A Compacting Garbage Collector for Unidirectional Heaps

Kent Boortz[1] and Dan Sahlin[2]

[1] kent@erlang.ericsson.se,
Erlang Systems, Ericsson Software Technology, Sweden
[2] dan@cslab.ericsson.se,
Computer Science Laboratory, Ericsson Telecom, Sweden

Abstract. A unidirectional heap is a heap where all pointers go in one direction, e.g. from newer to older objects. For a strict functional language, such as Erlang, the heap may be arranged so that it is unidirectional. We here present a compacting garbage collection algorithm which utilizes the fact that a heap is unidirectional.
Only one memory space is used in our algorithm. In fact, no extra memory is used at all, not even any reserved bits within the cells. The algorithm is quite easy to extend to a variant of generational garbage collection.

1 Introduction

The Erlang functional programming language [5] is typically implemented using a copying garbage collection algorithm. In [4] a completely new method was suggested utilizing the fact that objects on the heap may be allocated so that each object only points to earlier allocated ones. In order to be able to deallocate data in the middle of the heap, without disturbing the order of objects, all objects were linked, each with a 'history cell' showing the time of creation.

Here we propose an alternative method to utilize unidirectionality of the heap which has a quite standard representation of the heap, not using linked lists or history cells.

One main motivation to develop the algorithm was to be able to design a garbage collector for an Erlang engine that could run on systems with more limited memory resources. The algorithm presented has the following properties:

- It does not copy data to a new heap, thereby minimizing memory fragmentation.
- No extra memory is used, not even mark-bits or extra tags.
- The order of the objects on the heap is preserved.
- Execution time is linear proportional to the size of the data areas.
- A version of generational garbage collection is simple to implement.
- Overlapping objects are handled.

The algorithm has two phases:

1. Marking, pointer updating and compaction, all combined.
2. Sliding the heap and updating external pointers.

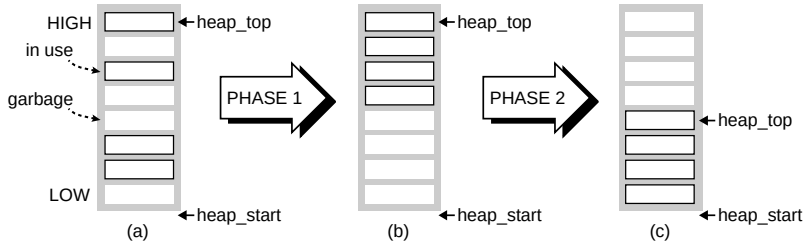

Fig. 1. Phases of the garbage collector

The second phase may sometimes be omitted as a correct heap is returned although in "the wrong place". This optimization is discussed in Sect. 4.5.

The paper is structured as follows. First some basic assumptions are discussed, then the algorithm itself is presented, followed by some optimizations and variants. Finally, the approach is discussed and compared to other approaches.

2 Basic assumptions

The most unusual assumption of the algorithm is the unidirectionality of all heap pointers, i.e. all pointers point from newer to older objects, which will shown in more detail below.

2.1 Memory model

In a typical Erlang implementation there are four memory areas to be considered:

1. Registers: holding function arguments and local variables
2. Stack: holds function activation records
3. Heap: holds dynamically created objects
4. "Static" data: e.g. code and symbol tables

The algorithm presented will only free memory in the heap. Three basic assumptions are made:

1. There may not be references from the heap to the stack or to the registers.
2. All references to the heap come from the registers or the stack.
3. All references in the heap are *unidirectional*, i.e. go from a more recently allocated object to an earlier allocated one. This is because new objects are created on the top of the heap in sequential order.

