# The implementation of a new segment preserving and/or (multi-)generational copying garbage collection for a WAM and its approximation

*Ruben Vandeginste, Bart Demoen*

*Report CW 319, July 2001*


Katholieke Universiteit Leuven
Department of Computer Science
Celestijnenlaan 200A – B-3001 Heverlee (Belgium)

# The implementation of a new segment preserving and/or (multi-)generational copying garbage collection for a WAM and its approximation

*Ruben Vandeginste, Bart Demoen*

*Report CW 319, July 2001*

Department of Computer Science, K.U.Leuven

**Abstract**

The implementation of previously developed generational copying garbage collection algorithms based on a novel way to preserve segments or approximate segments by pages, is described in the context of the ilProlog system. During the implementation, issues arose that were not foreseen during the introduction and more theoretical study of the principles. In particular cross page pointers and a trade off between generations and trailing were investigated. We present empirical evidence on the usefulness of the alternatives.

# The implementation of a new segment preserving and/or (multi-)generational copying garbage collection for a WAM and its approximation

Ruben Vandeginste
Bart Demoen

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

**Abstract**

The implementation of previously developed generational copying garbage collection algorithms based on a novel way to preserve segments or approximate segments by pages, is described in the context of the ilProlog system. During the implementation, issues arose that were not foreseen during the introduction and more theoretical study of the principles. In particular cross page pointers and a trade off between generations and trailing were investigated. We present empirical evidence on the usefulness of the alternatives.

## 1 Introduction

We assume familiarity with the WAM [18, 1], with Prolog garbage collection as in [2, 4, 8]. For an excellent overview of issues in Prolog garbage collection , see also [3]. More about garbage collection in general can be found in [11]. An early reference to generational garbage collection is [13].

In [15], a new way to preserve heap segments during a *copying* garbage collector is presented: while a *sliding* collector (based on the Morris algorithm for instance - see [14]) preserves even the order of individual cells - and thus a priori also the order heap segments delimited by choice points - a copying collector in general does not preserve the order of individual cells, and it requires extra work (and new algorithms) to preserve segment order nevertheless. One algorithm and implementation that does preserve the order of segments, is presented in [8] and is based on a reverse traversal of the segments during the copying phase. One characteristic of this algorithm is that in principle it *improves* the contents of segments, in the sense that it moves useful data to the most recent segment it is reachable from. However, when combined with a *generational* algorithm, it can also move data from an older to a younger generation. This is contrary to the idea of generational garbage collection , but it also shows that in a context with backtracking, there is no clear cut way to deal with generations. The preservation of order of segments presented in [15] is based on a very simple but powerful idea: count the number of useful cells in a segment. This is possible and useful only because *marking* is mandatory: marking is unusual in a copying context, but it is necessary in principle (see [4]) and useful in practice (see [8]). So, the method presented in [15] also relies on a marking phase that precedes the copying and makes further use of it.

In section 3 we give enough detail on the method in [15] to follow this paper, but we refer to [15] for more details.

We implemented the methods of [15] in ilProlog ([5]): ilProlog is not a standard conforming Prolog system but it is based on the WAM and similar enough to other Prolog systems to make the reported experience in this paper useful for other systems that are WAM based as far as the control data structures and the structure representations.

We start by describing what we want and what we do not want to investigate with the benchmarks in 2. We then explain briefly the basic idea of preserving segments in 3 and present also the basic implementation. Section 4 shows different ways of approximating the exact segment structure: performance is the reason for investigating this. Section 5 shows how to combine use the segment structure for generational garbage collection . in particular, the pushing of a generational choice point (section 5.2.1) and local stack generations are discussed (section 5.2.3). Also the problems in combining pages with generations is explored in section 5.3. Section 6 shows the benchmarking results of our garbage collectors, as well as the overhead of changes to the (ilProlog) WAM (section 6.3; we also discuss the results. Section 7 discusses our basic choices in policy. Section 8 gives a conclusion and indicates future work. The appendix contains a set of extreme and artificial examples.

## 2 Benchmarking philosophy

It is important to state what exactly we want to measure and how the benchmarks achieve this. It is also important to state what we do *not* want to measure.

