stichting
mathematisch
centrum

AFDELING INFORMATICA (DEPARTMENT OF COMPUTER SCIENCE)

IW 107/79 MAART

H.B.M. JONKERS

A FAST GARBAGE-COMPACTION ALGORITHM

Preprint

2e boerhaavestraat 49 amsterdam

BIBLIOTHEEK MATHEMATISCH CENTRUM
AMSTERDAM

Printed at the Mathematical Centre, 49, 2e Boerhaavestraat, Amsterdam.

The Mathematical Centre, founded the 11-th of February 1946, is a non-profit institution aiming at the promotion of pure mathematics and its applications. It is sponsored by the Netherlands Government through the Netherlands Organization for the Advancement of Pure Research (Z.W.O).

AMS(MOS) subject classification scheme (1970): 68A10

ACM-Computing Reviews-categories: 4.34, 4.49, 5.32

A fast garbage-compaction algorithm *)

by

H.B.M. Jonkers

ABSTRACT

A compaction algorithm for variable size storage elements, for use in a compacting garbage collector is described. Under certain plausible assumptions on the structure of these storage elements, the algorithm requires no space overhead and is faster than any compaction algorithm published before. It scans the area to be compacted twice, but examines pointers only once.

KEY WORDS & PHRASES: garbage collection, compaction, list processing.

*) This report will be submitted for publication elsewhere.

0. INTRODUCTION

This paper presents a compaction algorithm for variable size storage elements, for use in a compacting garbage collector. Under certain plausible assumptions on the structure of these storage elements, the algorithm requires no space overhead and is faster than any compaction algorithm published before. It scans the area to be compacted twice, but examines pointers only once. We start by defining the problem in Section 1. Then, in Section 2 the algorithm is described and illustrated. In Section 3 the correctness of the algorithm is considered. Finally, in Section 4 the efficiency of the algorithm is discussed as compared with other compaction algorithms.

1. PROBLEM

We have a machine memory, which is represented by an array M of cells. Every cell has an address, which is the index of the cell in M. A pointer is an address or nil, which is a special value not equal to any address. We assume that every cell can contain a pointer. There is a subarray S of M, called the store, which is the part of M to be compacted. S contains a number of disjoint subarrays of (possibly) varying sizes, which we call nodes. The address of a node is the address of its first cell. Every node contains a number of pointer cells, which are cells that contain a pointer. We assume a pointer q in a pointer cell of a node is either nil or points to a node, i.e. q is either nil or the address of a node. Furthermore we assume that every node has at least one cell that does not contain a pointer, but some other value which is distinguishable from a pointer. For the sake of convenience we take that cell to be the first cell of a node.

Just before the call of the compaction routine we have the following information (established by the marking phase of the garbage collector):

(1) The addresses $e_1, \dots, e_l$ of cells not contained in S, containing nil or a pointer to a node.
These external cells are the starting points of the marking phase.
(2) The addresses $k_1, \dots, k_m$ of the nodes contained in S.
These nodes have been determined by the marking phase as being non-garbage.

This situation is schematically shown in Figure 1. The problem now is to rearrange the nodes in S in such a way that

(1) The nodes occupy a compact piece of memory at the left (= lower) end of S;
(2) Pointers in the pointer cells of nodes still point to the same nodes as before;
(3) Their order in S is preserved.

