From (multi-)generational to segment order preserving copying garbage collection for the WAM

Konstantinos Sagonas
Computing Science Department
Uppsala Universitet
S-751 05 Uppsala, Sweden
kostis@csd.uu.se

Bart Demoen
Department of Computer Science
Katholieke Universiteit Leuven
B-3001 Heverlee, Belgium
bmd@cs.kuleuven.ac.be

Abstract

We develop an algorithm for generational copying garbage collection in the WAM based on three generations. We then generalize this 3-generational algorithm to any number of generations. A new segment order preserving copying garbage collection algorithm then follows in a natural way. In contrast to previous segment preserving algorithms, the new algorithm does not require traversing the stacks in the opposite order, at the cost of a $O(log(n))$ pointer lookup where n is the number of collected heap segments. A trade-off between precision in the preservation of heap segment order and $O(1)$ pointer lookup is presented. The algorithm gives an overall simple and practical way of implementing multi-generational copying garbage collection. Issues related to preserving generations on backtracking and cut are also discussed, as well as the impact of generations on early reset.

1 Introduction

Heap garbage collection has been studied in the context of Prolog and the WAM [16] in several places: [1, 14, 2, 3, 7, 9]. Two dimensions in the whole spectrum deserve particular attention: copying compacting techniques and generational techniques. Copying techniques are of interest when there is a high percentage of unreachable (garbage) data overall, and generational techniques are interesting when the generational hypothesis holds: when young objects are short-lived or die much more quickly than old objects; see e.g., [10, 8]. There has been a reservation against copying techniques within the Prolog community mainly because copying garbage collection does not retain the order of segments and thus prevents cheap—actually constant time—reclamation of the heap on backtracking. However, [3] presents data showing that in practice the performance cost of this loss is not so bad, while [5] shows how the order of the segments can be retained during the copying compaction so that the cheap reclamation of the heap on backtracking is not affected. Also, simple variants of generational copying garbage collection have been studied before; see Section 2. Generational garbage collection is quite natural in the context of the WAM: while in a different language (functional or imperative) the consideration of generations requires the introduction of a write (or read) barrier and an exception list, the WAM already caters for these concepts in the form of conditional trailing, because this mechanism is needed for backtracking-based execution. However, there are also some quirks to consider: as a generation is naturally associated with a (set of) heap segment(s) delimited by a choice point(s), the question arises what to do in deterministic computations. This issue is discussed in Section 6. Also, generational schemes based on just two generations (old/new) are sometimes unsatisfactory as they promote newly created objects too early: there is nothing between the nursery and the sanctuary. At least in the context of functional [12] or object-oriented [17, 15] languages, garbage collection schemes with more than two generations have been developed and proven to be useful. Still, in the context of Prolog, there is no evidence whether a scheme based on multiple generations (hereafter termed *n-generational*) is worthwhile, as no such implementation has ever been tried. There does not even exist a clear insight in the algorithms needed to implement such a scheme. This is exactly what we will do first: develop a generational copying algorithm based on three generations in Section 3. Already in this scheme, there is a number of different choices that we will make explicit. In the next step, Section 4, we generalize this scheme to support any number of generations and describe exactly the associated space and time costs. We also show that by mapping heap segments—as delimited by choice points—to generations, we arrive naturally at a segment order preserving collection scheme. Section 5 shows that a trade-off between the amount of preserved segments and the associated time cost can be made. To make the paper self-contained, we begin by a brief overview of garbage collection techniques and by introducing some terminology.

2 Garbage Collection Techniques: Overview and Related Work

In the WAM, the heap and trail are segmented by choice points. That is, a new heap (and trail) *segment* is started after the creation of a new choice point on the stack. Furthermore, the order of segments (and of their cells) is preserved. Thus, upon backtracking out of a choice point, the heap segment allocated after the choice point creation can be *instantly reclaimed* (in the terminology of [2]).

Mark&slide garbage collection

In order not to lose the ability to perform instant reclaiming of the heap upon backtracking, Prolog implementations have usually opted for a mark&slide garbage collector. At the higher level, marking makes explicit which cells are reachable so that this information can later be used in constant time and also counts the number of reachable cells—this number is needed in the sliding phase. At the lower level, marking follows a set of root pointers and the Prolog terms they point to while setting a mark bit for each reachable cell. Marking can be performed by a recursive function, a self-managed stack, or by a pointer reversal algorithm [1]. The sliding phase compacts the heap by shoving the useful cells in the direction of the heap bottom, thereby making a contiguous area free for cheap allocation by pointer bumping. At the same time the root pointers are relocated: Morris' algorithm [11] is frequently used. Apart from the mark bit for each heap cell, sliding also requires a chain bit for each cell in the root set and in the heap. Notable characteristics of sliding are that the order of cells on the heap is retained and that its complexity is linear in the size of the collected heap.

Copying garbage collection and the need for marking

Compared to a sliding collector, a copying garbage collector is attractive because its complexity is linear in the size of the useful data in the heap (i.e., heap cells that are garbage are not visited). The usual copying algorithm (Cheney's [4]) compacts the heap (named the *from-space*) by copying the terms reachable from root pointers to a second semi-space (named the *to-space*). After the collection is finished, the roles of the two semi-spaces are switched and the computation happens in the old *to-space*. Naturally, the allocation of new terms also happens at this second semi-space. Later collections will again switch the roles of the two semi-spaces. The main characteristics of copying garbage collection are: 1) the order of cells on the heap is not necessarily retained and this interferes with WAM's instant reclaiming; 2) the computation can use only half of the available heap space (because the two semi-spaces exist at the same time).

Copying garbage collection algorithms in general do not require marking. However, as pointed out in [3], in the context of the WAM, double copying of internal cells is possible and thus one runs the risk of ending up with a heap that is *bigger* than before garbage collection. Obviously such a situation is undesirable and either a marking phase should be performed (as in [3, 5, 7]), or one has to postpone the treatment of such cells (as recently in [18]). Even though the latter method performs well in the context of the collector described there (the garbage collection is restricted to the topmost segment only), postponing the treatment of internal cells seems impractical for collection of large portions of the heap. Furthermore, as was noticed in [5], the marking phase is independently useful because it counts the number of useful cells, which leads in turn to a more economical utilization of memory by permitting an allocation of an *exact to-space* area. We will henceforth assume that copying is always preceded by a marking phase, the allocation of the *to-space* is exact and happens only on demand, and that at the end of the copying phase a *copyback* operation of the *to-space* to the *from-space* is performed: a simple memcopy suffices for this because all pointers are set to their final destination in the first pass. The additional advantage of this copyback is that the heap remains contiguous. This simplifies the usual WAM operations and its cost is negligible.