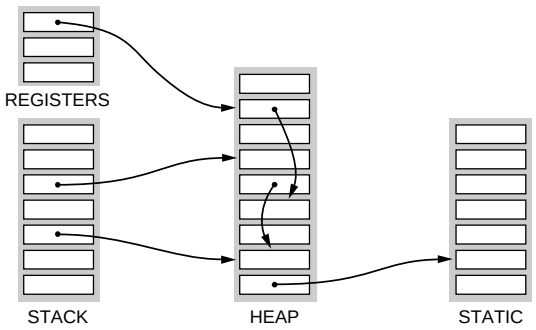

Fig. 2. The data areas

The first two assumptions are quite standard. For most programming languages, e.g. lazy functional, logical or imperative languages, the third assumption would be unrealistic. For a strict functional language without updateable structures, like Erlang, it is in fact an assumption that is almost automatically fulfilled as newly created objects can only point to older ones. However, a copying garbage collection algorithm would immediately destroy this property. Our algorithm, in contrast, preserves the unidirectionality and will also strongly utilize it.

2.2 Unidirectional memory in Erlang

At first, the unidirectionality assumptions may seem unreasonable in the sense that no useful programming language could fulfill them. However, for Erlang the unidirectionality assumptions can be fulfilled, and Erlang is indeed a practical and useful programming language, as demonstrated by a number of very large programs (more than 100.000 lines of code) used in products being sold by Ericsson[3].

Perhaps one key factor making Erlang a practical language is the fact that processes may be created within the language. The processes communicate by message passing. A special data type, *process identifier*, is used when referring to a process. Each process has its own heap which is garbage collected separately as there may be no references into the heap from other processes.

However, a heap may well contain a process identifier referring to a different process, thereby effectively making it possible to create cyclic process references. As only the heaps of the processes, not the processes themselves are being garbage collected, these cyclic references cause no problem for our garbage collection algorithm.

2.3 Data representation

The description is simplified by assuming that the heap only contains three types of objects; tuples, list cells and atoms. All cells are tagged. Typically a cell fits

into a 32-bit word, with a few bits used for the tag and the rest of the word containing a pointer or an atom identifier. Note that no bits or tags are reserved for garbage collection. In the following pseudo-C code `tag` is typically two bits and union `v` is the remaining 30 bits.

```
typedef struct cell {
        unsigned int tag;           /* ATOM, LIST, TUPLE, ARITY */
        union { struct cell *ptr;   /* LIST or TUPLE pointer */
                int id;             /* ATOM id */
                int size;           /* ARITY size */
              } v;
      } Cell;
```

An `ATOM` cell corresponds to an Erlang atom, e.g. 'hello', where the `id` field is unique for the atom. A `LIST` cell points to the first of two consecutive cells. A `TUPLE` points to a sequence of cells where the first cell is an `ARITY` cell containing the length of the sequence.

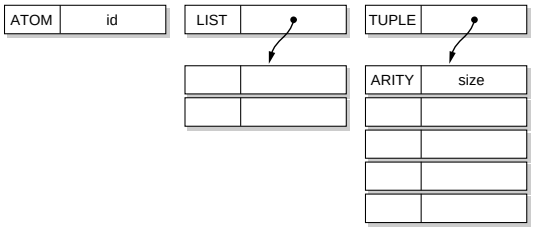

Fig. 3. Representation of data types in Erlang

In the garbage collection algorithm it will be assumed that the pointer to an object points to the part of the object closest to the top the heap. However, as will be shown in Sect. 4.6, this assumption may be lifted and that in fact most other kinds of objects could also be handled by the algorithm.

3 Garbage collection algorithm

3.1 Relocation chains

The *relocation chains* are central for the garbage collection algorithm. A relocation chain for a heap cell `c` is a linked list starting at `c` and containing all the cells that originally pointed to `c`. The original contents of the heap cell is found at the end of the chain.

But having a relocation chain, how is the end of the chain found? It may contain a pointer just as the rest of the cells!

Here the unidirectionality of the memory area is used. Since all heap cells originally only contain pointers pointing downwards, all cells but the last one in a relocation chain point upwards. For a member of the relocation chain located on the *stack* (which we define to be higher than the heap) this is trivially true. As the stack is swept from high to low it will remain true even if several pointers from the stack point to the same cell on the heap.