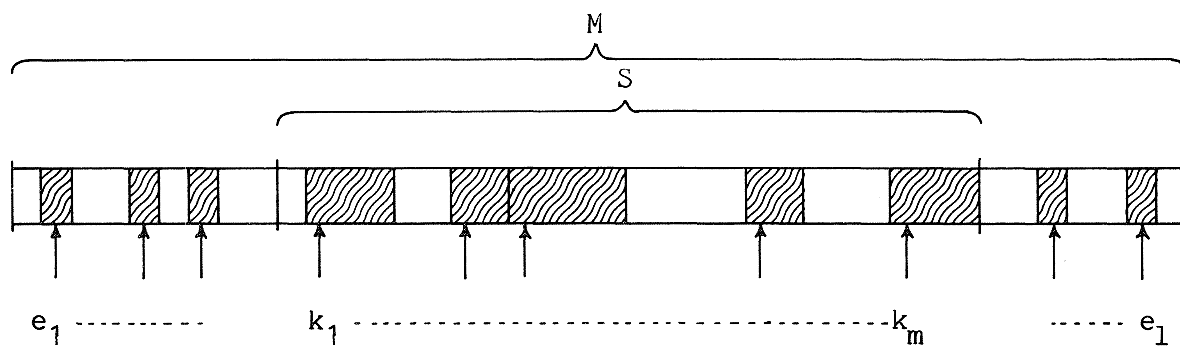

Figure 1

2. ALGORITHM

We shall describe the algorithm informally first and illustrate it by an example simultaneously (Figure 2.a; only the pointers to an arbitrary node v are shown). The algorithm starts by visiting all external cells. Upon visiting the external cell e containing a pointer to a node v, e cannot be updated immediately since the new address of v is not known yet. Therefore e is threaded to v. How this threading is achieved is not important yet, we will discuss that later. Having visited all external cells this way (Figure 2.b where <--- indicates the "is threaded to" relation), the algorithm then scans the store twice, visiting all nodes (hence all pointer cells) in the order from left to right. Upon visiting a node v in the first scan (Figure 2.c), the new address of v is known (from an accumulated counter). Using this information all cells threaded to v are updated (Figure 2.d), where "updating" a cell also means "unthreading" it. Subsequently all pointer cells of v containing a pointer to a node are threaded to that node (Figure 2.e). After the first scan all external cells will have been updated this way. Furthermore, all pointer cells at the left of a node v originally containing a pointer to v will have been updated, and all pointer cells in or at the right of v containing a pointer to v will be threaded to v (Figure 2.f). In the second scan all nodes are visited in the order from left to right again. Upon visiting a node v (Figure 2.g), all cells threaded to v are in or at the right of v, and neither v nor any node at the right of v will have been moved yet. All cells threaded to v can therefore correctly be updated to the new address of v (Figure 2.h). Now all cells originally containing a pointer to v will have been updated. Since all nodes at the left of v are already moved, v can safely be moved to its new address (Figure 2.i). At the end of the second scan all nodes will have been moved and all pointer cells - internal and external - will have been updated (Figure 2.j). The whole procedure is summarized below.