Generational garbage collection

Generational garbage collection [10] exploits the dynamic property exhibited by many programs that most objects die young while a small percentage live much longer. The heap is divided into a number of areas, called *generations*, with each area containing objects of a particular range of ages. The generations are collected independently with the younger generations being collected more often. The frequent young generation (or *minor*) collections reclaim the space of the many short-lived objects without incurring the execution cost of collecting the entire heap which contains all the long-lived data (which is likely to be still useful). In addition, as reported in the literature, minor garbage collections have a much smaller working set and result in less paging. Objects which survive long enough are *promoted* (or *advanced*) to an older generation, thus avoiding the cost of repeatedly rescuing them during minor collections.

In order for generational schemes to properly work, it should be possible to find all references from objects in old generations (those that are not collected) to objects in the collected ones *without* visiting the old generations. This can be achieved by the introduction of a *write-barrier*[1] which records the creation of inter-generational pointers on an *exception list* so that these pointers can be used as part of the root set during each minor collection. This results in a non-negligible runtime cost for supporting generational scavengers. As mentioned, the WAM already caters for these concepts in the form of *conditional trailing* and there is no extra cost associated with generational garbage collection provided that generations consist of a set of heap segments (as delimited by the choice points). This property has been exploited in the form of *segmented garbage collection* (i.e., where each generation coincides with a heap segment) in the mark&slide collector of SICStus [1]. Except for the maintenance of exact generations after garbage collection (an issue discussed in Section 6), adding generations to a sliding collector is relatively straightforward as sliding preserves

[1] Also *read-barrier* methods exist.

the order of segments.

On the other hand, in the context of Prolog, only simple variants of generational copying garbage collection have been studied before: Touati and Hama [14] use a copying collector when the new generation consists of the topmost heap segment (i.e., when no choice point is present in the new generation), while Bevemyr and Lindgren use a 2-generational scheme with an immediate promotion policy which performs well on a set of three small benchmarks [3]. The experience from another 2-generational scheme, having the additional constraint that the region to collect must fit in the cache, is similar; see [9].

Segments and generations seem to be closely related: a generation is a set of contiguous segments. However, as we will describe in more detail in Section 5, there is a third structure on the heap that can be superimposed on segments and generations: we will name it a *page*. A page can span several segments and generations. Also segments or generations can be spread over more than one page. This independent page structure allows for more flexible generation management and provides a time-space tradeoff between pointer lookup during collection and instant reclaiming.

3 A 3-generational Garbage Collection Scheme

We name the three generations the *nursery*, the *kindergarten*, and the *sanctuary* (also known as *mature object space*). The aim is to delay the promotion of young data that might soon die by one (or more) minor garbage collection cycle(s). Such a 3-generational scheme is well known in other contexts [15, 8] and the usual implementation—using two kindergarten areas of equal size—is depicted in Figure 1. In words: at garbage collection time, a 3-generational scheme employs a more conservative tenuring policy that promotes useful data from the old kindergarten to the sanctuary (identified as action 1 in the figure) and from the nursery to the new kindergarten (action 2). The roles of the kindergartens are then interchanged. This scheme is potentially more effective than a 2-generational one, because it is better at avoiding promotion to the sanctuary of data that are very young when garbage collection takes place and were not yet given a proper chance to die.

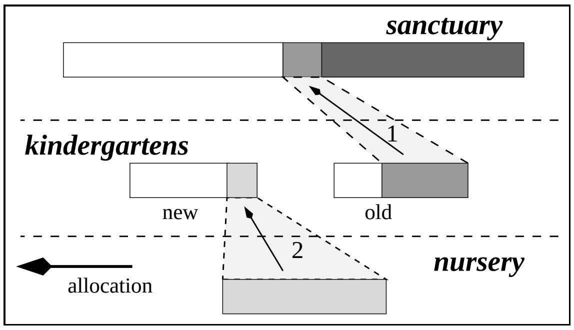


Figure 1: A generic scheme for 3-generational garbage collection.

In the context of the WAM however, one also has to take into account backtracking and in particular the ability to perform instant reclaiming. In order to support this, usually Prolog implementations allocate the heap as one contiguous area. This means that putting the three generations in different areas as in Figure 1 is not practical. Also, we would like to utilize the heap optimally, that is, we do not want to leave space within the contiguous heap between the generations. We will henceforth assume that the heap is one contiguous area where the generations appear consecutively in reverse chronological order.

An initial scheme

In Algorithm 1 we present our initial 3-generational copying garbage collection scheme.

Algorithm 1 A 3-generational copying garbage collection (first take).

1. Mark the useful cells in the nursery and the kindergarten while keeping a count of how many useful cells each contains, say N and K;
2. Allocate a *to-space* of size $(N + K)$ and an area that can hold a bit which indicates whether a *to-space* cell belongs to the old nursery (when not set) or to the old kindergarten (when set);
3. Perform a Cheney copy—as described in e.g., [3]—of the useful cells in the two youngest generations of the heap to the *to-space*, filling the bit; pointers are relocated in this step so as to point to the final destination of the cell they point to;
4. The *copyback* from the *to-space* to the heap is now a simple loop:

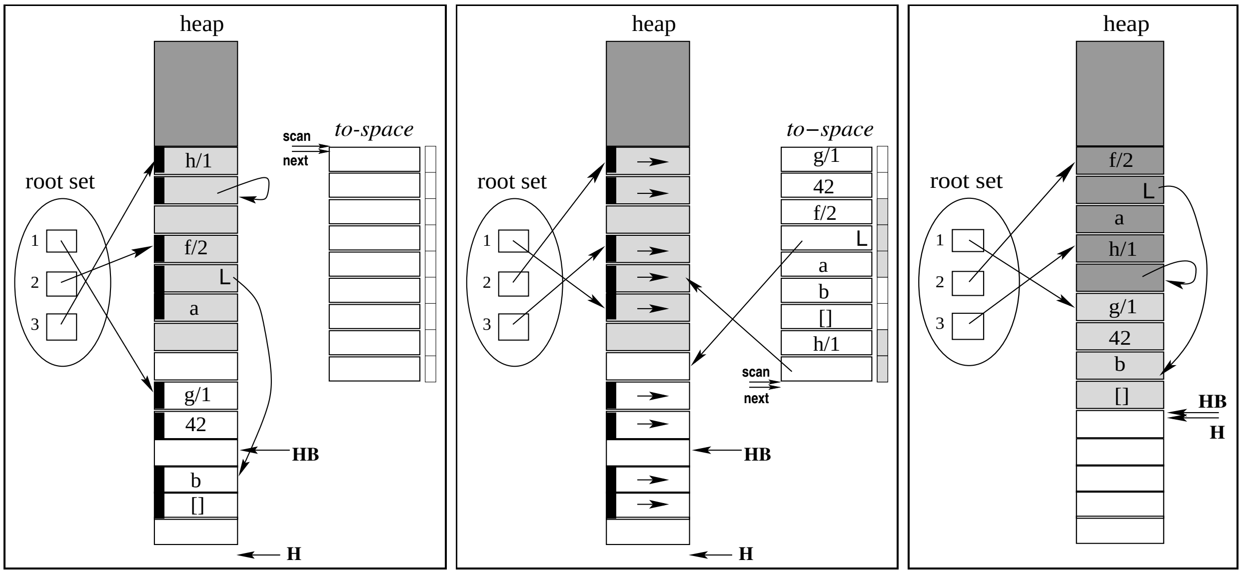
   $kindergarten\_p = kindergarten\_begin$; /* oldest cell of kindergarten */
   $nursery\_p = kindergarten\_p + K$;
   for ($p = tospace\_begin$; $p < tospace\_end$; p++)
       if ($in\_oldkindergarten(p)$) **then** $*kindergarten\_p$++ $= *p$; **else** $*nursery\_p$++ $= *p$;
5. Promote objects according to the *promotion policy* employed;
   for example, for immediate promotion use:

