A Variant of the Fisher-Morris Garbage Compaction Algorithm.

by

Johannes J. Martin

Department of Computer Science
Virginia Tech
Blacksburg, VA 24061

Technical Report No. CS80006-R

Keywords

Garbage collection, compaction, compactification, storage reclamation, storage allocation, relocation, list processing, pointers, references, data structures

CR Categories: 4.34, 4.49, 5.32

ABSTRACT: The garbage compactification algorithm described works in linear time and, for the most part, does not require any work space. It combines marking and compactification into a two-step algorithm. The first step marks all non-garbage cells and, at the same time, rearranges the pointers such that the cells can be moved, the second step performs the actual compatification.

1. Introduction

The purpose of a garbage compaction program can be described as follows. In a storage area divided into cells of possibly different sizes where some but usually not all are accessable by the user's program, a garbage compactor is to move the accessable cells to one end of the storage area and to update all pointers to these cells (pointers stored in cells as well as those that point to cells from the outside) such that they point to the new locations of the cells. At the other end of the storage area, this process consequently creates a block of adjacent, unused fields as large as the sum of the fields occupied by all inaccessable (garbage) cells. Algorithms for this task consist of two phases: The first phase identifies all accessable cells by marker bits; the second phase performs the actual compactification.

Ealier algorithms, reviewed and assessed in [Morris78], suffer from either storage or time inefficiencies avoided by the Fisher-Morris method. In essence, this method reversibly rearranges pointers to cells in such a way that the next field to be moved is not the target of of any pointer. Fisher's version of the algorithm is applicable only if all pointers run in the same direction whereas Morris' method (discovered independently in 1976) works for the general case. The Fisher-Morris algorithm starts with the cell space already marked. In addition to the usual marker bit, it requires one bit per pointer field for its own bookkeeping.

The variant described here combines the marking phase with the step that rearranges the pointers. It does not need the usual marker bit because, for marking, it utilizes the extra bits required in pointer fields, nor does it need other additional storage space provided that one of the following conditions is met.

1. All pointers run in the same direction (say 'down').

2. Pointers pointing up refer to fields that do not themselves contain pointers.

3. Fields accessed by up-pointers do contain pointers but they are also accessed by down-pointers and, during the marking phase, reached by down-pointers first.

Only if an up-pointer points to a field that also contains a pointer and if, during marking, this field is not reached by a down-pointer before it is found through an up-pointer, then an additional pointer field outside of the cell space is needed for keeping track of the original structure. If all cells allowed to be accessed by up-pointers have a header field (e.g. containing information about the cell type but no pointers), this case will not occur, and no additional space is ever needed.

In its present form, the method is limited to structures where all pointers that access the same cell point to the same field of the cell.

For the most part, the marking phase, although conceptually recursive, does not need stack space; it rather uses the rearranged pointers to find its way through the cell structure. In this respect, the method is related to the Schorr-Waite marking algorithm [Schorr67]. Therefore, rearranging the pointers can be considered free since costs can be charged to the marking phase as overhead needed to avoid stacking.

Since the method viewed as a whole is rather complex, its description is broken down into a list of simpler assertions which are proved one by one.

2.0 Theory of the method

The cell space is a finite ordered set of fields. Fields may contain pointers to other fields or data. Pointer fields are distinguishable from data fields; pointers can be flagged. Data items can be flagged if they are stored in certain types of fields (described below). Since the set of fields is ordered, an up-pointer (a pointer stored in a field with a lower ordinal than the field pointed to) can be destinguished from a down-pointer. A particular instance of the cell space where every field is given a value is called a configuration.

In a given configuration, three kinds of fields can be distinguished by their properties:

1. Leaf fields are not reached by pointers; they are, though, otherwise known to the user's program.

2. Node fields are reached by pointers such that they are accessable by

the user's program. Data items stored in node fields may be flagged.