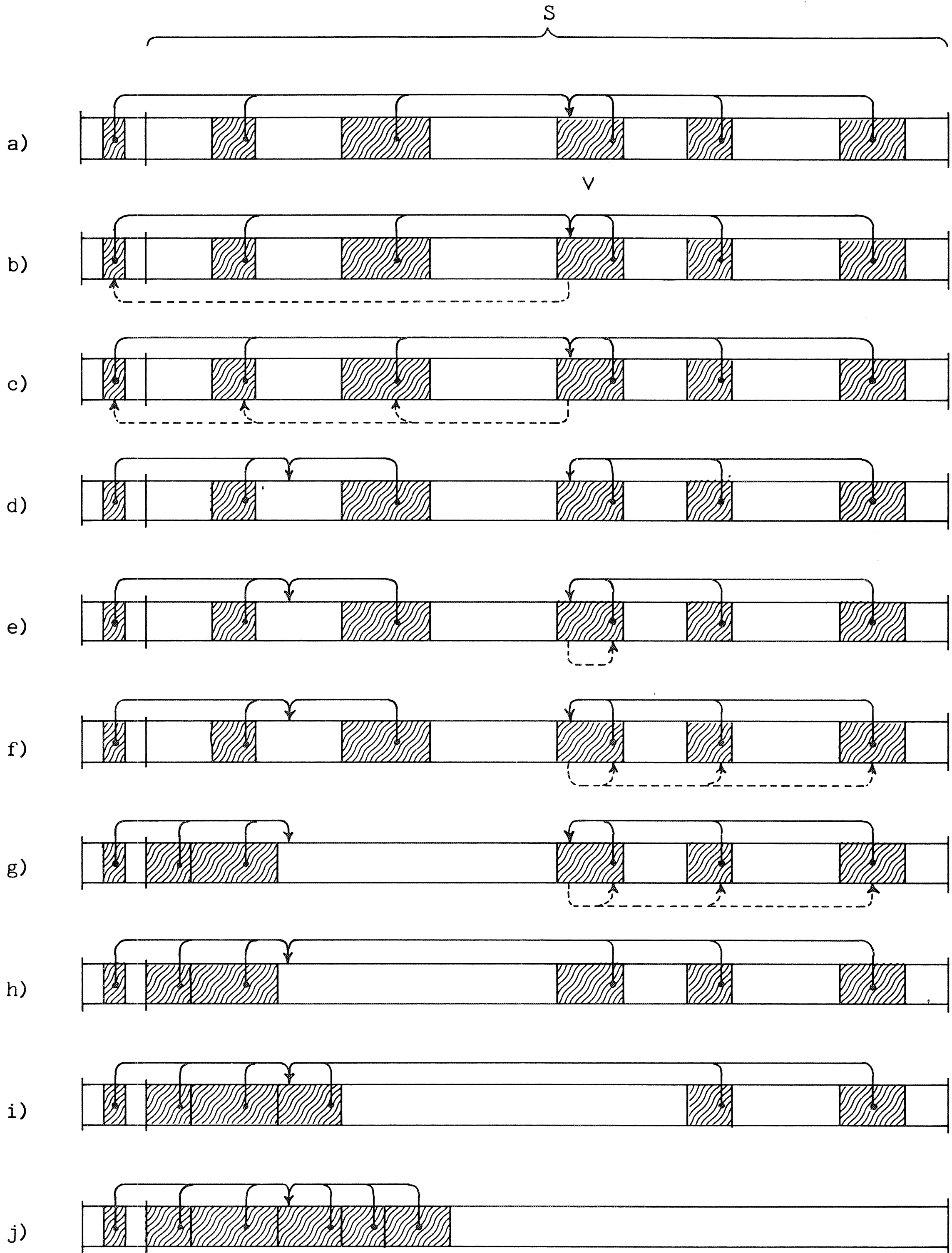

Figure 2

Algorithm 1 (informal description)

```
For every external cell e do
|  If e contains a pointer to a node v,
|  then thread e to v.

For every node v from left to right do
|  Calculate new address of v.
|  Update all cells threaded to v.
|  For every pointer cell p of v do
|  |  If p contains a pointer to a node w,
|  |  then thread p to w.

For every node v from left to right do
|  Calculate new address of v.
|  Update all cells threaded to v.
|  Move v to its new address.
```

We have not discussed yet how cells are "threaded" to a node. This threading of cells to a node can be done without space overhead using a well-known trick (also used in [1, 4, 6, 8]). We know two things:

(1) The first cell of a node does not contain a pointer.
(2) All cells threaded to a node (originally) contain a pointer to that node.

This situation is shown in Figure 3.a. Without loss of information we can transform this situation so that the cells threaded to a node are chained together in a list, using the first cell of the node as a list head and the original value of this cell as a list terminator, as in Figure 3.b. Updating all cells threaded to a node now simply is a matter of traversing this list, updating the cells in it and assigning the list terminator to the first cell of the node. The old value of the first cell of the node is thereby restored. This is important since this cell may contain information which is subsequently used to find the pointer cells of the

node (in the first scan) or to move the node (in the second scan). Figures 2.a through 2.j now have to be modified as in Figure 4.