3.2 Invocation of the garbage collection

Only objects not reachable from a set of *root nodes* may be deallocated during garbage collection. These root nodes are the current registers[1] and the stack frames. The structure of the algorithm is shown below.

```
garbage_collection()
{
    int distance;
    push_registers();
    link_cells_in_stack();
    distance = collect_heap();
    slide_heap(distance);
    update_stack_pointers(distance);
    pop_registers();
}
```

In order to be able to point to the registers during garbage collection, these are pushed onto the stack. The procedure `push_registers()` which does this is not shown in detail here.

To simplify the description of the rest of the algorithm, we will assume that the stack is located above the heap. The algorithm may easily be modified to cater for a different placement of the heap by making the function `points_up()` somewhat more complicated.

[1] A mechanism must exist for deciding which registers are "current". This is often determined by looking at the code pointed to by the program counter. We will not elaborate on this, as this is a standard practice in garbage collection.

3.3 Linking cells in the stack

All cells, including the pushed registers, in the stack are scanned for pointers to the heap by `link_cells_in_stack()`, see Fig. 4.

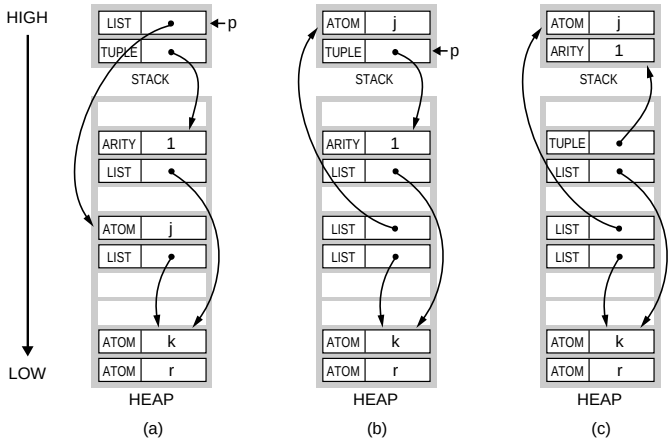

Fig. 4. Linking the cells in the stack. The two cells in the stack point to the Erlang terms [j | [k | r]] and { [k | r] }. In (b) the reference to the `LIST` has entered its relocation chain, and in (c) the reference to the `TUPLE` has also entered its relocation chain.

```
link_cells_in_stack()
{
    for each cell p in the stack from high to low
        if p is a pointer to the heap
            link_cell(*p,p);
}

static void link_cell(Cell c, Cell *to)
{
  Cell *x;

  x = c.v.ptr;
  *to = *x;
  c.v.ptr = to;
  *x = c;
}
```

The procedure `link_cell(c,to)` will link cell `c` into the relocation chain for the heap cell that `c` points to. `link_cell()` is more general than needed by `link_cells_in_stack()` as `c` is always equal to `*to`. In the main procedure, `collect_heap`, that will however not be the case.

3.4 Collecting the heap

In this section, the main garbage collection procedure, `collect_heap()` (Fig.5), is described. It will remain true that all but the last member of the relocation chain in the *heap* will also point upwards.

Two global variables are used:

```
Cell *heap_start;
    /* pointing to the cell just before the heap bottom */
Cell *heap_top;
    /* pointing to the most recently allocated heap cell */
```

When entering the heap we know exactly which objects are reachable from the stack: those in a relocation chain, i.e. pointing upwards. The pointing direction will be used instead of a mark bit.