3. Garbage fields are not accessable by the user's program.

Not all configurations can actually occur in the course of an orderly use of an allocation scheme. In a possible configuration, the cell space is partitioned into cells. Cells are sets of (generally adjacent) fields; one field of each cell is a node field all others are leaves.

Without loss of generality, the following assumptions are made:

1. In a cell, the node field is the field with the lowest ordinal,

2. the size and the structure of a cell are known if the pointer to its node field is known,

3. cells are to be compacted toward the lower end of the cell space.

Note: The method described can also be applied if a computer word contains more than one pointer field. In this case, the pointer stored in the k-th field of a word x referencing word y, is assumed to reference the k-th field in y. This strategy is due to [Fisher74].

The process of compactification consists of two phases. The first one, that is the marking phase, flags all accessable node fields and changes pointers to prepare the structure for the second phase, namely the actual compactification.

Four functions will be used in the discussion below; these involve the following domains.

1. S : the set of all configurations of the cell space,
2. POINTER: the set of all valid pointers into the cell space,
3. DATA: things that fit into fields and are distinguishable from pointers,
4. {true,false}.

Note: In reality, the cell space S would, of course, not occur as an explicit parameter; here, however, it simplifies the formal description.

The functions are

```
CONT:  POINTER x S                    ->  (POINTER U DATA)
                   /* contents of field pointed to */

SET :  POINTER x (POINTER U DATA) x S  -> S
                   /* alter contents of field      */

FLAG:  POINTER x S                    -> {true,false}
                   /* is contents flagged?         */

STFLG: POINTER x {true,false} x S  -> S
                   /* set flag in field            */
```

These functions are partially defined by the following algebraic axioms [Guttag75 or Horowitz76].

For all P,Q in POINTER, R in (POINTER U DATA)

and C in S let:

```
CONT(P, SET(Q,R,C))   = if (P=Q) then R
                                 else CONT(P,C)

FLAG(P, STFLG(Q,B,C)) = if (P=Q) then B
                                 else FLAG(P,C)
```

(These axioms are not complete since they do not provide a starting point for the generation of the elements of S; they are, however, sufficient for our purpose).

With these functions, the operation REV is implemented. REV(P,C) reverses the pointer stored in the field P and inverts its flag (see figure 2.1). Thus, suppose the field P contains a pointer to Q whith a flag value of F and Q contains some value x, then, after REV(P,C), P will contain x and Q will contain a pointer to P with the flag value of ~F.

```
P        Q           P        Q

o-- F--->o   ==>     o<--- ~F--o
         |           |
         x           x
```

FIGURE 2.1

Formally,

```
REV:  POINTER x S   ->   S

and

 For all P,Q in POINTER and C in S

CONT(Q, REV(P,C)) = if IS_POINTER(CONT(P,C))
                    then case
                          Q=P        : CONT(CONT(P,C),C);
                          Q=CONT(P,C): P;
                          otherwise  : CONT(Q,C);
                          end case;
                    else error;

FLAG(Q, REV(P,C)) = if IS_POINTER(CONT(P,C))
                    then case
                          Q=P        : FLAG(CONT(P,C),C);
                          Q=CONT(P,C): ~FLAG(P,C);
                          otherwise  : FLAG(Q,C);
                          end case;
                    else error;
```

From this definition of REV, it can readily be verified that the effect of REV(P,C) is reversed by applying REV a second time but to CONT(P,C), the other field involved. Thus

$$REV(CONT(P,C), REV(P,C)) = C.$$

For example, suppose the field P points to the node Q. Then the state of the cell space

$C1 = REV(P,C)$ can be transformed back to C by computing

$C = REV(Q,C1)$.

Of course, the contents of both P and Q must not be changed between the two applications of REV.

This property of reversibility is the basis for the compactifying algorithm described here.