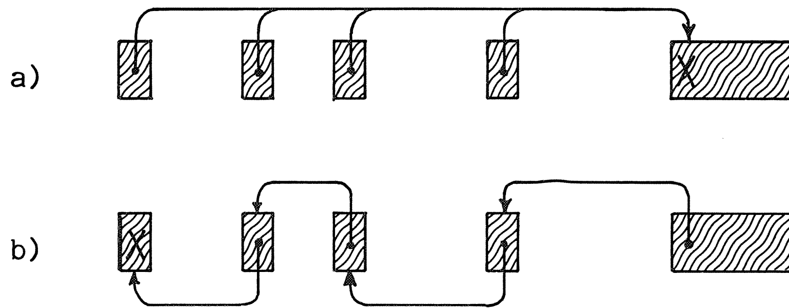

Figure 3

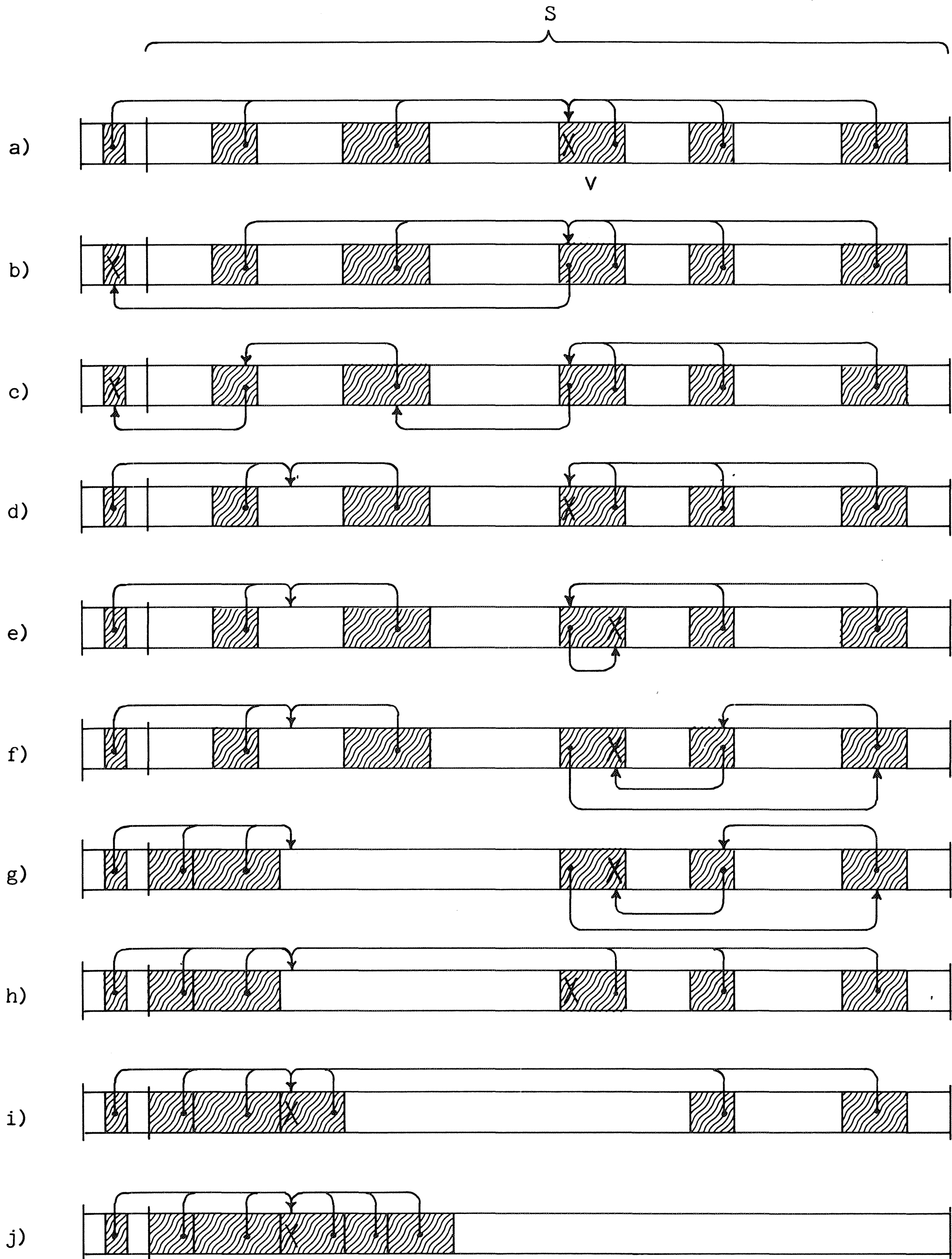

Figure 4

We are now able to describe the algorithm formally. For that purpose we use a kind of pseudo ALGOL 68, which semantics is (hopefully) obvious.