When collecting the garbage, the heap is swept from high to low using a `from` pointer. Each cell determined not to be garbage gets moved to the next free cell at the top of the heap (the `to` cell).

```
static int collect_heap()
{
  Cell *to, *from;

  to = heap_top;
  fields = 0;
  for (from = heap_top; from > heap_start; from--)
    {
      /* update pointers in relocation chain */

      Cell c = *from;

      while (is_pointer(c.tag) && points_up(to,c.v.ptr))
        {
          fields = max(fields,pointer_length(c.tag));
          c = unlink_cell(c, to);
        }

      if (fields > 0)           /* cell is reachable */
        {
          fields--;
          fields = max(fields,object_length(c));
          if(is_pointer(c.tag))
            link_cell(c,to);    /* enter relocation chain */
          else
            *to = c;
          to--;
        }
    }
  return (to - heap_start);
}
```

Fig. 5. The main garbage collection procedure: `collect_heap()`

There are two ways a cell may not be garbage: it is either reachable directly from an other cell, or the cell is a field in a larger object whose head is reachable. Recall that all cells reachable from the stack have pointers going up.

During the traversal of the heap, if a cell points up, it is the first cell in a relocation chain, and all cells in that chain should now be changed to point to the new location of the cell. This operation is performed by calling `unlink_cell(c,to)` until the end of the chain is reached.

While traversing the relocation chain, the variable `fields` is updated so that it contains the length of the referred object. The function `pointer_length()`

returns the length of an object. However, for the Erlang tuples, the length is stored in the object itself, and this will also be catered for below. For simplicity and generality, the maximum of all `pointer_length()` calculations will be stored in `fields`.

If `fields` becomes greater than zero, the cell is reachable and is moved towards the top of the heap. The function `object_length()` checks if the cell has the tag `ARITY`, in which case the field gets updated to the length of the tuple.

Finally, for a reachable object, if it is a pointer, it is entered into the relocation chain for the cell that it refers to.

Thus `collect_heap()` compacts the heap and updates all pointers. Unfortunately the heap gets compacted in the "wrong" direction, towards the top of the heap, so it has to be slided down. The distance that the heap needs to get slided is returned by `collect_heap()` and is used by `slide_heap()`.

```
static Cell unlink_cell(Cell c, Cell* to)
{
  Cell *next, next_c;

  next = c.v.ptr;
  next_c = *next;
  c.v.ptr = to;
  *next = c;
  return next_c;
}

int is_pointer(unsigned tag)
{
  return (tag == LIST || tag == TUPLE);
}

static int points_up(Cell *addr,Cell *ptr)
{
  return (addr < ptr);
}

static int pointer_length(unsigned tag)
{
  if(tag == LIST)
    return 2;
  else
    return 1;
}
```

```
static int object_length(Cell c)
{
  if(c.tag==ARITY)
    return c.v.size;
  else
    return 0;
}
```

