is how a majority of programs can be expected to look. The algorithm takes a program flow graph and inserts safe and effective transactions for all the cell allocations.

The program is organized into routines as follows:

MAIN iterates over each NEW statement.

FINDSETDS expands the set of variable names known to point to NEWCELL on flow graph edges until no more can be found.

FINDSETMS removes variable names from the MS sets as long as there are names that can be shown not to point to NEWCELL.

ENTERTRANSACTIONS effectively walks all flow paths originating at the NEW statements in order, examining the MS and DS sets. It processes each path in the particular way appropriate, depending on which of the three cases above describe it. The effect of these infinitely many walks is achieved by propagating markers associated with each node of the flow graph.

```
ALLOCATE/CANCELLATION ALGORITHM
(* Enters all transactions for allocation needed in program *)
BEGIN
   FOR PICKEDNEW := each NEW statement in program DO
      IF argument to PICKEDNEW is a variable THEN
         BEGIN
            FINDSETDS;
            FINDSETMS;
            ENTERTRANSACTIONS;
         END;
END; (* MAIN *)


PROCEDURE FINDSETDS
BEGIN
   FOR E := each edge in flow graph DO
      E.DS := {all variables in program};

   ENTRYEDGETOPROGRAM.DS := $\emptyset$;

   VISIT all nodes, N, in the flow graph using ordering FIXEDORDER
   UNTIL nothing changes for an entire pass and DO
      COND
         N is the PICKEDNEW:
            EDGESOUTOFN.DS := {argument of PICKEDNEW};

         N is an assignment statement, LHS is a variable,
         and RHS $\notin \cap$EDGESINTON.DS:
            EDGESOUTOFN.DS := $\cap$EDGESINTON.DS - {LHS};

         all other cases:
            EDGESOUTOFN.DS := $\cap$EDGESINTON.DS;
      END (* COND *);
END (* FINDSETDS *);


PROCEDURE FINDSETMS
BEGIN
   FOR E := each edge in flow graph DO
      E.MS := $\emptyset$;

   ENTRYEDGETOPROGRAM.MS := $\emptyset$;

   VISIT all nodes, N, in the flow graph using ordering FIXEDORDER
   UNTIL nothing changes for an entire pass and DO
      COND
         N is the PICKEDNEW:
            EDGESOUTOFN.MS := {argument of PICKEDNEW};

         N is an assignment statement, LHS is a variable,
         and RHS $\in \cup$EDGESINTON.MS:
            EDGESOUTOFN.MS := $\cup$EDGESINTON.MS $\cup$ {LHS};

         all other cases:
            EDGESOUTOFN.MS := $\cup$EDGESINTON.MS;
      END (* COND *)
END (* FINDSETMS *);
```

```
PROCEDURE ENTERTRANSACTIONS
BEGIN
   FOR E := each edge in flow graph DO
      E.MARK := 0;

   FOR E := each edge emanating from node PICKEDNEW DO
      BEGIN
         E.MARK := 2;
         IF node entered by E has all entering edges marked 2
            THEN place entered node on QUEUE;
      END;

   REPEAT
      N := first element on QUEUE;
      DELETE N from QUEUE;
      FOR E := each entering edge of N DO
         E.MARK := 1;
      COND
         N is PICKEDNEW:
            generate an ALLOCATE transaction on each entering edge of
            N using a variable in the DS set of that edge;

         N is an assignment statement, LHS is a heap resident pointer,
         and RHS $\varepsilon$ $\cap$EDGESINTON.DS:
            nothing, this is ALLOCATE followed by CREATEREF.  Note
            somewhere that this assignment should not generate a CREATEREF;

         N is an assignment statement, LHS is a heap resident pointer,
         and RHS $\varepsilon$ $\cup$EDGESINTON.MS:
            on all entering edges generate an ALLOCATE transaction.
            Can't tell if cell is getting a pointer from the heap;

         EDGESOUTOFN.DS = $\emptyset$ and EDGESOUTOFN.MS = $\emptyset$:
            generate code to free the cell formerly pointed to by LHS.
            This is ALLOCATE followed by a loss of the cell.  See proof
            to see why this has to be an assignment statement;

         ENDSOUTOFN.DS = $\emptyset$:
            generate an ALLOCATE on each entering edge to N using some
            variable in the DS set on each edge.  This is leakage in
            the algorithm, since the MS set might go away later, but we
            must insure that a definite pointer to the cell is available
            anywhere a transaction might be generated;

         all other cases:
            FOR E := each edge emanating from N DO
               BEGIN
                  IF E.MARK $\neq$ 1 THEN E.MARK := 2
                  ELSE generate an ALLOCATE transaction on E using
                     a variable in DS;

                  IF all marks on edges entering the node entered by E
                     are 2 THEN put this node on the QUEUE;
               END;
      END; (* COND *)

   UNTIL QUEUE is empty;

   FOR E := each edge marked 2 DO
      generate an ALLOCATE transaction on E using some variable in E.DS;
END (* ENTERTRANSACTIONS *);
```