- the benchmarks do not aim at comparing sliding with copying collectors: this has been done in the past in several contexts (see e.g. [16])
- the benchmarks do not aim at comparing *policies* of the memory management: we have obviusly made choices about the number of generations, the expansion treshold, the tenuring policy etc.; but this paper does not investigate their impact on overall performance
- the benchmarks do not aim at judging the *allocation* policy of the underlying abstract machine: ilProlog is WAM based, but one deviation from WAM that is important for memory management, is that free variables are always allocated on the heap;
- we do not try to minimize the total runtime (mutator + collector) of a benchmark, instead, we run benchmarks always with the least possible heap size so that no expansion is performed; this stresses the garbage collector most, so that we get more information out of the figures;
- we are not measuring how well the different schemas *scale*; we do not run the same benchmarks with increasing input; instead we have chosen one particular input which gave running times that are reasonably significant
- we do not aim at comparing the implementation of our collector schemas with collectors in other Prolog systems; only occasionally we will mention for a particular program how long the SICStus Prolog garbage collector (3.8.2) or Yap (4.3.19) takes; the SICStus and Yap collectors are based on sliding and not fully generational

So, we are after comparing different implementation choices within the same system: a copying collector in a WAM implementation. In particular, we compare the novel techniques described in [15] with the original copying collector within ilProlog: this collector is non-generational and collapses all segments into one, as described in [4].

Most benchmarks were taken from [12] - the same benchmarks show up in performance tests. These programs are quite small but their input can be set large enough to strain the garbage collector. They usually do not represent real-life programs. We have excluded two benchmarks from this set (queens and rev) because the former uses too few heap cells (queens-24 uses about 356) and the latter has a very peculiar memory behaviour: see appendix A.6.

One benchmark consists of an old version of the ilProlog compiler compiling itself.

Two more benchmarks come from the real-life application in data mining ACE [5]: ilProlog was especially constructed to support such activity. These benchmarks have a more realistic mix of deterministic and non-deterministic computation, require larger heap sizes and have longer running times: ACE and ilProlog deal with applications of data mining that require runs that last several hours on databases of several gigabytes of memory.

Finally, throughout the text and in the appendix, we will use artificial programs to illustrate a particular point.

# 3 The basics of the segment counting method

As explained in [15] we count during the marking phase how many cells survive in each segment. To this purpose, at the start of the collection, a *segment list* is made with an associated counter. This counter is updated during the marking phase. These counts are then used at the beginning of the copying phase for making the new segments in the to-space. More details are in the next sections. The version of the garbage collector that implements this is later in the tables referred to as *new*. It performs always a major collection, i.e. the whole heap is subject to collection, as in the old version of the collector, which we refer to as *oldgc*. Note that the ilProlog compiler generates precise information (in the form of live variable maps) which environment cells are live at each possible continuation point in the program; these maps are used during marking.

## 3.1 Making the segment list

Since we need to count the number of live cells in each segment, we must be able to determine during the marking phase, for each cell that is marked, to determine in which segment the cell resides. Similarly, during the copying phase, we need to know for each cell we copy to which segment in the to-space it must be copied. Thus we need a function to determine to which segment a cell belongs. We will call this function the *segment lookup* function. For each surviving cell we need two segment lookups: one during the marking phase and one during the copying phase. This makes the performance of the lookup function quite important. We implemented the segment lookup function as a binary search in the segment list, which simply contains all segment pointers (pointers in the heap to the start of a segment). The segment list is built by traversing the choice point stack and storing the heap pointer found in each choice point.

## 3.2 Marking

The difference between the marking phase in the old collector and the new one is the use of several counters. We need as many counters as segments. For each marked cell, we augment the counter corresponding with the segment the cell belongs to. These counters then allow us to allocate the new to-space and determine precisely the segments in it.

### 3.3 Copying

In the copying phase we use the Cheney algorithm which copies heap cells from the from-space to the to-space [6]. The original Cheney algorithm uses two pointers *scan* and *next*. The algorithm is easily adapted to copy to segments in the to-space. We simply use one scan and next pointer for each segment in the to-space. These pointers are both initialised to the beginning of their segment. To compute the start of each segment in the to-space we use the list of counters created in the marking phase. After this has been done, we copy each marked cell to the next-pointer of its segment. Copying is finished when all scan pointers are equal to their respective next pointer.

As soon as the size of the new segments is known, we can adjust the heap pointers from the choice points: this is combined with the forwarding of the root set in the choice points. The heap pointer in the choice point is set to the start of the new segment. This is essential for preserving the segments. In [4], all heap pointers from choice points point to the new top of the heap and no segments are preserved. Although segments and their order are preserved, the order of cells within one segment is not retained. Retaining segments after collection is good because it allows instant reclaiming of heap space on backtracking. Respecting segment order is beneficial for generational garbage collection .