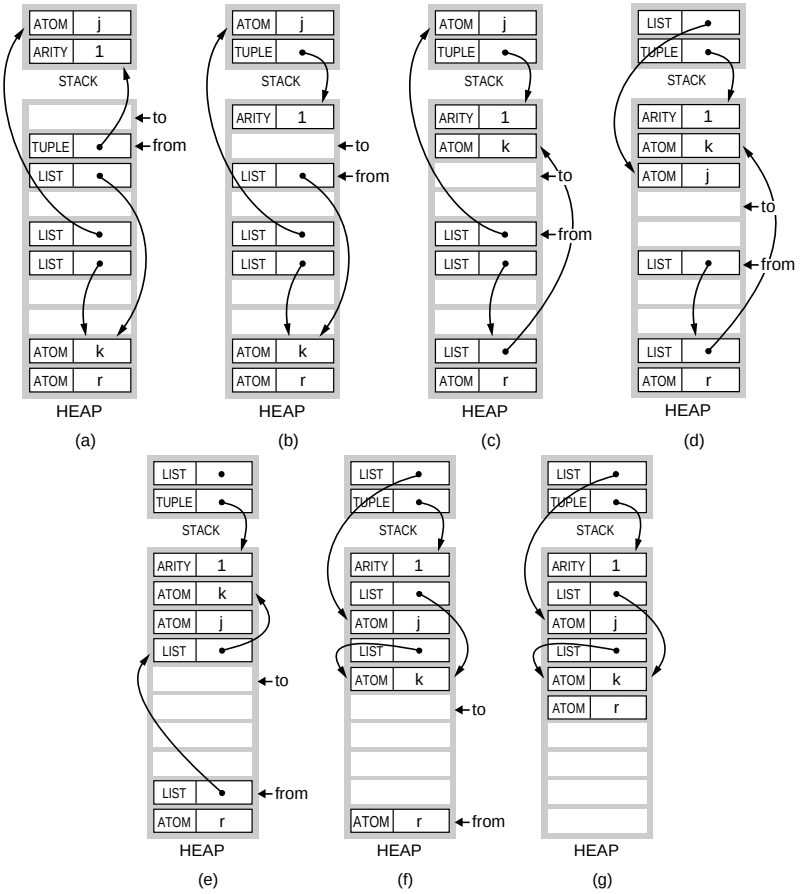

Fig. 6. Compacting the heap, continuing the example from Fig. 4

.

The `from`-cell in Fig. 6 (a) points upwards indicating it being reachable and the beginning of a relocation chain. An `unlink_cell` operation is therefore performed on this `TUPLE` cell which has arity 1.

In (b) the `from`-cell points downwards, but is nevertheless reachable as a `TUPLE` structure of arity 1 contains two cells. So the `from`-cell containing a `LIST` pointer has to be linked into the relocation chain for the list being referred.

The heap is continued being scanned downwards with the `from` pointer until a cell containing a upward going pointer is found in figure (c). An `unlink_cell` operation is performed for the `LIST` cell found, see figure (d). As a list structure always contains two cells, the next cell, pointed to by `from` is also reachable.

That cell also contains a `LIST` pointer, and is entered into the relocation chain for that list and the `from`-pointer is advanced to the next up-going pointer, see figure (e).

The `from`-cell now points to the beginning of a relocation chain, containing two elements, corresponding to the fact that originally there were two pointers to this list cell, see Fig. 4 (a).

In (f) that relocation chain has been traversed, updating the pointers to the new location of the list structure. Finally, in (g) the `ATOM` is moved as it is reachable being the second of the two list cells.

3.5 Heap sliding

When the heap gets slided down all pointer objects become updated.

```
static void slide_heap(int distance)
{
  Cell *from, *to, c;

  to = heap_start+1;
  for(from = to+distance; from <= heap_top; from++,to++)
    {
      c = *from;

      if(is_pointer(c.tag))
        c.v.ptr -= distance;

      *to = c;
    }
  heap_top = to-1;
}
```

Also the stack needs to get updated as it still contains pointers to the old locations on the heap. This is done by calling `update_stack_pointers(distance)`

which updates all pointers to the heap. It now only remains to restore all registers by calling `pop_registers()`.

4 Performance

cell type		copying gc	compacting gc
non-reachable		0	1R
reachable non-pointer		2R + 1W	2R + 2W
reachable pointer	unique	2R + 3W	4R + 4W
	non-unique	3R+ 2W	

Fig. 7. Comparison between Cheney copying GC and our compacting GC with respect to the number of read (R) and write (W) operations for each cell to main memory.

One way to estimate the performance of a garbage collection algorithm is to look at how many read and write operations are required to the main memory for each cell.

If a cell is not reachable, it nevertheless has to be read once, as shown in Fig. 7. If overlapping objects are not allowed, this may be reduced to one read operation per *object* instead of one read operation per *cell*, see Sect.4.6 ("Non overlapping objects").

A reachable cell that is not a pointer (i.e. an `ATOM` or `ARITY` cell) is copied to the new location requiring one read and one write operation. Together with the heap sliding in pass two, which performs one read and one write operation per reachable cell, that sums up to two read operations and two write operations per cell.

In pass one, a reachable cell containing a pointer will be swapped with what it points to and copied in one step requiring two read and two write operations. We then have to add one read and one write operation as part of the unlinking. In pass two, the heap sliding adds one read and one write operation, so in total we get four read and four write operations for a reachable pointer cell.