   $kindergarten\_begin = kindergarten\_p$; /* alternatively: $kindergarten\_begin$ += K; */
   $nursery\_begin = nursery\_p$; /* set it to current top of heap */

Using an example, the first two steps of the algorithm are illustrated in Figure 2(a): starting from three root cells, the useful cells of the nursery and kindergarten areas of the heap have been marked. Note that the nursery consists of more than one segment (this can be seen by the **HB** register). The heap mark bits are shown as black rectangles at the left of the cells; heap cells without values are garbage and their values are not shown. An exact *to-space* of 5+4 cells is allocated together with a parallel bit array (shown to the right of *to-space*) that keeps track of the generation of the corresponding *to-space* cell. The situation after the end of step 3 of the algorithm is shown in Figure 2(b): useful cells have been rescued by following the root cells in the numeric order shown and the result is a *to-space* where nursery and kindergarten cells are intermixed. Note how root cells and pointers in the *to-space* have been relocated to point to their final destination. The small arrows in the marked cells on the heap, denote the forwarding pointers left by the copying algorithm. The *copyback* to the heap (step 4) is simple; it just requires two pointers (one for each of the two areas in the old heap). As shown in Figure 2(c), the outcome of the algorithm is a heap in which the kindergarten cells can be added easily to the sanctuary (by moving the boundary between them) and by immediately promoting the nursery cells into the kindergarten (by setting the nursery boundary to the new heap top). Note that the segments in the old nursery are collapsed and that the order of useful cells within each generation is not necessarily retained.

An alternative scheme

Instead of copying to the *to-space* in such a way that cells from different generations end up intermixed, one can also divide the *to-space* before the rescuing begins to a memory area for the K-cells and one for the N-cells. This is easy to do since the K and N counts are known from step 1 of



(a) After marking & *to-space* alloc.   (b) After Cheney copy to *to-space*.   (c) After *copyback* to heap.

Figure 2: Illustration of Algorithm 1.

the algorithm. We name these two memory areas (which do not necessarily have to be contiguous) segments because after garbage collection they correspond to segments of the heap. The situation when all useful cells have been rescued now becomes as in Figure 3. One of the advantages of this alternative 3-generational scheme is that no extra bits are needed for these segments: one just makes sure that the useful cells of the *from-space* are copied to the correct *to-space* segment. When the two segments are contiguous, another advantage is that, like in a non-generational copying garbage collection, the *copyback* now becomes a simple `memcopy`. After garbage collection is finished, the heap is as in Figure 2(c). The disadvantage is that, as shown in Figure 3, one now needs a separate `scan` and `next` pointer for each of the kindergarten and nursery segments of the *to-space*, i.e. two pointers for each collected generation.

It should be clear that both schemes preserve the complexity properties of copying garbage collection: the work performed is proportional to the size of *useful* cells in the collected generations. The same also holds for the other schemes presented later.

4 From Multiple Generations to Segment Order Preservation

4.1 A n-generational garbage collection scheme

There is a rather straightforward generalization of this alternative scheme to more than three generations: for each generation, one needs a counter keeping track the number of useful cells in it. The update of this counter happens during the marking phase and the main problem there is given a pointer into the heap, find the generation it points to. We name this operation the *lookup* of a heap pointer. Since the boundaries of these generations are known and can be kept as a ordered data structure, the lookup operation can be made logarithmic in the number of collected generations. Known techniques can make the lookup even constant time, but these rely on page

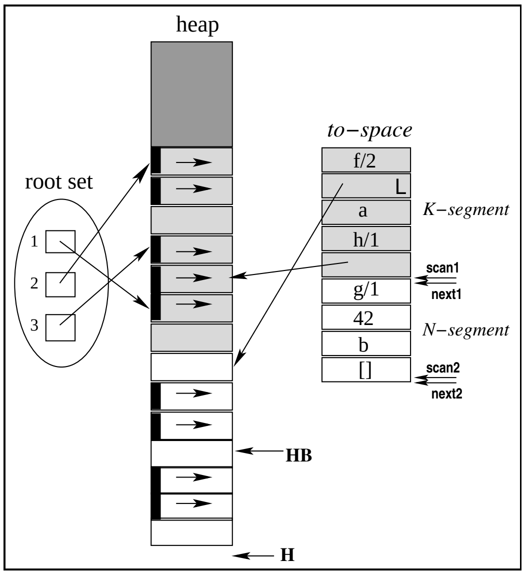


Figure 3: After Cheney copy to a segmented *to-space*.

boundaries and this affects the Prolog heap philosophy. We postpone discussion of this issue till Section 5. Also, one needs a scan and next pointer for each generation.

To sum up: the cost of n generations in this scheme, is the allocation of n counters, n scan and n next pointers, and a $O(log(n))$ lookup function.

4.2 Segment order preserving generations

The schemes presented so far, are not segment order preserving because after garbage collection all segments of particular generation are collapsed into a single segment (cf. Figure 2(c)).

However, it is rather simple to see that we already have enough ingredients for a segment order preserving garbage collector: just let every segment coincide with a generation (i.e., restrict the previous scheme so that generations consist of only one segment). In contrast to the method described in [5], we do not need to traverse the WAM control stacks (choice point and local stack) in the opposite order making the basic scheme simpler. We defer further comparison to Section 7.