In the Appendix this algorithm is proven to enter all the transactions needed for cell allocation in a program. The running time is shown to be $O(|P||N||E||V|)$ where $|P| \geq 1$ is the number of NEW statements with variable argument. The transformations induced by the algorithm are proven to be safe and effective.

Note that the Allocation/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 a particular threshold number of code insertions would need to be done. A likely value for the threshold would be one.

The Create/Cancellation Algorithm attempts to find instances of a CREATEREF followed by a DELETEREF on the same address. In principle, it might be possible to see a sequence like:

```
P1↑.F1 := SOMEPOINTER
  .             (* nothing bad intervening *)
   .
    .
P2↑.F2 := ANYPOINTERORNIL
```

and know that P2↑.F2 had value SOMEPOINTER before its assignment. This case of Create/Cancellation will be hard to find except if it looks like:

```
P1↑.F1 := P2↑.F2    (* CREATEREF to P2↑.F2↑ *)
  .             (* nothing bad intervening *)
   .
    .
P2↑.F2 := ANYPOINTERORNIL   (* DELETEREF to P2↑.F2↑ *)
```

which is typical code for inserting a cell in a list structure, where P2↑ is inserted after P1↑.

There are also realistic possibilities for finding the sequence:

```
P↑.CAR := SOMEPOINTER
  .          (* nothing bad intervening *)
  .
P↑.CAR := ANYPOINTERORNIL
```

where the CREATEREF following the first statement cancels the DELETEREF preceding the second.

There are two major problems to deal with in finding Create/Cancellation. We must understand precisely what the vague statement "nothing bad intervening" means and we must take care that safety is retained.

The following sequence illustrates one kind of bad intervening code from which a characterization of all bad intervening code is obtained:

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

Should P = P1 on one execution of this sequence, the cancellation transformation produces wrong results. The second assignment generates a DELETEREF on a cell address for which no CREATEREF was done and no DELETEREF is generated in the third assignment statement for the reference created in the second. In the absence of further information, assignments into the heap in intervening code must be forbidden. This condition alone is strong enough since it assures that the reference not created is the same as the one not deleted by the cancellation.

The safety question arises in the case that SOMEPOINTER↑ may for an interim have exactly one reference and no variables pointing to it. If this is the reference that the transformed sequence fails to create, the cell could be inappropriately collected.

Having specified the potential hazards, the Create/Cancellation Algorithm follows directly. At an assignment into the heap, we "remember" the heap variables assigned to and, if it exists, the heap variable assigned from. Along flow paths, we look for instances in which either one of the heap variables being remembered is assigned. No information is allowed to pass over flow graph nodes that assign into the heap. Given that two potentially canceling transactions are found, a check is made to see that the cell in question always has additional references. If the RHS of the first assignment is a variable, a modification of the FINDSETDS subroutine supplies the information. If the RHS of the first assignment is a heap resident pointer, then it is sufficient to check that this heap pointer is still in scope, since no assignment could have occurred into the heap removing this reference.