The table in Fig.7 compares our algorithm with a typical Cheney copying garbage collector[10]. The work done for each reachable cell is higher in our algorithm mainly because of the heap sliding that requires one additional read and write operation. In Sect.4.5 it is shown how the heap sliding phase may be eliminated, thereby making the two methods more comparable. Our algorithm reads data that is garbage and this is a major drawback compared to the Cheney algorithm if the heap contains lots of garbage. Sections 4.4 and 4.6 discuss how to improve on this situation.

4.1 Generational garbage collection

Some programs gain garbage collection speed by only performing garbage collection to the top cells of the heap[7]. The rational for this is that usually most objects are short-lived, and garbage collection for the newly collected objects may be particularly useful.

Generational garbage collection is extremely simple to implement, just two things have to be changed:

1. `heap_start` is moved up in the heap. In fact, `heap_start` may be set at any point in the heap, thereby making it very flexible to decide how much of the heap will be garbage collected.
2. `is_pointer()` returns false for all pointers below `heap_start`

This optimization resembles generational garbage collection [7], but old objects are never moved into "old space" as the order of the objects must be preserved.

4.2 Towards real-time garbage collection

One of the main motivations in [4] to utilize the unidirectionality of the heap was the possibility to get a truly real-time garbage collector. In their system the garbage collector may interleave with ordinary execution, and suspend and resume at any time.

In the currently distributed Erlang JAM system each process has its own heap [2] and uses a traditional copying algorithm. This makes the average process heap a lot smaller than in a unified heap system and garbage collection time has less influence on the response time. But in order to maintain good real-time response, each process heap must not become too large in JAM.

Our algorithm is not a true real-time garbage collector as [4], but will like [2] collect each process heap separately. However, by invoking generational garbage collection the execution time collecting each heap can be made even shorter. The garbage collection time can be fairly accurately predicted as it is limited by a linear function of the heap size to be collected and the stack size. However, when the oldest cells become garbage it may occasionally be necessary to do a complete garbage collection.

4.3 Trimming the heap

If the most recently created structures are garbage, the top of the heap will contain only non-reachable data.

Again we may utilize the fact that the heap is unidirectional, as we know that nothing in the heap above the topmost cell with an up-going pointer is reachable.

By keeping track of which cell is the topmost reachable heap cell when `link_cells_in_stack()` is executed, the heap only needs to be scanned starting from that cell and downwards.

In a similar manner, `collect_heap()` and `link_cells_in_stack()` may be extended to keep track of the lowest object reachable. When that object is reached during heap scanning, and it does not contain any pointers, that will be the last object reachable on the heap.

An alternative way of finding the last object on the heap is using a global reference counter, as outlined below.

4.4 A global reference counter

The algorithm may be enhanced by maintaining a global reference counter which counts up for each call to `link_cell()` and down for each `unlink_cell()`. When this global reference counter reaches 0, we know that all reachable objects have been handled, and there is no need to continue sweeping the heap.

If that global reference counter is called `ref_count`, besides the changes in `link_cell` and `unlink_cell` only `collect_heap()` needs to get modified as shown in Fig. 8.

Normally the counter seldom would reach 0 much before the end of the heap, as most programs create long-lived data in the beginning of an execution. But in combination with a generational garbage collector, those long-lived data would not add to the global reference counter, making this enhancement more likely to be useful.

```
if (fields > 0)           /* cell is reachable */
  {
    fields--;
    fields = max(fields,object_length(c));
    if(is_pointer(c.tag))
      link_cell(c,to);    /* enter relocation chain */
    else
      *to = c;
    to--;
  }
else if (ref_count == 0)
  break;
```

Fig. 8. Adding a global reference counter. `collect_heap()` in Fig.5 is changed so that the heap sweep is stopped as soon as `ref_count` reaches 0 and `fields==0`

4.5 One pass GC: skipping the heap sliding

The heap sliding may be skipped if it is acceptable that the memory freed is at the bottom of the heap rather than the top of the heap.