4.3 The non-determinism in n-generational copying

The usual (non-generational) Cheney copying algorithm consists of an inner loop of the form:

while ($scan < next$)
  if ($points\_to\_from\_space(*scan)$)
    then $copy\_block(scan)$; /* this advances $next$ */ **fi**
  $scan$++;

For n generations, we have pointers $scan_i$ and $next_i$ for $i = 1..n$. The inner loop now becomes:

while ($(S = \{j \mid scan_j < next_j\}) \neq \emptyset$)
  $i = pick\_from(S)$;
  if ($points\_to\_from\_space(*scan_i)$)
    then $copy\_block(scan_i)$; /* this advances some $next_j$ */ **fi**
  $scan_i$++;

The non-determinism in this algorithm is in the choice of the segment to work on: any choice is correct, and we see no *a priori* reason to prefer one choice over the other. Possibly locality reasons can dictate a particular choice.

5 Pages as an Approximation of Segments

For the sake of explanation, we consider the heap consisting of a region that will be collected and a sanctuary, and perform collections by a (variant of the) segment order preserving copying collector described in Section 4. As we noted there, the cost of looking up the segment number at which a pointer p points is in principle $O(log(n))$ where n is the number of segments. If segments were all the same size (and contiguous as we always have kept them) such a lookup function could be made constant time by the formula:

$$(p - start\_heap)/segment\_size$$

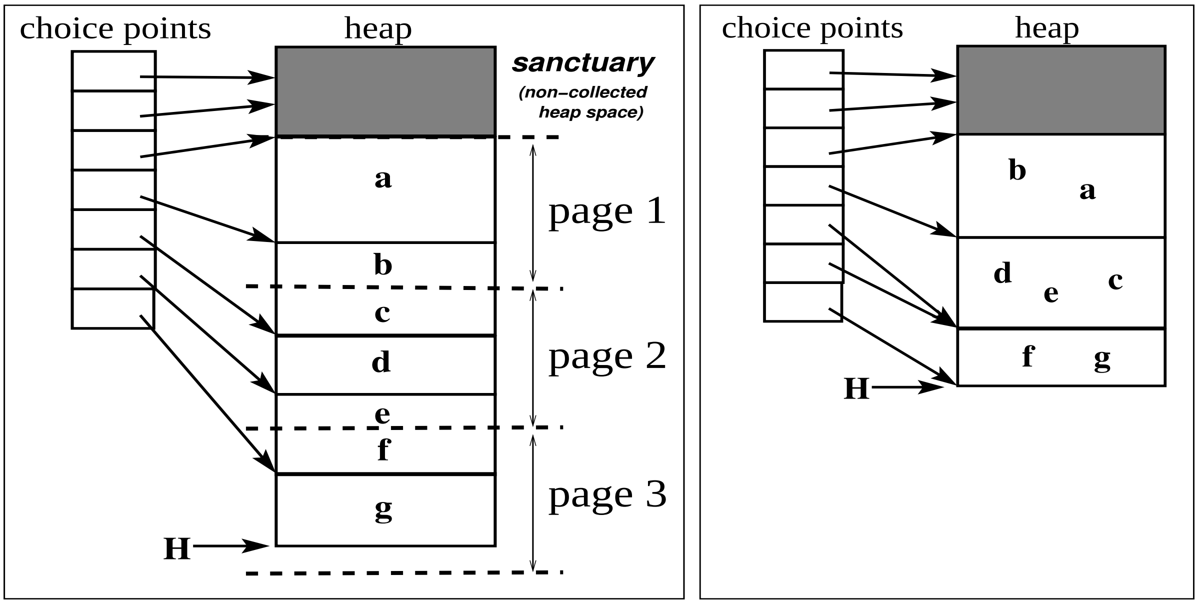
and if the $segment\_size$ were a power of 2, the division could even be replaced by a right shift. Such nicely-sized segments would then each correspond to a *page*. Now suppose that two consecutive segments each have half the size of a page. In order to use the page idea, one could decide to collapse these two segments, i.e., after garbage collection, some segment order is lost, but most is retained. Clearly, one has made a trade-off between the precision with which segment order is preserved and the usage of pages to speed up the pointer lookup. In practice of course, segments do not have nice sizes, so the question naturally arises whether the idea of approximating segments by pages can be extended and what the trade-off really means. First, note that the actual size of a page (power of 2) is not important, so when we use the word *page* later on, it will mean a region with a fixed, but otherwise unspecified size.

Consider the situation of heap and choice point stack in Figure 4(a): the collected heap contains 5 segments, delimited by 5 choice points and the current **H** pointer. Some of the useful data is made explicit: `a`, `b`, `c`, `d`, `e`, `f` and `g`. Three pages have been superimposed on these segments: the page structure is clearly independent of the segmentation. In Figure 4(b), we find the situation after the collection: cells `a` and `b` used to belong to a different segment, but because they belonged to the same first page, they now belong to the same segment. In contrast, cells `b` and `c` used to live in the same segment, but in different pages, so after collection, they end up in different segments. The loss in precision of segments is quite clear now because the new segments correspond to the collected pages, or actually rather to a subset of them, since some pages might contain no useful data and of course one cannot have more segments than choice points.

Multiple generations can also be considered in combination with pages. First of all, the first page must start at the boundary between collected and non-collected generations; this boundary needs to also act as a write-barrier. Secondly, pages now can approximate generations, which means that not all data in a generation after a collection necessarily survived the same number of collections.

It now becomes more apparent that we are dealing with 3 divisions of the heap, which are to some extent independent: segments are prescribed by the H pointers from the choice points; pages are chosen freely, as long as the upper boundary coincides with a segment boundary above which nothing is collected; while generations are collections of data that survived the same number of collections. Because of pages, this notion of generation is not very useful anymore.

Pages show that there is a graceful degradation from totally segment order preservation to the copying scheme of [3] which from the page point of view puts all segments on the same page.



(a) Three pages superimposed on five segments—before GC. (b) Pages have become segments after GC.

Figure 4: Segments and pages before and after garbage collection

If there is the need or wish to absolutely preserve the order of all the segments, one can try to use WAM characteristics to avoid the general lookup procedure, e.g., in the WAM, the arguments of a structure necessarily belong to the same segment as the header. More heuristics might be discovered when more detailed heap profiling data becomes available. Also, there exist techniques which first do an approximate lookup with a simple address calculation which narrows down the logarithmic search afterwards; see e.g., [8].

An extreme case: one cell per page

Suppose heap cells all have the same size (this is true in most Prolog implementations). One can then choose the page size exactly the same as the size of a cell, i.e., each cell has its own page. The count of useful data for each page can be at most 1, and actually the count for a page equals the mark bit for the cell it contains, so no counters for pages are needed. On the other hand, we need a `scan` and a `next` pointer for each page (even for the empty ones if we want constant time access to them); since `scan` and `next` either differ by 1, or are no longer needed, one can do with a `scan` pointer only. The linear table in which to store these `scan` pointers is now actually a translation table from old to new addresses for useful cells and it has the size of the heap. This extreme case corresponds to a scheme that preserves the order of all cells and not just that of the segments as in [5]. In this respect it is like a sliding algorithm, albeit at a very high space cost. It is also reminiscent of break table methods [8].