The transformation is safe because the construction explicitly guarantees it. It is effective since DELETEREF transactions are only removed to prevent them from decrementing counts that never were incremented.

Discussion

There is a noteworthy difference in the expected impact of the two types of cancellation. The ALLOCATE information propagates with robustness, becoming imprecise at flow graph joins. The CREATEREF information is abruptly halted with the least perturbation in the heap. The underlying reason is that in the latter case a pointer object is in the heap where it is difficult to keep track of.

The finding of Create/Cancellation transformations can be improved in the presence of additional information at compile time. We characterized "bad intervening code" as containing assignments into the heap. Actually, the necessary condition is that we can tell that the reference not created is the one not deleted. That is, we must be sure that the reference is still where we put it at the time the DELETEREF would be issued. Consider:

```
P1↑.F1 = P2↑.F2    (* fields are pointer valued *)
```

in a language in which pointers can be declared of not mutually assignable types. Assignments into the heap can not be "bad" unless the pointer assigned is of the same type as P1↑.F1 and P2↑.F2. Additionally, assigning P3↑.F1 whose type matches P1↑.F1 can not be "bad" if the type of P3 differs from P1. In general, intervening code is "bad" if it includes an assignment into the heap which is both of the same pointer type as the reference being remembered and if the cell being stored into is of the same type as the cell in which the remembered pointer was assigned.

At this point a peculiarity enters into this analysis. The design decisions of the programmer come into play, including efforts he may make to improve the compile time information gathering process. Whereas this may be a controversial point, it seems that whatever node partitioning is done through the types probably adds security and structure to the program.

Batching

The next transformation to be explored is compile time batching. This will take two drastically different forms, referred to as simple batching and heterogeneous batching.

In order to do simple batching, the transaction repertoire is enhanced to include

1) Create N references to a cell (CREATENREFS).

2) Delete N references to a cell (DELETENREFS).

and the MRT is allowed to also maintain counts less than zero. The motivation for these extensions is that at user run time only one transaction need be written to the transaction file, a time savings. 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.

Simple batching can be implemented by a variation of the now familiar global flow techniques. Rather than describe a simple batching algorithm, the issues that must be dealt with are presented.

There are two ways that CREATENREFS can be used in safety. One way is to trace flow paths from a heap assignment to other heap assignments, putting the CREATENREFS transaction after the computation path. Under this scheme, just as in Create/Cancellation, an independent assurance that the node has other references is necessary. The alternative is to work backwards from heap assignments and put the CREATENREFS transaction at the location where the first CREATEREF would have been. This is clearly superior in straight line code, but the former captures some additional cases of batching.

An example code sequence illustrates all of these points:

```
P1↑.CAR := SOMEPOINTER
P2↑.CAR := SOMEPOINTER
IF boolexp THEN P3↑.CAR := SOMEPOINTER
           ELSE stmt
```

Ideally, a CREATE3REFS to SOMEPOINTER↑ is added to the THEN part and a CREATE2REFS to SOMEPOINTER↑ is added to the ELSE part. Consider the case where SOMEPOINTER is a variable. Even if P1 = P2 = P3 and P1↑.CAR = SOMEPOINTER before the code is executed, no unsafe reclamation can occur because SOMEPOINTER prevents the collection while the cell is in the ZCT. Note that the count for SOMEPOINTER↑ will be driven negative if it starts with count one, the P1↑.CAR reference. If SOMEPOINTER is a heap resident pointer, and nothing is known about the cells pointed to by P1, P2, and P3, then at best we can batch the first two CREATEREFs and place the CREATE2REFS before the code sequence.

Fortunately, effectiveness will be preserved regardless of where the CREATENREFS batch is placed. Even if it is done early, it will only be done on cells that are uncollectable until all N references go away.

