Making Garbage Collection Independent of the Amount of Garbage

by

Dan Sahlin

SICS R87008
Research Report
ISSN 0283-3638

Appendix to SICS Research Report R86009 "*Garbage Collection for Prolog Based on WAM*":

Making Garbage Collection Independent of the Amount of Garbage

Dan Sahlin

SICS
Box 1263
S-163 13 SPÅNGA
SWEDEN

Electronic mail: `dan@sics.uucp`

Abstract

This appendix shows in detail how to make the time for the garbage collection algorithm presented in [AHS 86] become proportional to $n \log n$, where n is the number of non-garbage cells.
It is assumed that the reader is familiar with [AHS 86] since no further presentation of the notation used is made here. The compaction phase of that algorithm is proportional to the amount of memory, i.e. proportional to the sum of garbage and non-garbage. This is unfortunate since for instance a garbage collection using *copying* is just proportional to the amount of non-garbage. If a program generates much more garbage than non-garbage, it might be a severe drawback of a garbage collection algorithm to depend on the amount of garbage. This paper shows how to make the garbage collection in [AHS 86] independent of the amount of garbage.

Introduction

This appendix shows in detail how to make the time for the garbage collection algorithm presented in [AHS 86] to become proportional to $n \log n$, where n is the number of non-garbage cells.
It is assumed that the reader is familiar with [AHS 86] since no further presentation of the notation used is made here. This paper suggests that an extra phase, called *gathering*, is inserted between the marking phase and the compaction phase. Since the compaction phase needs to scan all non-garbage sequentially, the gathering constructs a linked list of blocks with the marked cells.

Each block consists of a link cell and a sequence of non-garbage cells. The gathering is made after the marking since the link cell must be a garbage cell, and this is not possible to determine until the whole marking is done.
When the compaction is given this linked list of blocks, it is able to skip potentially large amounts of garbage, thus saving considerable time scanning through garbage.
Like the rest of this garbage collection algorithm, no extra storage is needed, just the two bits allocated for each word will be used.
First the gathering is described and then the modifications to the compaction.

Gathering

The gathering consists of two parts:

- **Linking non-garbage**: This is simply done by using a modified version of the marking algorithm. Whenever a cell is marked (again), the *following* cell is also investigated. If it is garbage, it will form a link cell of a block of non-garbage. Each link cell is then linked into a list of all link cells.
- **Sorting the blocks**: As the linked list of block is unordered, and the compaction needs to scan all cells sequentially, the linked list is sorted.

The sorting method used (mergesort described below) is the only part of the algorithm proportional to $n \log n$, where n is the number of non-garbage cells. All other parts are directly linear to the number of non-garbage cells, thus the algorithm as a whole becomes proportional to $n \log n$, for large ns.

Linking non-garbage

As mentioned, the gathering repeats the marking, this time inserting into a linked list a garbage cell found just before a non-garbage cell.

Recall that in the marking two bits called m and f were used to distinguish four states in the following manner:

Bit usage for the marking phase

m	f	cell type
FALSE	FALSE	Unmarked
TRUE	FALSE	Marked
FALSE	TRUE	Unmarked internal (i.e. not the first cell of a structure)
TRUE	TRUE	Marked internal (i.e. not the first cell of a structure)

Right after the marking, the memory only consists of Unmarked and Marked cells. Since the gathering is just a repeated variant of the marking, all cells encountered are already Marked, i.e. have their m-bit TRUE and their f-bit FALSE. If the gathering also would consider those cells marked, nothing would be accomplished since it would return immediately. To make the gathering work properly, those Marked cells must anyway be *viewed* as Unmarked. Similarly the still Unmarked cells after the marking are now known to be Garbage.
If the gathering instead uses the following table, the marking can be repeated again successfully.

Bit usage for the gathering phase

m	f	cell type
TRUE	FALSE	Unmarked
FALSE	TRUE	Marked
FALSE	FALSE	Unmarked internal or Garbage
TRUE	TRUE	Marked internal

The two bits are used to distinguish *five* types of cells: Unmarked, Marked, Marked internal, Unmarked internal and Garbage. Below it will be explained how it is possible for Unmarked internal and Garbage cells to share the same bit-pattern.
The bit pattern for Marked cells is chosen so that minimum modifications will be necessary for the compaction which follows. Instead of having the m-bit set TRUE for Marked cells, the compaction will instead find the f-bit set TRUE for those cells. Thus essentially the same algorithms will do for the compaction, just the f and m bits have to be switched conceptually.