6 Delimiting Collected Generations

In generational garbage collection, one needs to record somehow the pointers that the mutator writes in a collected generation that point to the new generation. Or put in other words, pointers from the region that will not be subject to garbage collection to the region that will. Read- or write-barriers have been used, but for Prolog, it is natural to use the trailing mechanism as a write-barrier, because the WAM already provides it and making another use of the same mechanism does not put an extra cost on the execution. The general idea is that the most recent choice point at the time of garbage collection will delimit with its H-pointer the collected generation: this is clearly an approximation for a segment order preserving collector (sliding or not).

Besides precision, there are at least four more issues to consider: deterministic execution, backtracking, the Prolog cut, and analysis. The problem with backtracking is that the write-barrier might have to move backwards. The problem with cut is that the write-barrier might disappear. The problem with deterministic execution is that normally no trailing takes place. The problem with analysis is that there exist analysis techniques which can determine that trailing is not necessary. The latter issue can be dealt with in two ways: either not use the results of such a *do-not-trail* analysis, or use the analysis but also store compile time information that allows at garbage collection time to find out which cells have not been trailed because of analysis. The latter technique is reminiscent of live variable maps, or type maps, but as far as we know has never been described nor explored. The rest of this section deals with the other three issues.

6.1 Deterministic execution

Consider the following program:

```
main(N) :-
    mklist(N,L),
    do_gc(N),
    use(L).

mklist(N,L) :-
    (N =:= 0 ->
      L = []
    ;
      M is N - 1,
      L = [N|R],
      mklist(M,R)
    ).

do_gc(N) :-
    N > 0,
    gc_heap,
    M is N - 1,
    do_gc(M).

use(_).
```

For a given integer N, the query `?- main(N).` is completely deterministic: no choice points are created. If not for the call to the collector, its complexity would be linear in N. However, since there is no choice point acting as a delimiting barrier between the collected area and the new area, the complexity is effectively quadratic in N, because the created list (of length N) is subject to collection N times. Such behavior has been observed in practice.

Now consider the following variant of the above program:

```
main(N) :-
    mklist(N,L),
    do_gc(N),
    use(L).

gc_cp :-
    gc_heap.
gc_cp :-
    fail.

mklist(N,L) :-
    (N =:= 0 ->
      L = []
    ;
      M is N - 1,
      L = [N|R],
      mklist(M,R)
    ).

do_gc(N) :-
    N > 0,
    gc_cp,
    M is N - 1,
    do_gc(M).

use(_).
```

The only difference is that just before each collection, a choice point is pushed (by the `gc_cp/0` predicate): this choice point will act as a generation-delimiter choice point which can be used by a generational collector to avoid repeated collection of old data. The effect is that the query becomes linear again. The action to take on garbage collection is thus simply to:

> *push a generation-delimiter choice point (denoted gen_cp).*

For the current purpose, the alternative field (the failure continuation) of *gen_cp* can be set to a special instruction that basically removes the choice point and fails, but which is distinguishable from any other possible alternative. From the choice point chain at the moment of the next collection, it is now easy to recognize the choice point(s) delimiting a generation.

This solution to the determinism problem of generational garbage collection seems folklore and used to be implemented in ECLiPSe but was later switched off by default because it seemed to cause problems.[2] It has recently been adopted in SICStus Prolog. When segment preservation is not an issue, an alternative to pushing a generation-delimiter choice point, is to set the H pointer of the most recent choice point[3] and **HB** to the current top of heap, and perform a similar action for the environment stack. Finally, in a generational scheme with an immediate promotion policy, one can easily avoid pushing more than one such choice points immediately after each other.

This solution retains the advantages of generational garbage collection in deterministic programs, but the next section will illustrate the problems with ...

6.2 Backtracking

We will first assume that one wants to distinguish precisely the data that survived a heap garbage collection from the new data; in Section 6.4 we will shortly discuss the impact of relaxing this requirement. We also need to discuss separately the case of segment order preserving and the case of non segment order preserving garbage collection.

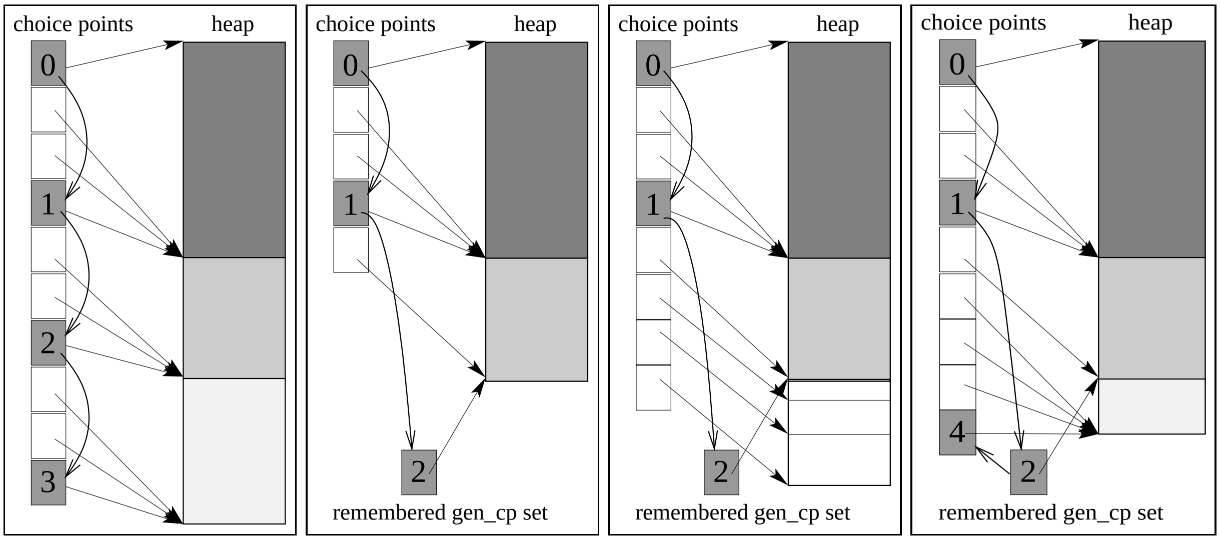
6.2.1 Non segment order preserving garbage collection