Similarly, care must be taken with DELETENREFS with respect to effectiveness and safety. More interesting is the observation that it is going to be harder to find DELETENREFS batches. We must find N references, all of which previously pointed to a particular cell. This is in contrast to the CREATENREFS situation in which a pointer value is followed around to see what it gets assigned to.

The expectation is that in list structures with multiple links these batches will be found. Consider adding a new leaf to a tree structure with "sibling" pointers. The added cell is likely to obtain references in batchable proximity.

Heterogeneous batching is the collection of as much information as is possible in the given program text into one specialized transaction. Here the transaction repertoire is amended to include transactions T_i for as many new transactions as are needed in the particular program. The compile time system produces a table for the collection algorithm which specifies what actions are associated with each new transaction. That is, the collection system is viewed as an interpreter executing a program of sequential T_i instructions whose logic is table specified.

Heterogeneous batching completely handles the simple batching case, but with additional collection time overhead. Even if the more inclusive process is carried out, the faster and simpler special cases may be worth separate treatment.

The implementation idea here is to start anywhere in the code, collecting actions along a flow path so long as increasing definite information is available. A new transaction, whose collection time semantics are tailored to the collected information, will be generated somewhere along this path.

In addition to the added collection time complexity, heterogeneous batching raises nontrivial safety and effectiveness questions for the optimizer. Suppose, for example, a DELETENREFS transaction for cell C1 is followed in straight line code by a CREATEMREFS transaction for cell C2. Grouping the two at the latter point may violate effectiveness in delaying the release of cell C1. Adding a third cell, C3, we can exhibit a situation in which there is no location that is both safe and effective for the largest heterogeneous batch, by appending a DELETEREF on C3 to those above:

```
DELETENREFS on C1    (* cannot be delayed effectively *)
CREATEMREFS on C2
DELETEREF on C3      (* cannot be done earlier safely *)
```

For these reasons, automating this optimization is impractical.

Invariants

It may be worthwhile to check at compile time to see if some specific class of cells will never be freed at run time. A typical such class might be cells which permanently hang from a global pointer which is never assigned. The economy aimed for here is in both time and space since no MRT table entry need be made for them.

The primary danger to cope with is that a DELETEREF transaction may spuriously enter the cell's address into the ZCT leading to its unsafe collection. The criterion for this transformation is: There can be no statement of the form

```
        P↑.CAR := ANYPOINTER
```

if there is uncertainty whether P↑.CAR points to a cell for which counts are not being kept before the assignment.

The reader may doubt that such a condition could ever be deduced at compile time. Consider the following typical scenario for building a singly linked list which will never be deallocated. There will probably be a global LISTHEADP and created cells will be linked between it and the former first element. We might well expect that after all of the several possible NEW calls for this cell type, the LINK field is immediately filled, at a time when it is easily seen to be still NIL, and it is subsequently pointed to by LISTHEADP. The former assures that no interesting DELETEREF could be lost and the latter that the list is not reclaimable. This is, in fact, the most direct program to build a list, or for that matter a tree, doubly linked list, tree with backlinks, etc. so long as no deallocation is planned. Extending the example somewhat, it may be possible to notice that something like a balanced binary tree has its LINK fields reassigned, but only in some locally rebalancing way. This example illustrates that some class of cells may be busy, in this case pointers may crawl all over the cells, but no cell of the type in the list ever need have a transaction generated or processed at run time.

We call a class of cells <u>invariant accessible</u> if all existing cells in the class are always accessible and we can effectively prevent any DELETEREF transaction from entering the address of any cell in the class into the ZCT. In general we do not require that CREATEREF be findable and forbidden, but in practice we eliminate all found CREATEREFs. A class of cells which are invariant accessible need not, but often will, correspond to a pointer type. This makes the task of finding the locations that generate DELETEREF and CREATEREF transactions easy.