### 3.4 Further implementation issues

Earlier, we mentioned that for each live cell, the lookup function is called twice. This is not completely correct: during copying, we follow [4] and forward as a whole contiguous blocks of marked cells. This means that during marking typically for exactly one cell in such a block the lookup function is called. The origin (or destination) segment of other cells is determined by simply checking whether the boundary between the current segment and the next one is crossed. This effect is clearly visible in the tables later, where marking in the segment preserving can become significantly more expensive, accounting for most of the higher garbage collection time.

We mentioned also that the scan pointer of every segment must have caught up with its next pointer for the copying to end. In the segment preserving versions (*new* for instance - see later table 1) it is enough to iterate twice over all segments. In the *pages* version (see section 4.2), pointers which do not appear in the trail, can cross pages and the number of times one must iterate over all pages is not known in advance.

## 4 Approximations to preserving segments

### 4.1 Collapsing segments

The most visible extra cost in the segment counting method (compared to a non segment preserving copying garbage collector) is the segment lookup function. We use a binary search for this, which is $O(log(n))$ with $n$ the number of segments. Since this function is executed at least once for each live heap cell, it might take an important amount of time in the collection cycle.

Having too many segments indeed results in a real slowdown. Not only the lookup function will make performance worse, but also the fact that each segment needs its own scan-next pair of pointers. A solution to this is to merge small adjacent segments into one segment while making the segment list. We call this collapsing segments. We set a minimum segment size and collapse a segment which is smaller than the minimum size with the older segment just next to it. An alternative to setting a minimum size in advance, is to let the minimal allowed size of a segment depend on the total size of the heap and take for instance $heap\_size/64$ as the minimum: this has

a extra benefit that there will be at most 64 segments ever, putting some upper bound on the cost of the lookup function.

The effect of merging segments is that after collection there will be one mixed segment for each set of collapsed segments. This also means that on backtracking over one of the choice points which previously delimited one of the segments in the mixed segment, there will be no instant reclaiming. But if we only collapse small segments, the small amount of heap space we would recover on instant reclaiming might not be worth the speed we gain with a faster lookup function: indeed, when collections take less time, we can afford doing more collections. So, in collapsing segments there is a trade-off between the speedup because of a faster lookup and the instant reclaiming of heap space.

### 4.2 Pages

Another approach to have a faster lookup is the use of pages as an approximation for segments. This has been described in [15]. If we use pages with a size of a power of 2, the lookup function is very simple and indeed $O(1)$, i.e. independent from the number of pages.

It is clear that by using pages, some opportunity for instant reclaiming is lost as in the case of collapsing segments: if the page size is set large, then a page might contain several segments. If the page size is set small, there will be less loss of instant reclaiming, but since in general page boundaries don't match segment boundaries, the loss will be there.

Another important consequence of the fact that a page boundary does not necessarily coincide with a segment boundary, is that such a boundary can also be in the middle of a structure, by which we mean a functor+its arguments. This is different from segments, because in the ilProlog system we know that a structure will be completely in one segment and not spread over two (or more) segments. Regardless of the aspect of segments or not, the order of cells inside a structure determines the meaning of the structure and this order should be kept during copying. In fact we must consider a structure as one block of cells which should be treated as one heap object. If we are about to copy the argument of a functor for instance, we must copy the whole structure at once so as to preserve the layout of structures. Normally this is done by copying the whole continuous block of active cells around the to-copy-cell at once. It should be clear that this works if our to-space doesn't contain several segments. If there are several segments, but each structure block contains cells belonging to the same segment, everything will be copied to the same new segment and still be one continuous block. The problem in the case of pages is that even if we copy the whole block of active cells at once, we will not be sure that the layout will be preserved. Indeed, suppose that the first part of a structure is in page $A$ and the rest in page $A+1$. If this structure is the first object copied to the new page $A$, then after copying there will be a gap between the two parts of the structure, resulting in a corrupt heap. The solution we implemented is to copy the first part of the structure to the end of page $A$ and make sure that the second part of the structure is copied to the beginning of page $A+1$. The copy function checks whether the block of active cells crosses a page boundary. If this is the case, then we lower the scan-next pointers of page $A+1$ in the to-space with the amount of cells in page $A$ that are in the active block. To summarize, we make the new page $A+1$ bigger, so that we can put the first part of the structure in it. Note that this method requires that we try to copy these structures before we start the normal copying phase. If we would already have copied cells to the new page $A+1$, we wouldn't be able to make the new page bigger by lowering the scan-next pointers. This is only possible if the new page $A+1$ is still empty. There is no much extra overhead for handling these special cases. The cost is at most the number of pages times the time needed to check whether a cell is already forwarded.