Algorithm 1 (formal description)

```
proc compact = void :
begin mode address = int;
   address new;

   for i to l do thread(e_i) od;

   new := lwb S;
   for i to m
   do update(k_i, new); scan(k_i); new := new + size(k_i) od;

   new := lwb S;
   for i to m
   do update(k_i, new); move(k_i, new); new := new + size(k_i) od;

   proc thread = (address p) void :
   if M[p] ≠ nil
      then address k = M[p]; M[p] := M[k]; M[k] := p
   fi;

   proc scan = (address k) void :
   begin let p_1, ... , p_n be the addresses of the
      pointer cells of the node with address k;
      for j to n do thread(p_j) od
   end;
```

```
proc update = (address old, new) void :
begin int p := M[old];
   while address(p)
   do int q = M[p]; M[p] := new; p := q od;
   M[old] := p
end;

proc move = (address old, new) void :
begin address p := new, q:= old;
   to size(old)
   do M[p] := M[q]; p := p + 1; q := q + 1 od
end

end
```

Remarks:

(1) "size(k)" is the size of the node with address k.

(2) "address(p)" yields true iff p is an address.

(3) By adjusting the operation "new := new + size(k_i)" the algorithm can also be used for paged memories.

3. CORRECTNESS

We shall not prove the correctness of Algorithm 1 formally here. Instead we shall give the principal invariants of the two main loops of the algorithm. They give the necessary hints to construct a proof of correctness. For that purpose we describe Algorithm 1 informally, using while loops for the two main loops (v_i is the node with address k_i):

```
For every external cell e do
|  If e contains a pointer to a node v,
|  then thread e to v.
i := 1;
While i ≤ m do
|  Calculate new address of v_i.
|  Update all cells threaded to v_i.
|  For every pointer cell p of v_i do
|  |  If p contains a pointer to a node w,
|  |  then thread p to w.
|  i := i + 1.
i := 1;
While i ≤ m do
|  Calculate new address of v_i.
|  Update all cells threaded to v_i.
|  Move v_i to its new address.
|  i := i + 1.
```

We shall call a cell of a node or an external cell

intact : if it contains the same value as before compaction;
threaded: if it is threaded to a node;
updated : if it contains the value it should contain after compaction.

The following invariants apply to the first while loop:

(1) Every external cell is threaded or updated.

(2) For every $j = 1, \ldots, m$

v_j is not moved.

If $j < i$ then

Every pointer cell of v_j is threaded or updated.

Every cell threaded to v_j is a pointer cell of a node v_k with $k \geq j$.

If $j \geq i$ then

Every pointer cell of v_j is intact.

Every cell threaded to v_j is an external cell or a pointer cell of a node v_k with $k < i$.

The following invariants apply to the second while loop:

(1) Every external cell is updated.

(2) For every $j = 1, \ldots, m$

If $j < i$ then

v_j is moved.

Every pointer cell of v_j is updated.

No cells are threaded to v_j.

If $j \geq i$ then

v_j is not moved.

Every pointer cell of v_j is threaded or updated.

Every cell threaded to v_j is a pointer cell of a node v_k with $k \geq j$.

4. EFFICIENCY

The algorithm presented here operates in linear time. This can easily be seen, because (in terms of the informal description) the algorithm visits every node twice, and executes at most one test, thread and update operation per external and pointer cell and exactly one move operation per node. Moreover, under the assumption that we have at least one cell per node, not containing a pointer, the algorithm requires no space overhead. The latter assumption is not unrealistic. In the implementation of many programming languages, such as SNOBOL or ALGOL 68, we are dealing with nodes of varying types and sizes, requiring the introduction of a type and/or size field in every node. Generally the contents of this type or size field can easily be distinguished from a pointer, thereby satisfying the assumption.

Several other solutions to the compaction problem dealt with in this paper have been published [3, 5, 9, 2, 6]. Comparing them to the solution presented here, we see that they either do not operate in linear time [3, 9, 2], or require a substantial space overhead [5]. The only exception is the algorithm presented in [6]. Somewhat modified (but not essentially) we can describe this algorithm informally as below:

Algorithm 2 (informal description)

```
For every external cell e do
|  If e contains a pointer to a node v,
|  then thread e to v.

For every node v from right to left do
|  Calculate new address of v.
|  Update all cells threaded to v.
|  For every pointer cell p of v do
|  |  If p contains a pointer to a node w,
|  |  then
|  |  |  If w is at the left of v,
|  |  |  then thread p to w,
|  |  |  else if w = v then update p.

For every node v from left to right do
|  Calculate new address of v.
|  Update all cells threaded to v.
|  Move v to its new address.
|  For every pointer cell p of v do
|  |  If p contains a pointer to a node w,
|  |  then
|  |  |  If w is at the right of v,
|  |  |  then thread p to w.
```

As compared with Algorithm 1, this algorithm has the following pros and cons:

Cons:

(1) We have to scan the pointer cells of a node twice instead of once. If finding the pointer cells of a node is a non-trivial operation, this can cause a substantial overhead.