A satisfying aspect to this optimization is that the programmer who attempts to capitalize on it by partitioning nodes into type classes as finely as possible gets a program with improved security and structure.

More obscure from the point of view of compile time deduction are a host of other invariants. A pointer type may be known invariant accessible with respect to a particular scope. This leads to a situation where counting may be omitted, and all cells of the type be released on scope exit. It is conceivable that parity invariants could be deduced, where references to a class of cells come in pairs so one need count only one of the twins. It may be possible to observe a class of cells for which references always increase at least as fast as they are destroyed with respect to each individual node. An example of this might be the double linked list header pointed to by the first and last sequence element.

There is no end to the kinds of invariants one might be able to speculate on. It would be interesting to see empirical work done to determine what is deducible in practice and how much the information is worth. A profoundly useful result would be frequently determinable criterion to capture the commonly used data structures under conditions of sublist release.

A side benefit of determining invariants is that when the ZCT is scanned against the variables on the run time stack, fewer variables need be looked at. Specifically, those variables known to point to invariant cells may be skipped. This suggests a generalization. A variable may be invariant in the property that it need not be scanned against the ZCT. Variables for which non invariant variables, or cell references always exist which point to the same cell are in this class.

Discussion

The previous section suggests optimizations that can be better realized through language design than by compile time deduction. Should it not be possible to declare a class of cells to be not collectable? This could be an attribute of selected pointer types. The programmer does not lost security in case of misuse, only fails to get effective storage reclamation. Since an infrequent classical garbage collection is recommended by Deutsch and Bobrow to reclaim circular structures, warnings could be obtained at this time that unexpected cells were inaccessible.

An extension to mixed automatic storage reclamation and explicit storage release in the same program is possible. A program declares a pointer type to be EXPLICIT and no reference counts need be kept for it. The downfall of this plan is that whatever method of release is supplied will destroy compiler guaranteed safety. If this is considered acceptable, and it probably should not be, then a cell DISPOSE as well as a cell and all its descendants DISPOSE should exist. The well disciplined data structures, like list or tree, can now conveniently be disposed of by section and at low run time overhead.

The next section proposes a far more drastic approach aimed at giving the programmer greater control of storage reclamation, with hopefully more local ability to check the correctness of his actions.

Extensible Storage Reclamation

The original goal of freeing the programmer from the tedium of heap storage management inspired automatic storage reclamation. The utility is bought at considerable expense. The ideas presented in this paper tend to move the associated costs to compile time. Most of the techniques can be exploited to far less than their true potential because obtaining information by statically inspecting a program is inherently difficult. What will be explored in this section is returning partial control back to the user who desires it, hopefully without reintroducing the tedium previously experienced.

There are two levels of control that can be distinguished. The programmer may be given the opportunity to generate transactions, and suppress defaults, or he may be allowed to amend the transaction interpreter. In either case, his knowledge of the program's behavior is essential, and the danger of mishap is real. It appears that the inconvenience of adding handwritten transaction code as an afterthought to a program is an order of complexity lower than being forced to manipulate storage from scratch. The programmer might do this only for the inner loop sections of code. Since economy may be gained because larger heterogeneous batches can be recognized by hand, it appears reasonable to allow extensions to the transaction interpreter. These extension programs would be written in a language consisting of other transactions and appropriate control structure. At a lower level, the programmer might be given direct access to the tables and the free list.

The extender might want to specify points in his source program at which the storage reclaimer is invoked, and other areas in which it is forbidden. In this way, he can use his knowledge of the program's behavior to avoid being caught in the reclaimer in sections that are temporarily embarassing for the shortcuts he has employed. Also, he can limit the number of known variables at reclamation time, economizing on the time needed to scan the variables against the ZCT.

Conclusion

Automatic storage reclamation, when viewed in the Deutsch-Bobrow transaction model, can benefit from compile time optimization. Some amount of this can be automatically done using global flow analysis. To achieve greater improvement, the programmer can be asked to supply information or instructions.