Using pages seems a good solution because of the constant cost lookup function. Since the cost of the lookup function doesn't depend on the number of pages, nor on the size of pages, we are able to chose the pages as small as we want without changing the cost of the lookup. Using smaller pages reduces the loss on instant reclaiming. Smaller pages will also keep better overall order on the heap in case of large segments. This can benefit locality. One disadvantage of smaller pages is that you need more scan-next pointers. Later we will mention a disadvantage of pages when we try to combine the idea with generations (see section 5.3). The use of pages is implemented in a version that is later referred to as *pages* in the tables. Also here, we have experimented with a fixed page size and one that depends on the size of the heap.

### 4.3 Further implementation issues

For both pages and collapsing segments, all data structures already mentioned in the explanation of the segment counting method can be used in the same way as before. For both the use of pages and the use of collapsing segments, care has to be taken to put the heap pointers in the choice points right. The heap pointer in a choice point should point to just after the end of the new collapsed segment. We can keep on using the algorithm mentioned before because it does this.

## 5 Generations and segments

### 5.1 Basics

Introducing generations in an existing collector seems attractive and at first sight also easy. One famous implementor [7] said recently *I thought that I would just remember H at the moment of garbage collection and at a later garbage collection ignore everything above it.* Things are a wee bit more complicated.

It has been noted already in [2] and later in [4] that the choice point at the moment of a garbage collection can be used as the barrier between the old and the new generation: the trail naturally acts as an exception list and the barrier is in fact a write barrier. This idea can be generalized as follows in two ways:

- at garbage collection time, *any* choice point can be used to delimit the part of the heap that will be collected and the part that is not
- a sequence of choice points can be used to delimit different generations - this observation was made in [15], but might have been made before by others

With this approach, the collected part of the heap always consists of a set of most recent segments on the heap. This is compatible with sliding as well as copying collectors and even with non segment preserving ones like in [4].

The implementation that uses the choice point at the moment of collection as a generational write barrier is referred to as *newgen*.

Using the choice points that arise from normal execution of the program has the advantage of simplicity, but the disadvantage that during a long deterministic computation with garbage collections , no new write barriers are set up. This can even result in a worsening of complexity, exactly like non-generational garbage collection can, as an example in [15] shows. Such behaviour has been observed also in practical programs and benchmarks.

That is why the idea has occurred to several people to push after garbage collection a *generational choice point* whose failure continuation just cuts-fails. This seems like a simple idea, but the

practice is different. In comp.lang.prolog, Joachim Schimpf wrote at some point (not exactly): *we used this generational choice point in ECLiPSe for some time, but it caused too many bugs and it was taken out again.* [15] points out ways to manage the generational choice points. We describe the actual implementation later. We refer to it as *gencp*.

Finally, it is also possible to have the same generational division of the local stack [1]: this saves time during marking and copying - see for instance [2]. In the context of ilProlog, this was not so obvious as free permanent variables are always allocated on the heap and prior to the introduction of generational garbage collection , environment cells never needed trailing. The system that implements this is referred to as *traills* and discussed in more detail in section 5.2.3. All other implemented versions of the garbage collector scan the whole environment stack (during marking and copying). The trade off is efficiency of normal execution and garbage collection .

## 5.2 Extensions and choices

### 5.2.1 Managing generations and generational choice points.

The basic idea is to push a generational choice point after garbage collection : this choice point will naturally act as a write barrier so that the next garbage collection can ignore everything older than that choice point. There are the following problems with this:

1. the generational choice point can disappear due to backtracking or a cut
2. the generational choice point blocks the environment that is current at the moment of the garbage collection , even if - without garbage collection - that environment would have been deallocated by TRO
3. the generational choice point keeps the variables that are in the blocked environment alive
4. when garbage collection occurs frequently - in a long running application for example - there will be many generational choice points and it becomes crucial that they do not introduce complexity problems

In [15] several ways to deal with disappearing choice points were described. We have implemented the following two:

- the collector keeps a list of generational choice points; at each collection, an *appropriate* choice point is searched for so that garbage collection is (hopefully) generational: an appropriate choice point has to meet following conditions :
  - the choice point should be a real choice point, appearing in the linked list of choice points
  - if the first condition is met, we have a valid choice point, but it's not guaranteed to be the same as when it was recorded in the list; to be more accurate on this, we also record the heap pointer found in the choice point in the list of generational choice points; depending on whether the heap pointer found in the choice point is the same, lower or higher than the recorded value, we respectively accept, accept or reject this choice point as a generational choice point; the point here is that we only reject it if accepting it would make the old generation bigger

  normal execution is not affected, i.e. no abstract machine instruction needs to be changed

[1] for the trail, this is obvious

- at cut and at execution of the *trust* instruction (and variants of trust), the list of generational choice points is updated if any generational choice point disappears; this puts a mild overhead on normal execution, which will be measured in section 6.3; we name this *precise_gcp*

The generational choice point blocks environments that belong to otherwise deterministic clauses and keeps variables in these clauses alive. This is a consequence of a naive implementation of the generational choice point idea. The consequences are disastrous to some benchmarks (e.g. tak): the blocked environments result in a much larger local stack [2] and are repeatedly traversed.