For the following discussion, it is convenient to derive two subproperties from the property of reversibility:

Definition:

Let P be the pointer to some field F and $Q = CONT(P,C)$ a pointer to a field G than the pointer from G to F created by

$$C' = REV(P,C)$$

is called 'tip reversible' in C'' iff

$$CONT(P,C'') = CONT(P,C'),$$

it is called 'tail reversible' in C'' iff

CONT(Q,C'') = P and FLAG(Q,C'') = FLAG(Q,C').

Obviously, a pointer that is both tip and tail reversible is reversible.

2.1 Cell space with down-pointers only

2.11 The marking phase.

At the beginning of the marking phase, by postulate, the cell space does not contain any flagged pointers.

Marking is now a very simple process. It consists of applying REV, in any order, to all those leaves that contain pointers not yet flagged. Thus, the marking program has the form

```
WHILE there is a leaf P with a pointer not yet flagged
  DO  C := REV(P,C) END;
```

Lemma 1: the above process will flag all nodes accessable from leaves.

Proof: (induction over the depth of the pointer structure)

The depth of the pointer structure in the cell space is equal to the sum of the lengths of all chains of unflagged pointers that starts at a leaf. Since there are only down-pointers, loops do not exist.

If the depth is zero, then no pointers exist at all (except possibly in garbage cells). Thus no nodes exist, nothing needs to be flagged. Obviously, the method works in this case.

Suppose the method works for structures of depth up to n. Then given a structure of depth n+1, the application of REV to a leaf that contains the first pointer of a chain of length k puts into this leaf the contents of the node pointed to, which is the first pointer to a chain to length k-1 or, if k = 1, the null pointer or a data item. Thus, the depth of the structure is now not more than n. A flagged pointer is stored in the node field hence the node field is flagged in the process. All nodes previously accessable from leaves still are except possibly for the one just flagged. q.e.d.

In contrast to the algorithm above, any actual marking program must commit itself to some order for processing the leaves; however, since the correctness of such a program does not depend on the order, the order can be chosen to suit convenience.

Lemma 2: Flagged pointers stored in node fields are tail reversible.

Proof: Marking applies REV only to leaves; thus, node fields are always involved as 'the other field', which receives a freshly flagged pointer back to the leaf to which REV is applied. The last such application involving a node N determines the pointer stored in N. q.e.d.

The marked cell space has some other properties that are important for the compactification process. In order to understand them, the marking algorithm must be studied in more depth. To this end, the marking program is modified to stop after the first n applications of REV:

```
  FOR  i:=1 to n
 WHILE there is a leaf P with a pointer not yet flagged
   DO  C := REV(P,C) END;
```

The configuration of the cell space after the execution of this program is called Cn. Since the order in which the leaves are processed by the marking algorithm is not determined, there are many configurations Cn. The set of all Cn is called CCn.

Lemma 3: All Cn have exactly n flagged pointers.

Proof: By postulate, C0 has no flagged pointers; each application of REV adds exactly one flagged pointer (in the marking program, the flag inverted by REV is, in fact, always set - never cleared - since REV is applied only to leaves that contain non flagged pointers).

q.e.d.

Lemma 4: For i not equal j, the intersection of CCi and CCj is empty.

Proof: trivial because of lemma 3.

Lemma 5: In Cn, let N be the lowest node that contains a flagged pointer, then

REV(N,Cn) is in CC(n-1)

Proof: It must be demonstrated that the flagged pointer stored in N is, indeed, reversible.

The last application of REV that involved the node N clearly was of the form

C := REV(P,C(n-i)) with CONT(P,C(n-i)) = N

that is, P is a leaf that, in C(n-i), contains a non-flagged pointer to N.

Three cases can be distinguished:

1. CONT(N,C(n-i)) = D is a data item; then CONT(P,C(n-i+1)) = D and, hence, P will not be processed any further by the marking program (recall that marking processes only leaves that contain pointers). Therefore, the flagged pointer stored in N (it points to P!) is tip reversible.

2. CONT(N,C(n-i)) = Q is a flagged pointer;

then an argument similar to the one under 1. applies.