There are two major unanswered questions. It is uncertain how the transaction model would work in a language with the properties assumed here. Secondly, it requires substantial empirical study to determine whether conditions necessary for the transformations can be determined and whether this will result in much actual time saving.

Inspection of some sample programs seems to indicate that Allocation/Cancellation and detecting pointers that are NIL before assignment are the easiest transformations to deduce and the most worthwhile in terms of the number of occurrences found. Invariants are well understood by programmers and, being hard to deduce, should be user supplied optimizations. Batching is going to be most useful in the kind of system which allows the transactions to be hand supplied.

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. This kind of thinking is important for the future when run time problems will be more and more compile time oriented. What was once debugging has been superseded by the compile time function of verification. Dynamic program measurement will be feasible if it is partially preprocessed at compile time. Given that enough of these economies and services can be made available, it will pay to routinely design compiler systems that have built in machinery to handle the necessary bookkeeping.

Acknowledgments

I would like to thank Professor Susan Graham and Mark Wegman, both of the University of California, Berkeley, for their valuable assistance at different stages in the preparation of this paper.

APPENDIX

Proof of Correctness for Allocation/Cancellation Algorithm

Claim. $v \in \cap$EDGESINTON.DS implies for all computation paths to N, v points to the last allocated cell of PICKEDNEW.

Proof. Assume for contradiction that there are incorrect variables in the DS sets. A variable is incorrect at a node N if there exists a computation path from the entry node of the program to N on which the variable is incorrectly set to point to something else at N. Pick the minimal length such computation path:

Case 1: Path of zero length. Then N is the entry node and $\cap$EDGEINTON.DS $= \emptyset$ and claim holds vacuously. Contradiction.

Case 2: Path of non-trivial length. Then a node M immediately precedes N on the path. Since $v \in$ EDGESOUTOFM.DS.

Case 2.1: M is PICKEDNEW. Then $v \in$ EDGESOUTOFM.DS means v was the argument of PICKEDNEW. Contradiction.

Case 2.2: M is an assignment statement. If v was not LHS, then it could not have been altered. So, $v \in \cap$EDGESINTOM.DS. Since shortest path picked for which claim fails, v must point to last node allocated by PICKEDNEW. If v was LHS then RHS must have pointed to last allocated node by the same argument. Contradiction.

Case 2.3: In no other way could v have been altered, so again $v \in \cap$EDGESINTOM.DS implies by shortest path assumption that v is correctly valued. Contradiction. □

Claim. $v \notin \cup$EDGESINTON.MS implies there is no computation path to N that would allow v to point to the last allocated cell of PICKEDNEW.

Proof. The proof is like above. Pick the shortest path that exhibits an incorrect omission to an MS set. The critical argument is now rephrased to "since v is not in $\cup$EDGESINTON.MS, it is not in EDGESOUTOFM.MS for its predecessor M in the shortest path". Beyond this it is only to check that the program handles Case 1 correctly and substitutes union for intersection. □

Claim. ENTERTRANSACTIONS correctly enters transactions and code to put cells back on the free list.

Proof. A MARK of 2 travels down a flow path and says "this path is short a delayed ALLOCATE". An edge can be defaulted by changing the mark of 2 to the appropriate ALLOCATE transaction.

All edges with the MARK of 2 have nonempty DS sets.

An owed ALLOCATE flowing into PICKEDNEW can be defaulted.

An owed ALLOCATE flowing into a heap assignment can be canceled when RHS is known to point to the last cell allocated by PICKEDNEW.

Given a node where both MS and DS disappear, code to free storage can be generated. No more variables point to the cell since MS is empty and the fact that an ALLOCATE is owed implies that no heap pointer to it exists. Since on each edge MS $\subseteq$ DS, and EDGESOUTOFN.MS is empty, all of the incoming edges must have had MS = DS = {some single variable v}. This follows from the non-emptiness of each DS set on an edge marked 2. Moreover, v is removed from $\cup$EDGESINTONODE.MS by what must have been an assignment to v. Thus, using the value of LHS before the assignment as the cell to free after the assignment is executed is correct.