(2) Visiting a pointer cell containing a pointer to a node, we have to execute an extra test to determine the direction the pointer is pointing in.

(3) The two scans of the store go in opposite directions. Since visiting the nodes going in one direction can be considerably more difficult than going in the other direction, this can also give an overhead.

(4) We have to know the amount of garbage beforehand.

Pros:

(1) Algorithm 2 is also applicable if we do not have a cell in every node, not containing a pointer, provided we have a mark bit in each cell. (The following invariant applies to Algorithm 2: if one or more cells are threaded to a node v, none of the pointer cells of v is threaded to a node. So we can use an arbitrary cell of v as the head of the list of cells threaded to v, even if this cell contains a pointer, and use the mark bit of a cell as the list terminator. Algorithm 1 does not satisfy the invariant.)

Conclusion: under the given assumptions Algorithm 1 is the most efficient compaction algorithm known so far; if not every node has a cell, not containing a pointer, Algorithm 2 is. This conclusion should not be misinterpreted, however. It does not mean that, if possible, Algorithm 1 should be used in a compacting garbage collector. A compacting garbage collector (conceptually) consists of a marking algorithm and a compaction algorithm. If every garbage collection should involve a compaction, it is possible to let the marking algorithm do the threading of external and pointer cells to their target nodes (the marking algorithm visits these

cells anyway). Then, in two separate scans the compaction algorithm can update these cells and move the nodes. Though the compaction algorithm still needs two scans, it now does not have to scan the pointer cells of a node any more. It is even possible to use the threading trick described before [1, 4, 8]. The only objection against this method is, that in the marking phase we may need the (original) value of the cell we use as a list head for the list of cells threaded to a node (e.g. because this value indicates the type of the node). Upon returning to a node (using a recursive marking algorithm like [7, p.567]) or arrival at a node (using an iterative marking algorithm like [7, p.560]) we may find this value to be at the end of a long list. As a result, getting hold of this value may involve an additional overhead. If we do not need this value in the marking phase, there is no overhead. Before incorporating Algorithm 1 in a compacting garbage collector, the method of threading cells in the marking phase should therefore first be considered. If not every garbage collection automatically involves a compaction, e.g. if compaction is dependent upon the fragmentation of the free space, this method should of course not be applied. So, in each case where we need a stand-alone compaction algorithm for the problem described in Section 1, Algorithm 1 is the best choice.

REFERENCES

[1] Dewar, R.B.K. and A.P. McCann,
MACRO SPITBOL - a SNOBOL4 compiler,
Software-Practice and Experience 7 (1977), 95-113.

[2] Fitch, J.P. and A.C. Norman,
A note on compacting garbage collection,
The Computer Journal 21 (1978), 31-34.

[3] Haddon, B.K. and W.M. Waite,
A compaction procedure for variable-length storage elements,
The Computer Journal 10 (1967), 162-165.

[4] Hanson, D.R.,
Storage management for an implementation of SNOBOL4,
Software-Practice and Experience 7 (1977), 179-192.

[5] Knuth, D.E.,
The Art of Computer Programming, Vol. 1: Fundamental Algorithms,
Addison-Wesley, Reading, Mass. (1968).

[6] Lockwood Morris, F.,
A time- and space-efficient garbage compaction algorithm,
Communications of the ACM 21 (1978), 662-665.

[7] Thorelli, L.,
Marking algorithms,
BIT 12 (1972), 555-568.

[8] Thorelli, L.,
A fast compactifying garbage collector,
BIT 16 (1976), 426-441.

[9] Wegbreit, B.,
A generalised compactifying garbage collector,
The Computer Journal 15 (1972), 204-208.

ONTVANGEN 28 MEI 1979