We assume the possibility of pushing a *gen_cp* choice point in the choice point stack: *gen_cp* can be distinguished from any other choice point and we can put in its fields whatever information we want. The general idea is to push a *gen_cp* every time a garbage collection occurs. Also, a *gen_cp* is pushed at the top-level, i.e., the oldest choice point on the stack is a *gen_cp*. After a number of collections have taken place, the stacks might look as in Figure 5(a): the *gen_cp* choice points are shown shaded. We keep links in the *gen_cp* choice points from one *gen_cp* choice point to the next younger one. We will also keep an extra data structure—an array will do and its management is not difficult—which contains information about the backtracked over *gen_cp*s as long as they are relevant: we name this *the remembered gen_cp set*. Suppose backtracking occurs to a choice point between *gen_cp*'s `1` and `2`. This happens in several stages: First *gen_cp* `3` is backtracked to: its alternative field points to code that copies the *gen_cp* choice point[4] to the remembered *gen_cp* set. The link from *gen_cp* `2` is set to point to the copy of *gen_cp* `3`. Then *gen_cp* `3` is removed from the stack and failure is initiated. Later, backtracking occurs to *gen_cp* `2`: it is copied to the remembered *gen_cp* set and *gen_cp* `3` is removed from it, because it is no longer the delimiter of an existing generation. The link from (the copy of) *gen_cp* `2` to the copy of *gen_cp* `3` is removed and

[2] Note by J. Schimpf in comp.lang.prolog, October 1999.

[3] One can assume there is always at least one choice point.

[4] It is enough to copy only some of the information in a *gen_cp*.



(a) After 3 collections. (b) After backtracking. (c) Forward execution. (d) After new collection.

Figure 5: Use of generation-delimiter choice points.

the link from *gen_cp* 1 is adapted to now point to a choice point in the remembered *gen_cp* set. *gen_cp* 2 is removed from the stack and failure occurs. The situation becomes as in Figure 5(b).

Let there now be some forward computation, with choice points creating 3 new segments. We arrive at the situation shown in Figure 5(c). If garbage collection now occurs, we can find, starting from the oldest *gen_cp*, all the generations. If we also keep in a *gen_cp* its age, i.e., the number of times this generation was collected, we have complete information on the generations. After garbage collection, we push a new *gen_cp* 4, which is linked to the saved *gen_cp* 2; see Figure 5(d).

6.2.2 Segment order preserving garbage collection

When segments are preserved—either by sliding or using another way—the situation becomes slightly more complicated, as now the boundary of the topmost generation can move because of backtracking. Still, the *CHAT*-trick ([6]) can be applied here: on backtracking over a *gen_cp*, we will promote the now top Prolog choice point to a *gen_cp* by stealing its failure continuation. We make it point to an instruction (called `gen_gc`) that contains as argument the old failure continuation and an indication on where to find the generation and linking information in the array. The situation is slightly different from the one in [6] because the execution of the `gen_gc` instruction must adapt its own operand to reflect the change of alternative of the original choice point, instead of filling out a new alternative in the choice point. Also, depending on whether the stolen Prolog alternative was a `retry` or a `trust`, the action is different, which means that in a native code implementation, this is slightly more complicated than in an emulator. The management of the saved (now possibly promoted) *gen_cp*, is similar to the one in the previous section.

6.3 Cut

Up to now, we have not changed the normal WAM execution, i.e., no WAM action has been changed. With cut, the situation is more involved: one of the normal actions of cut is to reset the **HB** register to the H-value of the youngest choice point that survives the cut. This action potentially destroys the write-barrier(s). Just remembering the barriers (e.g., by saving them as in the previous section) is not enough: if cut does not immediately re-install the write-barrier(s), some essential inter-generational pointers can be lost and it is unsafe to continue considering the generations as areas that can be collected separately.

So, if generations need to be retained exactly on cut, the solution we propose is the following: Cut checks which choice points it cuts away. Any cut away *gen_cp* must be pushed back on the choice point stack, after the normal cut action; this must include any *gen_cp* saved in the remembered *gen_cp* set data structure which is younger than the youngest surviving choice point. This way, all information about generation boundaries is preserved. Then **HB** is set to the H-value in the topmost choice point.

Many Prolog implementations already visit the choice points which are cut away for other reasons, so this scheme seems not to make cut considerably more expensive.

6.4 Relaxing the precision requirement

If the *gen_cp* choice points are not kept in a separate area on backtracking as in Section 6.2.1, or not pushed back as in Section 6.3 one can no longer reconstruct the generations exactly: the *gen_cp* choice points that are on the choice point stack are insufficient in general. However, one can decide to make the management of *gen_cp* choice points simpler, at the cost of suboptimal generation management. An experimental evaluation is needed to determine whether this trade-off is reasonable in practice.

7 Comparison with another Segment Order Preserving Method

As far as we know, only one segment order preserving copying collector in the context of WAM existed before: it was implemented in the BinProlog system [5]. The method starts by marking in the usual way from new to old. Then the copying phase visits segments in the root set (without environments in the BinProlog case, but the technique can be adapted to the usual WAM) from older to younger: in this way heap segments are copied also from old to new, which is one ingredient of the segment order preserving property. There are two other essential ingredients in this algorithm:

1. when a heap cell is copied to its segment and it points to a newer segment, this reference is not treated immediately; instead it is postponed until the moment it is encountered on the trail: it will necessarily appear there because of the trailing mechanism of the WAM;

2. dual to this, it is possible that a heap cell is copied because its first reference was found on the trail: in this case the cell resided in an older segment than the one in which it ends up after collection.

The latter, called *rejuvenation of data*, is a nice property of the collector in [5], because at no extra cost, it manages to make some data available for instant reclaming sooner than normally.

At first sight this rejuvenating potential of [5] seems a good idea. However, in the presence of generations, it is possible that data that flows from an older to a newer segment, is also moved from an older generation to a newer one, e.g.,, to the nursery, which means that possibly the same data is repeatedly subject to minor collections, which is exactly what generational garbage collection tries to avoid. Especially in view of the fact that this rejuvenating behavior cannot be avoided without a high cost, it is not completely clear whether it is an asset in the generational context.

Our segment order preserving collector on the other hand, does not move data from one segment to another: in fact, the traversal from older to newer segments is essential for rejuvenation. Also our method depends crucially on knowledge of the amount of live data in each segment (or generation) before the actual copying starts. Acquiring this information in a rejuvenating collector, would require an extra counting pass between the marking and the copying phase.

A second issue that needed to be dealt with in the context of BinProlog, was the preservation of term compression; see [13]. This is a particular term representation optimization that within the tag-on-data scheme allows for a form of generalized cudder-coding, so that the same space savings as for lists in a tag-on-pointer scheme can be obtained for all binary functors, and in fact for every compound last argument of a structure. It is essential that garbage collection retains this compression, otherwise the collected heap is possibly larger than the original. Since only contiguous marked areas are copied, this is rather straightforward, except that some compressed terms might span segments—or even generations. In that special care must be taken. The same techniques apply to both cases. This is discussed in more detail in Appendix B.