The bit-pattern for Unmarked internal cells is chosen so that both bits are set FALSE, i.e. the same as for Garbage. The reason for this choice is the following. Let us examine the possible markings the cells may have during the gathering.
First note that a single cell just reachable from a VAR-cell may only either be Unmarked or Marked.
For a sequence of cells only being reachable from a LIST or a STRUCT-cell the situation is more complex. Not only may all cells be Unmarked or Marked. Halfway through the marking the first cell is Unmarked, some cells may be Unmarked internal, one cell is Marked internal and finally some cells may be Marked (Figure 1). The situation is complicated further if we also consider the case where some of the cells of the structure have been individually marked since they have been reachable from VAR-cells. Then the first cell is either Marked or Unmarked, then some cells being either Marked internal or Unmarked internal, one cell being Marked internal and finally some Marked cells (Figure 2). This is the most complex situation that may occur and that has to be considered.

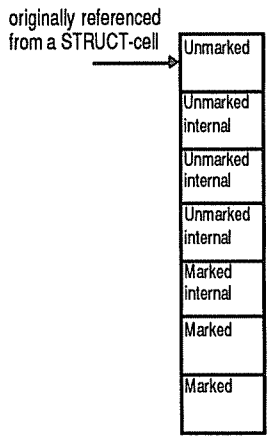


Figure 1

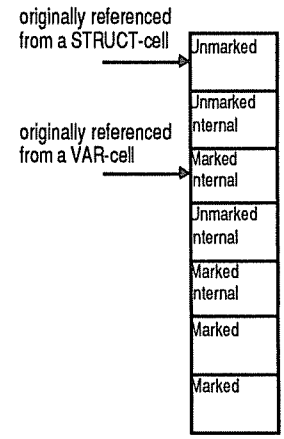


Figure 2

It is now clear that the last cell of a sequence of reachable cells may not be Unmarked internal. This fact can be used so that when the gathering algorithm encounters an Unmarked cell it investigates the *previous* cell, and if both the m and the f-bit are set false it can safely be assumed to be Garbage. That Garbage cell will then serve as a link-node towards the low end of this sequence of reachable cells. (Recall that the cells in the figures are numbered downwards, so that the low end is actually at the top of the figure.) This approach was also pursued in an earlier version of this appendix, but has now been abandoned in favor having the link-node towards the high end of the block which is more advantageous in the compaction phase.
This alternative approach, which we will use henceforth, instead investigates the *following* cell whenever an Unmarked cell is encountered. By "following" for a structure, we mean the cell directly after the whole structure. We must no convince ourselves that if both mark-bits are set false in the following cell, it cannot be an Unmarked internal cell. We do that by examining the various cases.
The following cell may be the first of a sequence of reachable cells and, by the argument above, must either Unmarked or Marked. Alternatively, that cell may be the second cell of a reachable structure. This may happen if the first cell is referred both from a LIST and a VAR-cell. In that case the cell may be Marked internal (Figure 3). What is crucial here is that first cell cannot be referred from both a STRUCT and a VAR-cell, since it violates the assumptions made about the Prolog data structures (Figure 4). We are very lucky here, since otherwise the following

cell could very well be Unmarked internal, which is indistinguishable from a Garbage cell.

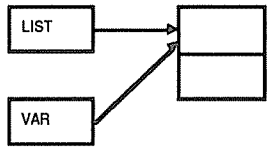


Figure 3

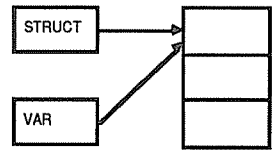


Figure 4

Finally the only alternative for the Marked internal cells is to have both their bits set.

Modifications due to the changed bit usage

It is a fairly straightforward exercise in program transformation to convert the marking algorithm to the gathering algorithm which uses the two bits differently.
When comparing the bit usage of the marking and the gathering phase, it can be seen that the m-bit of the marking simply corresponds to the f-bit of the gathering. For the f-bit however, there is no such simple mapping. The meaning of the m and f-bits taken as separate entities is no longer as clear; they must instead often be considered in pairs. The corresponding modifications to the code will often make it necessary to test and assign both bits where just one was sufficient previously.
Using some invariants of the marking algorithm will however simplify the code again.

For example, consider the following piece of code in the *marking* phase

```
backward:
/* invariant: current->m == TRUE */
if(current->f == FALSE)
        { Undo(current,next); goto backward;}
/* invariant: current->f == TRUE */
current->f = FALSE;
```

which actually means

```
backward:
/* invariant: current->mf == Marked OR
        current->MarkedInternal */
if(current->mf == Unmarked OR
  current->mf == Marked)
        { Undo(current,next); goto backward;}
/* invariant: current->mf == MarkedInternal */
if(current->mf == MarkedInternal)
        current->mf = Marked;
```