For an operating system where the memory management unit may be somewhat controlled by the user, it might be possible to return arbitrary sections of the memory. In that case, the heap sliding might not always be necessary.

An alternative approach does not rely on the co-operation of the operating system. Instead the heap is allocated as a linked list of numbered blocks. Blocks may be allocated at the top of the heap and deallocated at the bottom of the heap, thereby eliminating the need to slide the heap.

The test for telling if a pointer "points up" however becomes somewhat more complicated. An address comparison is only sufficient if two pointers belong to the same block, otherwise the numbers of the blocks must be compared.

For this to be practical there must exist a fast method that given a pointer find the number of the block that it points to. One standard way to accomplish this is to allocate aligned blocks of the same size, e.g. 1 kbyte blocks on even 1 kbyte boundaries. Given a pointer, the beginning of the block is found by clearing the low ten bits (1 kbyte=2^{10} byte) of the pointer.

4.6 Other data objects

In the code above only three kinds of objects were handled: atoms, lists and tuples. Most other objects may be catered for by only changing `pointer_length()` and `object_length()`. For instance, a short integer is stored almost like an atom. Similarly, a pointer to a more static area should also be treated like the `id` field of the atom cell, i.e. it should be unchanged by garbage collection.

The algorithm is very easy to extend to handle almost all sorts of objects as all pointers to the object, as well as the object itself of course, are available just before the object is moved.

Binary data on the heap In the algorithm it is assumed that all cells are tagged, but this restriction can easily be removed. E.g. a binary object usually contains a tagged word with length information in the first cell. When the heap is scanned, the binary object is simply moved. However, the heap sliding operation goes from low to high addresses, and at the end of the object no distinguishing tag might be present.

This problem is solved by swapping cells: In `collect_heap` after having moved the binary object, the first and last cells are swapped so the length information is stored at the end of the object. Then, during heap sliding, the length information is read (so that the objects is not scanned for pointers to updated), the object is moved, and the cells are swapped back again.

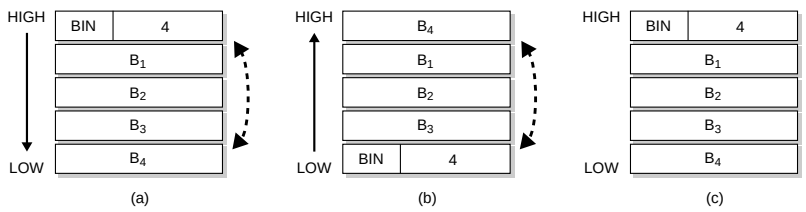

Fig. 9. Moving binary header while scanning in both directions

Overlapping objects The algorithm actually accepts that pointers may refer to parts of objects or even overlapping objects. Figure 10 shows a situation that may not occur in ordinary Erlang execution, but that is nevertheless handled correctly by the algorithm.

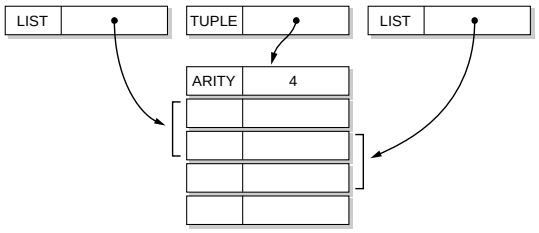

Fig. 10. Overlapping objects

Non overlapping objects In Erlang, objects are not overlapping and all references to an object are to its first cell. Thus, when it is possible to calculate the size of the object from the first cell (e.g. a tuple), a non-reachable object when encountered may be skipped entirely. This optimization is particularly important when the heap contains many large objects that are no longer reachable.

Objects with the header towards the bottom of the heap In the above presentation it has been assumed that a pointer to an object points to the part of the object closest to the top the heap. For an implementation based on copying garbage collection the arrangement is probably the opposite: the head of the object is towards the bottom of the heap.