To sum up the differences and similarities between the two segment order preserving copying methods:

- both use the same new to old marking, but the new method also counts per segment (or generation) the useful cells during the marking; in principle, this counting involves a $O(nlog(n))$ time cost where n is the number of segments. If segments coincide with pages, the extra space cost in bits is

  $$log(pagesize) * heapsize/pagesize + 2 * bitsizeof(pointer) * heapsize/pagesize$$

  where *heapsize* is the size of the collected heap; the first term in the formula corresponds to the size of the counters, the second to the space needed for the `scan` and `next` pointers for each page
- during the copying phase, the old method must treat root pointers from old to new: this requires an unnatural traversal of the control stack; in contrast: the new method can copy root pointers in arbitrary order
- the old method is by design rejuvenating: data ends up in the oldest segment from which it is reachable; the new method by design preserves the segments (and the generations) exactly.

8 Conclusion and Future Work

This work provides a better understanding of the issues in generational copying garbage collection in the context of the WAM. One major point is that the structure of generations and segments can be enriched by what we named pages, in order to allow full flexibility and control over the trade-off between the amount of segment order preservation and the associated time and space costs. Our work also shows that although it seems natural to make the generational aspect of the collector cooperate—or even coincide—with its segment order preservation aspect, these are in fact orthogonal issues to some extent.

By no means does this work alone suggest that an n-generational garbage collector ($n > 2$) in a WAM-based Prolog system is useful. In fact, for n large enough, it is unlikely. However, one should not easily dismiss the usefulness of multi-generational schemes; the experience from e.g., functional programming is different (SML/NJ uses a 14-generational scheme [12]). Without an actual implementation and extensive performance measurements, conclusive statements cannot easily be made. We do offer a practical scheme to implement an n-generational garbage collector and consequently plan to implement this scheme in the context of the XSB system where we have already developed a copying collector (described in [7]). We anticipate that apart from experimental evaluation of issues addressed in this paper, the implementation will also provide new data and insights in memory management for the WAM.

Acknowledgments

Most of this work was done while the second author was a guest at the Computing Science Department of the University of Uppsala. He is most grateful for its hospitality.

References

[1] K. Appleby, M. Carlsson, S. Haridi, and D. Sahlin. Garbage collection for Prolog based on WAM. *Communications of the ACM*, 31(6):719–741, June 1988.

[2] Y. Bekkers, O. Ridoux, and L. Ungaro. Dynamic memory management for sequential logic programming languages. In Y. Bekkers and J. Cohen, editors, *Proceedings of IWMM'92: International Workshop on Memory Management*, number 637 in LNCS, pages 82–102. Springer-Verlag, Sept. 1992.

[3] J. Bevemyr and T. Lindgren. A simple and efficient copying garbage collector for Prolog. In M. Hermenegildo and J. Penjam, editors, *Proceedings of the Sixth International Symposium on Programming Language Implementation and Logic Programming*, number 844 in LNCS, pages 88–101. Springer-Verlag, Sept. 1994.

[4] C. J. Cheney. A nonrecursive list compacting algorithm. *Communications of the ACM*, 13(11):677–678, Nov. 1970.

[5] B. Demoen, G. Engels, and P. Tarau. Segment order preserving copying garbage collection for WAM based Prolog. In *Proceedings of the 1996 ACM Symposium on Applied Computing*, pages 380–386. ACM Press, Feb. 1996.

[6] B. Demoen and K. Sagonas. CHAT is Θ(SLG-WAM). In H. Ganzinger, D. McAllester, and A. Voronkov, editors, *LPAR'99: Proceedings of the 6th International Conference on Logic for Programming and Automated Reasoning*, number 1705 in LNAI, pages 337–357. Springer, Sept. 1999.

[7] B. Demoen and K. Sagonas. Heap garbage collection in XSB: Practice and experience. In E. Pontelli and V. Santos Costa, editors, *Practical Aspects of Declarative Languages: Second International Workshop*, number 1753 in LNCS, pages 93–108. Springer, Jan. 2000.

[8] R. Jones and R. Lins. *Garbage Collection: Algorithms for automatic memory management.* John Wiley, 1996. See also http://www.cs.ukc.ac.uk/people/staff/rej/gcbook/gcbook.html.

[9] X. Li. Efficient memory management in a merged heap/stack Prolog machine. In *Proceedings of the 2nd International ACM SIGPLAN Conference on Principles and Practice of Declarative Programming (PPDP'00)*, pages 245–256. ACM Press, Sept. 2000.

[10] H. Lieberman and C. Hewitt. A real-time garbage collector based on the lifetimes of objects. *Communications of the ACM*, 26(8):419–429, June 1983.

[11] F. L. Morris. A time- and space-efficient garbage compaction algorithm. *Communications of the ACM*, 21(8):662–665, Aug. 1978.

[12] J. H. Reppy. A high-performance garbage collector for Standard ML. Technical report, AT&T Bell Laboratories, Dec. 1993.

[13] P. Tarau and U. Neumerkel. A novel term compression scheme and data representation in the BinWAM. In M. Hermenegildo and J. Penjam, editors, *Proceedings of the Sixth International Symposium on Programming Language Implementation and Logic Programming*, number 844 in LNCS, pages 73–87. Springer-Verlag, Sept. 1994.

[14] H. Touati and T. Hama. A light-weight Prolog garbage collector. In *Proceedings of the International Conference on Fifth Generation Computer Systems (FGCS'88)*, pages 922–930. OHMSHA Ltd. Tokyo and Springer-Verlag, Nov./Dec. 1988.

[15] D. Ungar and F. Jackson. An adaptive tenuring policy for generation scavengers. *ACM Transactions on Programming Languages and Systems*, 14(1):1–27, Jan. 1992.

[16] D. H. D. Warren. An abstract Prolog instruction set. Technical Report 309, SRI International, Menlo Park, U.S.A., Oct. 1983.

[17] P. R. Wilson and T. G. Moher. Design of the opportunistic garbage collector. In *Proceedings of the Conference on Object-Oriented Programming: Systems, Languages, and Applications (OOPSLA'89)*, pages 23–35, Oct. 1989. Also published as *ACM SIGPLAN Notices*, 24(10).

[18] N.-F. Zhou. Garbage collection in B-Prolog. In *Proceedings of the First Workshop on Memory Management in Logic Programming Implementations*, July 2000. Co-located with CL'2000. See http://www.cs.kuleuven.ac.be/~bmd/mmws.html.

A The Impact of Generations on Early Reset[5]

The idea of *early reset* (or *virtual backtracking*; see e.g., [2]) in the WAM is that a trailed heap cell which is not reachable for the forward computation (but might be so on backtracking) can be set to unbound during garbage collection and the trail entry itself can be discarded as well. Early reset opportunities are recognized during marking and it is essential that the forward continuation is marked before the alternatives.