3. CONT(N,C(n-i)) = Q is a pointer not yet flagged;

then, by the premise that all pointer in C0 are down-pointers, Q points to a node lower than N. Now, since N is by definition the lowest node flagged, P has not been processed by REV after it has received its contents (otherwise the node lower than N at the end of pointer Q would have been flagged). Thus, the flagged pointer in N is again tip reversible.

Therefore, the pointer stored in N is always tip reversible and, by lemma 2, it is tail reversible (since N is a node field), thus, it is reversible. q.e.d.

Theorem 1: By applying REV to each flagged node repeatedly until it is not flagged anymore, starting with the lowest node and proceeding in ascending order, any configuration Cn is transformed back into configuration C0.

Note: This theorem is of crucial importance since it ensures that the compactifying process, which starts at the low end of the cell space, will properly restore all pointers.

Proof: (induction over n)

C0 does not contain any flagged nodes, so the base case (n=0) is trivially established.

Lemma 5 provides the induction step. Lemma 4 ensures well ordering. q.e.d.

The following lemmas ensure that the lowest flagged node is not reached by any pointer and, hence, can freely be moved.

Lemma 6: No flagged pointer in Cn points to a node field.

Proof: The marking program applies REV(P,C) only to leaves P. The only flagged pointer introduced by REV(P,C) is the flagged pointer to (the leaf) P. Since C0 does not contain any flagged pointers, all flagged pointers in Cn point to leaves. q.e.d.

Definition: a node is fully processed if it is not reached by any pointer not yet flagged.

Lemma 7: A fully processed node is not reached by any pointer.

Proof: by lemma 6. q.e.d.

Lemma 8: If there is a leaf in Cn that is reached by a flagged pointer, then there is a node in Cn that contains a flagged pointer and that has a lower ordinal than the said leaf.

Proof: Although the pointer reaching the leaf may reside in another leaf rather than in a node, this (flagged) pointer was created by REV and placed in a node from where it may have been subsequently copied to a leaf. Since the original pointer from the leaf to the node was a down-pointer, the ordinal of the node must be lower than that of the leaf. q.e.d.

Lemma 9: Leaves not reached by flagged pointers contain their original information.

Proof: The contents of a leaf P is changed only if REV is applied to it. When that happens, REV always places a flagged pointer to P into some node. Although this pointer may be moved elsewhere by subsequent applications of REV leading to configurations Cm with a higher index m, the value of the pointer is not altered. Thus, a leaf that has been processed at all is always reached by a flagged pointer. q.e.d.

2.12 Compactification.

The compactification phase can now be described. Suppose Cn is a totally flagged configuration and N is the lowest flagged node. This is the first field that must be shifted to its final destination. A field can obviously be relocated by simply copying its contents to the new location if it is not reached by any pointer. By lemma 7, N is not reached by any pointer thus it can be moved, and, by lemma 5, REV(N,Cn) is in CC(n-1). Therefore, the compactification program prodeeds as follows: First, it copies the node to its new place and then it applies REV(N,C) repeatedly until FLAG(N,C) is false. Now all leaves that belong to N's cell can be moved since no pointers refer to them, for, by lemma 8, they are not reached by flagged pointers and, by definition (they are leaves!), they are not reached by non-flagged pointers. By lemma 9, they contain their original information.

After the leaves are moved the next cell to be moved is headed by the next higher flagged node. This is now the lowest flagged field in the cell space, and the above argument applies again.

Theorem 2: The compactification program described above properly compactifies and restores the cells.

Proof: The reasons given above can be strengthened into an inductive proof. q.e.d.

Note: Theorem 2 and its proof have not been given a stronger form because the shift operation has not been formalized. Adding this would not lead to new insight but burden the discussion with excessive detail.