In the EDGESOUTOFN.DS = $\emptyset$ case, the all other cases code, and the last FOR loop, default transactions are generated. □

Proof of Termination and Time Bounds for Allocation/Cancellation Algorithm

Claim. PROCEDURE FINDSETDS terminates in $O(|N||E||V|)$ bit vector steps.

Proof. Each time every node is visited, some membership in the DS sets changes, except the terminating pass. All changes monotonically decrease the total set population for the total collection of the DS sets. The original total population is $|E||V|$. Visiting each node is $O(|N|)$ bit vector steps. □

Claim. PROCEDURE FINDSETMS terminates in $O(|N||E||V|)$ bit vector steps.

Proof. Same as above with the MS sets monotonically increasing and maximum population being $O(|E||V|)$. □

Claim. PROCEDURE ENTERTRANSACTIONS terminates in $O(|N|^2)$ steps.

Proof. The first FOR loop looks at each edge, but there are no more than $|N|^2$ edges in the flow graph.

Each execution of the REPEAT statement causes an increase in the number of 1 marks. No 1 mark is ever erased. This gives a crude upper bound of E on the number of executions of the REPEAT statement. Actually, a node appears on the QUEUE at most once, since to appear its entering edges must all be marked 2 and changed to a permanent 1 when processed. So, the body of the REPEAT is executed at most $|N|$ times. Each case of the REPEAT body is easily executed in time $O(|N|)$ except the last one. We have $|N|$ edges, each of which enter a node. Checking for these nodes that all entering edges are marked 2 can be done with a count associated with the node. Thus, the last case is also $O(|N|)$.

It follows that the entire procedure is $O(|N|^2)$. □

Theorem. The running time for the Allocation/Cancellation Algorithm is $O(|P||N||E||V|)$ where $|P|$ is the number of NEW statements in the program with variable argument, and is not less than 1.

Proof. The main loop looks at each node in the flow graph once, and calls the three subroutines $|P|$ times. The theorem follows from the claims above and the fact that $O(|N|^2)$ is dominated by $O(|N||E||V|)$ in the case of connected flow graphs. □

Theorem. The transformations generated by the Allocation/Cancellation Algorithm are safe.

Proof. The only way 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 happen on 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. □

Theorem. The transformations generated by the Allocation/Cancellation Algorithm are effective.

Proof. The only time that a cell can be collected is when no variable points to it. By insuring that DS is non-empty on every edge of the flow graph for which a transaction is owed, effectiveness is insured. □

REFERENCES

[CG] Clark, Douglas and Green, C. Cordell. An empirical study of list structure in LISP. Stanford University Technical Report (in preparation).

[DB] Deutsch, L. Peter and Bobrow, Daniel G. An efficient incremental automatic garbage collector. Xerox Palo Alto Research Park Report (January 1975).

[HU] Hecht, Matthew S. and Ullman, Jeffrey D. Analysis of a simple algorithm for global data flow problems. Proceedings ACM Symposium on Principles of Programming Languages (October 1973), 207-217.

[KU] Kam, John B. and Ullman, Jeffrey D. Global optimization problems and iterative algorithms. Princeton Computer Science Laboratory Technical Report 146 (January 1974).

[K] Knuth, Donald E. The Art of Computer Programming, Vol. 1: Fundamental Algorithms. Addison-Wesley Publishing Co., Reading, Mass. (1969).

[R] Rosen, Barry K. Data flow analysis for recursive PL/1 programs. IBM Research Report, Yorktown Heights, N.Y. (January 1975).

[WJ] Wirth, Niklaus and Jensen, Kathleen. PASCAL User Manual and Report. Springer Verlag, New York (1974).