The corresponding code for the *gathering* phase then becomes:

```
backward:
/* invariant: (current->m == FALSE AND current->f == TRUE)OR
        (current->m == TRUE AND current->f == TRUE) */
if((current->m == TRUE AND current->f == FALSE) OR
   (current->m == FALSE AND current->f == TRUE))
        { Undo(current,next); goto backward;}
/* invariant: current->m == TRUE AND current->f == TRUE */
if(current->m == TRUE AND current->f == TRUE)
        {
        current->m = FALSE;
        current->f = TRUE;
        }
```

By using the invariants, this can in turn be "optimized" into

```
backward:
/* invariant: current->f==TRUE */
if(current->m == FALSE)
        { Undo(current,next); goto backward;}
/* invariant: current->m == TRUE AND current->f == TRUE */
current->m = FALSE;
```

The code of the gathering

The code of the gathering phase is very similar to the marking phase. The main procedure `gathering_phase` is defined as follows:

```
gathering_phase()
        {
        gather_registers();
        gather_environments(E,env_size(L));
        gather_choicepoints(B);
        }
```

These procedures correspond very closely to their counterparts in the marking phase, except for the way the tests of the m-bit are replaced by tests of the f-bit.
The procedure `gather_variable` also resembles `mark_variable`, but since the f-bit is used and tested here, the transformation becomes more complex.

```
gather_variable(start)
        struct valuecell *start;
        {
        struct valuecell *current, *next;

                current = start;
                next = current->value;
                /* invariant: current->mf == Unmarked */
                current->m = FALSE;
                /* invariant: current->mf == Unmarked Internal */
                total_marked = total_marked-1;
                goto forward;

        forward:
                if(current->f == TRUE) goto backward;
                current->m = not current->m;
                current->f = TRUE;
                total_marked = total_marked+1;

                switch(current->tag) {
                case VARIABLE:
                        if(next->m == next->f)
                                goto backward;
                        /* test if following cell is garbage: */
                        if((next+1)->m == FALSE AND
                             (next+1)->f == FALSE)
                                link_block(next+1);
                        Reverse(current,next);
                        goto forward;
                case CONSTANT:
                        goto backward;
```

```
                case LIST:
                case STRUCTURE:
                        if((next+1)->m == (next+1)->f)
                                goto backward;

                        for every cell in structure referred by next,
                        except head of structure
                                invert the m bit;

                        next = Lastcell(current,next);
                        /* test if following cell is garbage: */
                        if((next+1)->m == FALSE AND
                             (next+1)->f == FALSE)
                                link_block(next+1);
                        Reverse(current,next);
                        goto forward;

                }

        backward:if(current->m == FALSE)
                        { /* internal cell */
                        Undo(current,next);
                        goto backward;
                        }
                /* head of chain */
                current->m = FALSE;
                if(current == start)
                        return;
                current = current-1;
                Advance(current,next);
                goto forward;
        }
```

Notice also in the code above that the procedure `link_block(next+1)` is called whenever `next` is followed by a garbage cell. This cell will serve as a link cell for the block list of marked cells starting from `BL` which is originally initialized to `NULL`.

```
link_block(v)
        struct valuecell *v;
        {
        v->value = BL;
        v->m = TRUE;
        BL = v;
        }
```

Sorting the linked list

Since the compaction needs to scan the cells sequentially in the correct order, the linked list of block has to be sorted. *List merge* [Knuth 73, page 165] seems to be a good choice of sorting algorithm since it does not need any extra storage, besides one bit per node, and the execution time is proportional to $n \log n$, even in the worst case.
For completeness the full code of the sorting algorithm is given here.
Since the blocks have to be sorted in descending order, the `KEY(x)` should be defined as `-x`.

```
mergesort()
{
struct valuecell l0, l1, *s, *t, *p, *q;
int toggle;

/* first split the list BL into two lists s and t using a
   toggle to alternate between the lists */
s = NULL;
t = NULL;
toggle = TRUE;
while(BL≠NULL)
        {
        if(toggle)
                Reverse(s,BL) /* for definition see [AHS 86] */
        else
                Reverse(t,BL)
        toggle = not toggle;
        }

l0.value = s;
l1.value = t;
/* sorting by merging the two lists starting from l0 and l1 */
while(l1.value ≠ NULL)
        {
        s = &l0;
        t = &l1;
        p = s->value;
        q = t->value;
        /* merge links from p and q and insert by alternating */
        /* between s and t */
        while(q != NULL)
                {
                if(KEY(p) < KEY(q))
                        {
                        s->value=p;    /* insert smaller */
                        s = p;         /* and advance pointers */
                        p = p->value;
                        if(s->m == TRUE)
                                {
                                /* end of sub-list: */
                                /* insert rest of other sub-list */
                                s->m = FALSE;
                                while(s->m == FALSE)
                                        {
                                        s->value = q;
                                        s = q;
                                        q = q->value;
                                        }
                                Swap(s,t);/* start inserting at other */
                                }
                        }
                else
```

```
                        { /* symmetrical wrt p and q, see above */
                        s->value = q;
                        s = q;
                        q = q->value;
                        if(s->m == TRUE)
                                {
                                s->m = FALSE;
                                while(s->m == FALSE)
                                        {
                                        s->value = p;
                                        s = p;
                                        p = p->value;
                                        }
                                Swap(s,t);
                                }
                        }
                }
        s->value = p;
        t->value = NULL;
        }
/* when l1.value is NULL then l0.value contains the sorted list */
BL = l0.value;
}
```