2.2 Cell space with both up and down-pointers

For this next step of the discussion, it is still required that a node is reached by at least one down-pointer. The final marking program will ensure that this is the case by introducing temporary leaves where necessary. It is also required that an up-pointer can be recognized as such even after it has been copied into a field of a higher ordinal where it may appear to be a down-pointer. That this requirement can easily be satisfied will be seen in the description of the actual implementation. With these conditions satisfied, marking proceeds as before by simply treating up-pointers as data. As a result, all lemmas as well as theorem 1 are still true, but, nevertheless, compactification can not proceed as before since marking does not fully process nodes that are reached by up-pointers. Hence, these nodes can not yet be moved.

However in the totally marked cell space, compactification can move and restore the lowest node as before since the lowest node is, by definition, not reached by up-pointers. Then, the program continues by checking the contents restored whether it is an up-pointer. If so, then it reverses the pointer by REV. Since the lowest node is not changed any further, the flagged pointer created by REV is tip reversible. Similarly, the leaves belonging to the lowest cell are checked for up-pointers and up-pointers found are reversed. Also here, the flagged pointers created are tip reversible. Compactification now proceeds to next higher node that is flagged. Since all up-pointers from below have been reversed this node is fully processed and can be moved; Its contents can be restored since lemma 2 (tail reversibility) still applies (the proof of lemma 2 can easily be generalized to include the use of REV in the compaction phase). Infact, whenever a node becomes the lowest flagged node in the not yet compactified part of the cell space, it will be fully processed because up-pointers to it can originate only in the area already compressed and those up-pointers have been reversed by the compactification phase.

Thus, compactification modified in the way described will properly compress and restore the cell space.

3.0 Implementation

The order in which leaves can be processed is partially dictated by necessity since leaves that belong to cells can not be reached before the node field of the cell is reached. The only leaves immediately known to the garbage collector are the pointer variables outside of the cell space.

> Note: Fields outside the actual cell space can be considered to be of higher ordinals than all other fields by equating the attribute 'outside' with the attribute 'greater'. Thus, all pointers originally stored in these fields are considered down-pointers.

With the function C' = MARKLEAF(P,C) that creates a new configuration C' from C by marking all nodes directly and indirectly accessable from the leaf P, the marking program has the following form.

```
procedure  MARKALL(P,C);
  begin for_all outside leaves P
        do C := MARKLEAF(P,C)
  end;
```

Concepually, MARKLEAF is a recursive program. If applied to a leaf P that points to a cell Q, MARKLEAF acts on the pointer to the node field of Q (that is, it applies REV to a down-pointer but not to an up-pointer) and then applies itself in turn to all leaves of the cell Q.

Figure 3.1 shows this recursive program. Note that the ordinal (that is, the pointer) of the target Q is compared with TEMP, the original home of the pointer to Q, in order to determine whether the pointer's original direction is up or down. Also note that a new leaf is introduced if a node field is reached by an up-pointer. However, it turns out that this is only necessary if the node referenced contains a pointer. If it contains data then it suffices simply to flag it and process the rest of the cell as if the cell were reached by a down-pointer.

```
procedure  MARKLEAF(P,C);                {recursive version}
  begin TEMP := P; Q := CONT(P,C);
       while IS_POINTER(Q) & ~FLAG(P,C) & (Q<TEMP)
         begin {down-pointer}
           C := REV(P,C);
           if ~FLAG(P,C)
             then begin
                    for_all leaves L of cell Q
                       do C := MARKLEAF(L,C);
                    TEMP := Q; Q := CONT(P,C)
                  end
         end {while};
      if     IS_POINTER(Q) & ~FLAG(P,C) & (Q>TEMP & ~FLAG(Q,C))
             {TEMP=Q is an unrelated special case
              and, hence, not treated here        }

         then  {up-pointer}
           if IS_POINTER(CONT(Q,C))
             then begin
                    N := NEWLEAF; C := SET(N,Q,C);
                    C := MARKLEAF(N,C)
                  end
             else begin
                    C := STFLG(Q,true,C);
                    for_all leaves L of cell Q
                      do C := MARKLEAF(L,C)
                  end
        return
    end {MARKLEAF};
```

FIGURE 3.1

Most of the work space needed by this recursive routine can be saved. In most cases, the leaf at which marking starts can be recovered after a cell is processed by following back up the flagged pointer that REV, applied to the leaf, puts into the node field of the cell. For this purpose, REV can even be applied to up-pointers, if they point to nodes which contain data (not pointers). However, since the method does not call for (and can not bear) the reversal of an up-pointer during the marking phase, a second application of REV is necessary after the retreat is accomplished. This second application of REV does not necessarily undo the first one, though.