The algorithm can be changed to cater also for objects of that kind. When the marked head of the object is encountered (`points_up(from)` is true), the `from` pointer is reset to a position higher up on the heap where the object actually started and the `fields` value is set to the length of the object. Thus the whole object will then become copied.

There is a slight performance penalty when objects are stored in this way during garbage collection: To find the size of an object the relocation chain

starting at the head of the object must first be searched. Only then can we know where the object will be relocated, and the relocation chain has to be traversed again to do the relocation.

It does not seem possible to extend the algorithm to handle overlapping objects, or to skip over unreachable objects when the header is located towards the bottom of the heap.

5 Related work and discussion

Already in 1974, David Fisher published a paper on garbage collection for non-cyclic, i.e. unidirectional, list structures[6]. Mark bits were used for reachable data, and the algorithm needed three passes, two of which swept all memory cells.

One main source of inspiration for us was [4] where the advantages of a unidirectional heap were first exploited, implementing the heap as a linked list of numbered objects. That method has excellent real-time properties as the garbage collection may be stopped and resumed at any time during a computation. However, there is an up to 50% overhead in memory consumption due to the storage of the "history counters". Maintaining the free list of objects during execution and updating the "history counter" for each cell is also probably quite time consuming.

Pointer reversal techniques have successfully been used for marking of live data [1] and for compacting the heap [8]. By utilizing the unidirectionality of the heap we found that a single sweep through the heap was sufficient to both mark, compact and relocate all data. This phase sometimes has to be followed by a heap sliding to relocate the data towards the beginning of the heap. It seems unusual to exploit the unidirectionality of the heap in garbage collection. In a survey [10], no such techniques were mentioned.

Traditionally garbage collection based on copying has been used in Erlang implementations [9]. Copying garbage collectors only traverse live data, whereas the proposed algorithm may have to scan over all data once.

Still there are some advantages with our proposed algorithm. The historical order of the data is preserved and that often gives good locality of the data. It is easy to implement generational garbage collection that may be good enough for the real-time requirements. Last, and perhaps most important, no extra memory is used so fragmentation of the main memory is reduced.

6 Acknowledgments

Our colleagues at Ericsson have given us many helpful comments. The constructive comments by the referees have made us focus on a number of weak points in our presentation.

References

1. Karen Appleby, Mats Carlsson, Seif Haridi, and Dan Sahlin. Garbage collection for Prolog based on WAM. *CACM*, pages 719–741, 1988.
2. J. L. Armstrong, B. O. Däcker, S. R. Virding, and M. C. Williams. Implementing a functional language for highly parallel real time applications. In *SETSS 92*, 1992.
3. Joe Armstrong. Erlang – a survey of the language and its industrial applications. In *INAP'96 – The 9th Exhibitions and Symposium on Industrial Applications of Prolog*, October 1996. Hino, Tokyo, Japan.
4. Joe Armstrong and Robert Virding. One pass real-time generational mark-sweep garbage collection. In *International Workshop on Memory Management 1995*, 1995.
5. Joe Armstrong, Robert Virding, Claes Wikström, and Mike Williams. *Concurrent Programming in ERLANG*. Prentice Hall, 1996.
6. David A. Fisher. Bounded workspace garbage collection in an address-order preserving list processing environment. *Info. Proc. Letters*, 3(1), July 1974.
7. H. Lieberman and C. Hewitt. A real time garbage collector based on the life time of objects. *CACM*, 26(6):419–429, 1983.
8. F.L. Morris. A time and space efficient garbage compaction algorithm. *CACM 21*, 21(8), 1978.
9. Robert Virding. A garbage collector for the concurrent real-time language Erlang. In Henry G. Baker, editor, *International Workshop on Memory Management 1995*, number 986 in Lecture Notes in Computer Science. Springer, September 1995. ISBN 3-540-60368-9.
10. Paul R. Wilson. Uniprocessor garbage collection techniques. In *International Workshop on Memory Management 1992*, volume 637. Springer-Verlag, September 1992. A much expanded version of the paper is available from http://www.cs.utexas.edu/users/oops/papers.html.