Modifications to the compaction

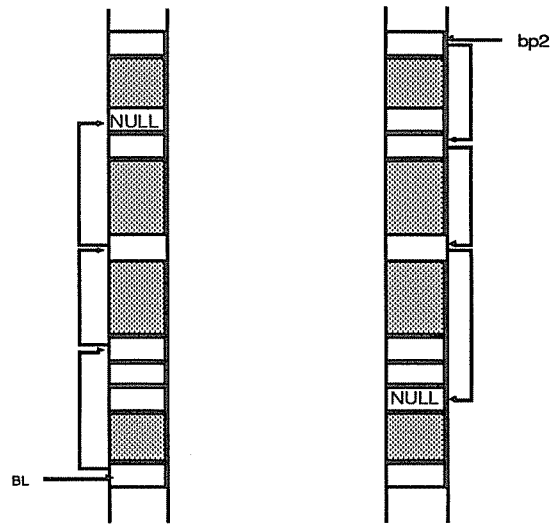
A considerable effort has been made in the gathering phase so that the compaction phase only has to scan blocks containing non-garbage. The new modified `compact_heap` now becomes

```
compact_heap()
{
struct valuecell *dest, *current, *bp, *bp2, *bpvalue;

bp2 = NULL;
bp = BL;
if(bp==NULL)  return;
bpvalue = bp->value;
/* the upward phase */
while(true)
        {
        current = bp-1;
        while(current->f==TRUE)
                {
                update_relocation_chain(current,dest);
                if(current->value is a heap pointer)
                        {
                        if(current->value < current)
                                into_relocation_chain(current->value,current);
                        else if(current == current->value)
                                /* a cell pointing to itself */
                                current->value = dest;
                        }
                dest = dest-1;
                current = current-1;
                }
```

```
        if(bpvalue==NULL)
                break; /* break out of the while-loop */
        bp = bpvalue;
        bpvalue = bpvalue->value;
        current->value = bp2;
        bp2 = current;
        }

current->value = bp2;
bp2 = current;

/* the downward phase */
dest = heap_low;
while(bp2 ≠ NULL)
        {
        current = bp2+1;
        /* just advance bp2 */
        bp2 = bp->value;

        while(current->f==TRUE)
                {
                update_relocation_chain(current,dest);
                if(current->value is a heap pointer AND
                        current->value > current)
                        {
                        /* move the current cell and insert it into
                                the relocation chain */
                        into_relocation_chain(current->value, dest);
                        dest->tag = current->tag;
                        }
                else
                        {
                        /* just move the current cell */
                        dest->value = current->value;
                        dest->tag = current->tag;
                        dest->m = FALSE;
                        }
                dest->f = FALSE;
                dest = dest+1;
                current = current+1;
                }
        }
}
```

The blocks of non-garbage have their link-nodes towards their high ends. This suits the first half of the compaction very well since the block are scanned from high to low. However, in the second half of the compaction the blocks are scanned from low to high. This is accomplished by reversing the pointers during the first traversal, and also by moving the link-nodes towards the low end of each block.

The figure below illustrates the various phases during compaction, where the shadowed areas denote blocks of reachable cells.



The figure also illustrates the fact that an extra cell may have to be allocated at each end of the heap to store the starting point of the two chains.

In `compact_heap` and all other procedures in the compaction, the usage of the bits f and m must be reversed. Since this is a trivial transformation the corresponding code is not shown here.

Discussion

The central question is of course whether this extra phase actually will speed up the garbage collection. As already mentioned, the answer depends completely on the ratio between the amount of non-garbage and garbage. After the marking phase this ratio is known, and it can be determined whether there is enough garbage to make it worth invoking the gathering phase. If it is not considered worth it, the garbage collection will continue directly with the compaction phase. On the other hand, if it is considered worthwhile, gathering and the modified compaction will be used instead. Thus the gathering is guaranteed not to slow down the garbage collection.

Acknowledgements

I am grateful for the valuable comments given by Karen Appleby on earlier versions of this appendix.

References

[AHS 86] Appleby, Haridi and Sahlin, *Garbage Collection Prolog based on WAM*, SICS Research report R86009, 1986

[Knuth 73] Donald E. Knuth, *The Art of Computer Programming, Volume 3 / Sorting and Searching*, Addison-Wesley 1973