However, the second REV is itself undone by the REV applied to the up-pointer during the compaction phase; consequently, each REV has its reversing counterpart. Only in those cases where a new leaf must be created, will it indeed be necessary to save the pointer to the leaf from which the cell was reached, on a stack.

Figure 3.2 shows the non-recursive version of the MARKLEAF program. The boolean function IS_NEWLEAF(field) has the obvious meaning, the function CHASE(field) follows a chain of flagged pointers until it finds a field that contains a non flagged pointer or data; CHASE then returns the pointer to this field. It is CHASE that finds the way back to the leaf from which marking was initiated and, thus, saves the stacking.

```
procedure  MARKLEAF(P,C);                {non-recursive version}
  begin
    1: TEMP := P; Q := CONT(P,C);
       while IS_POINTER(Q) & ~FLAG(P,C) & (Q<TEMP v ~FLAG(Q,C))
         do begin
              if Q>TEMP
                then {up-pointer}
                  if IS_POINTER(CONT(Q,C))
                    then begin
                           save P on stack;
                           P := NEWLEAF; C := SET(P,Q,C);
                         end
                    else begin
                           C := STFLG(Q,true,C);
                         end;
              C := REV(P,C);
              if ~FLAG(P,C)
                then begin
                       for_all leaves P of cell Q
                         do begin
                              goto 1    {C := MARKLEAF(P,C) };
                           2:  {return from pseudo recursion}
                            end;
                       compute pointer Q to cell from address
                       P of last leaf processed (see note);
                       TEMP := Q; P := CHASE(Q); Q := CONT(P,C)
           end {while};
       if IS_NEWLEAF(P)
         then pop P off stack {up-pointer to node with pointer};
       if FLAG(P,C) & ~IS_POINTER(Q)
         then C := REV(P,C̄)    {up-pointer to node with data};
       if P is not an outside reference
         then goto 2        {return to pseudo recursive call};
         else return
  end {MARKLEAF};
```

Note: The loop 'for_all leaves P of cell Q' can be implemented in different ways. One method takes advantage of the fact that, when the cell is entered for marking, only its node field is flagged. Therefore, the address of the highest leaf is computed (this can be done since, by assumption 2, the structure of the cell is known), and then the leaves are processed in decending order. Since data are destinguishable from pointers, leaves containing data are simply skipped. Eventually, by proceeding to fields with lower ordinals, the node field of the cell will be encountered and recognized because it is flagged. The address of the node field is, of course, the pointer of the cell Q.

FIGURE 3.2

The compaction program itself is strictly iterative and can be coded in a straight forward way from the description given above.

References

Fisher74 Fisher,D.A. "Bounded workspace garbage collection in an address-order preserving list processing environment," Information Processing Letters 3,1 (July 74) 29-32

Guttag75 Guttag,J.V. "The specification and application to programming of abstract data types," Ph.D.Thesis, University of Toronto, Department of Computer Science, 1975.

Horowitz76 Horowitz,E. and Sahni,S. Fundamentals of data structures, Computer Science Press, Inc. 1976

Morris78 Morris,F.L. "A time- and space-efficient garbage compaction algorithm," CACM 21,8 (August 78) 662-665

Schorr67 Schorr,H. and Waite,W.M. "An efficient machine-independent procedure for garbage collection in various list structures," CACM 10,8 (August 67) 501-506