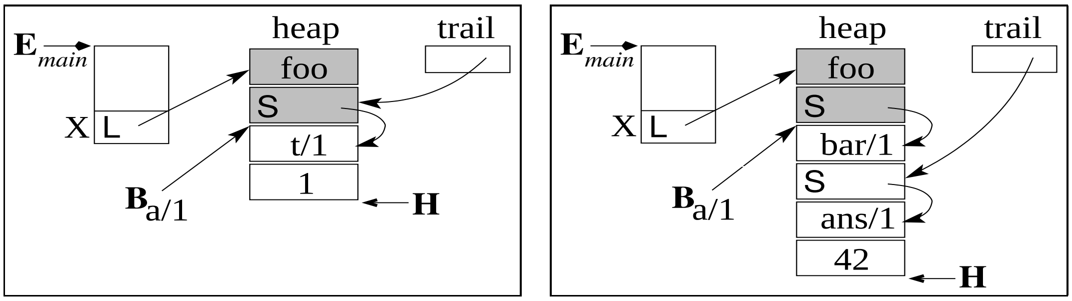
At first sight there seems no reason why early reset should be affected by generations: marking takes place in the order from newer to older segments, exactly as early reset needs. However, the aim of having generations is exactly not to mark nor traverse the old generation (for the sake of simplicity, assume there are just two generations). This means that marking does not follow root pointers (or pointers from the new generation) that point into the old generation. Pointers from the old into the new generation are found on the trail. Early reset in this context is no longer possible. Indeed, consider the following two very similar programs:

<table>
<tr><th>Program 1</th><th>Program 2</th><th>Common to both programs</th></tr>
<tr><td><pre>main :-
   X = [foo|Tail],
   new_generation,
   a(X),
   gc_heap.</pre></td><td><pre>main :-
   X = [foo|Tail],
   new_generation,
   a(X),
   gc_heap,
   use(X).</pre></td><td><pre>a([foo|t(1)]).
a(_).

use(_).

?- main.</pre></td></tr>
</table>

The predicate `new_generation/0` is meant to delimit the two generations: assume the collector was called. In Figure 6(a) this is visible by the shading of the old generation.



(a) State for Programs 1 and 2.

(b) State for Programs 3 and 4.

Figure 6: States just before the trail entry is used: nothing is marked yet.

The situation on the heap is shown in Figure 6(a) at the moment that the trail pointer is used: nothing is marked, even though the environment of `main/0` has been treated already (E_{main} points to this environment). As far as the snapshot is concerned, the situations are the same for both programs: the trail entry points to a non-marked cell. The deeper reason for this is different for the two programs: in the first program, `X` is not used in the forward continuation of the collection. This means that `X`, although in the environment of `main/0`, is not live for the forward computation and should not be marked. In the second program, `X` is live, but since its header resides in the old generation, it is not marked. It is now clear that early reset for the second program is wrong: the

[5]These two appendices contain supplementary material which is not part of the submission.

`use/1` goal would lose the `t(1)` term. Since from the mark bits at the moment of treating the `main/0` query it is impossible to distinguish the case where early reset would be correct (Program 1, since `X` is not even live) from the case where it is incorrect, early reset should not be performed. Moreover, the pointer from the old to the new generation must be followed for marking and this leads, in Program 1, to marking the data structure `t(1)` that will not be used after garbage collection.

The previous example showed that early reset *in the old generation* is not possible; this has also been mentioned in [1], albeit without detailed explanation. The next example shows that *also in the new generation* early reset is forbidden. This has not been reported in the literature.

Program 3	*Program 4*	*Common to both programs*

`main :-`
`    X = [foo|Tail],`
`    new_generation,`
`    Tail = bar(Y),`
`    a(Y),`
`    gc_heap.`
	`main :-`
	`    X = [foo|Tail],`
	`    new_generation,`
	`    Tail = bar(Y),`
	`    a(Y),`
	`    gc_heap,`
	`    use(X).`
		`a(Z) :- Z = ans(42).`
		`a(_).`
		
		`use(_).`
		
		`?- main.`

Whether the continuation of the `gc_heap/0` goal uses `X` or not, cannot be seen from the marking at the moment that trail entry for `Z` is found. Consequently, early reset cannot be performed safely. This time, the trail entry under consideration points to the new generation; see Figure 6(b).

Variable shunting (see e.g., [2]) on the other hand does not require marking from the root set, but from the trail. It is totally compatible with generational garbage collection.

B Preserving Term Compression during Garbage Collection

Term compression is a technique described in [13]. A heap garbage collector must preserve term compression, since otherwise the representation of the live data might grow. Note that a sliding compacting collector preserves term compression without requiring any changes. As for a copying collector, the marking phase which is used to prevent multiple copying of the same internal cell, largely caters for the preservation of term compression: since the region copied is always a maximal contiguous marked region, *most* of the time compressed terms are copied together, so compression is preserved. However, the term compression scheme also breaks the following WAM invariant:

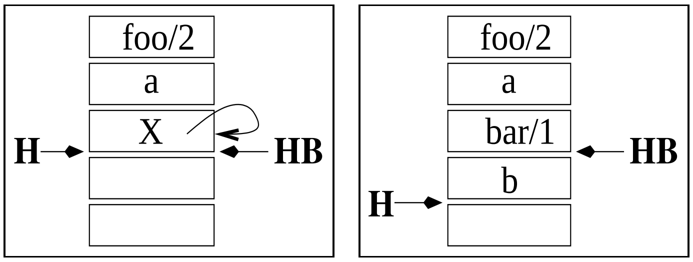
The direct arguments of a compound term belong to the same segment as the term's header.

With term compression, this is no longer true. Consider the execution of `?- main.` against the piece of program shown in Figure 7(a). In Figure 7(c), **HB** points to the segment boundary. It is clear how the term `bar(b)` has its header in the older segment and its argument in the newer one. If the term `foo(a,bar(b))` is live at the moment of garbage collection, its compressed representation must be preserved. Clearly, a copying collector does not have this property in general and changes must be made to it.

```
main :-
    put_in_heap(foo(a,X)),
    new_segment(X).

put_in_heap(_).

new_segment(Z) :- Z = bar(b).
new_segment(...).
```



(a) Program where term compression occurs. (b) Before unification of `Z`. (c) After unification.

Figure 7: Example program and heaps showing term compression across segments.

The idea used in the implementation of [5] can be adapted as follows: In the first phase of the algorithm, before any other cell of a segment S (or generation or page) is copied, one can copy the block of marked cells that span S and its successor to the end of the appropriate part of the *to-space*. A small adaptation in the termination test for the Cheney algorithm [4] is still needed, but this is straightforward. Note that in principle a term can span several segments.