Moreover these protected environments will keep extra cells alive and also worsen marking time, because the whole environment stack will be traversed during the marking phase. See appendix A.5 for an example.

However, one can observe that a generational choice point has no forward continuation. This means that the E (and CP) field in a generation choice point are never used during the execution, so it seems correct to also *ignore* the generational choice points during garbage collection. During the marking phase this fits in easily and also during the copying phase, this is possible. However, the environments remain blocked, thereby enlarging the memory footprint of the program. However, since a generational choice point does not need to protect environments, it's sufficient to let it protect the same environments as the previous choice point. This is achieved by putting in the generational choice point as top-of-local-stack pointer the same tops as in the previous choice point. In this way, the check for tail-recursion-optimization in the *deallocate* instruction - which looks at the tops in the current choice point - will not unnecessarily fail.

This solution doesn't introduce any extra problems and is implemented in the final version of the collector. Note that in a traditional WAM with a combined environment and choice point stack, this solution will not work: environments are not only logically but also physically blocked by the generational choice point.

### 5.2.2 Dealing with many generational choice points

Pushing generational choice points can introduce complexity problems when repeated collections make a chain of generational choice points, which delimit very small segments (takgc is an example of such behaviour). The problem with such behaviour is that the choice point stack will grow very large and each garbage collection we traverses it three times (during marking, reversing choice point pointers, copying). Several things can be done to reduce or solve this problem. A first point to note is that there is little use in pushing a generational choice point, if the segment that it will create is smaller than the defined minimum segment size. In the next garbage collection cycle, such a segment would be collapsed with another segment anyway. A second point is that there is no use anymore for a chain of generational choice points (pushed by the garbage collector), on the condition that the segments they delimit already reached tenuring age. One could decide to *update* the generational choice point after collection, instead of pushing a new one. Updating is done by replacing the heap pointer in the choice point. This prevents the building up of a chain of generational choice points. A more complex solution could allow a maximum number of chained pushed generational choice points.

### 5.2.3 Generations in the local stack

As also mentioned in [2] it is possible to treat the environment stack (or combined choice point/environment stack) in a generational way and only scan the most recent environments for root pointers. Of course

[2] ilProlog has a separate environment and choice point stack

to make this work, pointers from environment cells to the heap should be conditionally trailed. This required small changes to the ilProlog system, because formerly, such trailing of environment cells was not necessary. This extra trailing introduces a small overhead on the normal execution mechanism: timings can be found in 6.3.

Dealing with the environment cells in the trail turned out to be more difficult than expected. The problem here is cut or rather the fact that ilProlog does not tidy the trail on cut: this results in two more or less problematic types of pointers in the trail.

1. a cut can have the effect of allowing the later (or immediate) deallocation of an environment that was previously blocked by the now cut away choice point; this can result in pointers from the trail to a location on the local stack that (a) is not in any environment at all, (b) contains the previous E pointer or the continuation pointer of a (in the mean time newly allocated) environment, (c) a newly initialized cell in a new environment

2. more than one trail entry can point to the *some* location in the local stack

A small example shows the latter problem:

```
p :- (f(X), ! ; true), f(X), p.
f(_).
```

The query $? - p.$ results in an unbounded amount of trail and each trail entry points to the same location in the local stack. Since only bindings to a (tagged) pointer to the heap need to be trailed, the ehap and trail will grow at the same rate in this example (since ilProlog allocates variables always on the heap); the local stack in this example has constant size. Still the trail grows unlimited.

To get out of this mess, we saw three solutions:

- tidy the trail on cut: we could implement a tidy trail and that would keep the trail clean; we rejected that solution because of its worst case time performance - see for instance [9]

- just before the first goal in a clause that requires an environment, initialize all environment cells by binding them to a free variable on the heap; this removes the need for trailing of environment cells, since now all bindings happen in the heap; this solution would involve changes in the compiler and affects space and time behaviour of programs not needing garbage collection ; so we also rejected this solution

- a *cautious* approach in which we check during garbage collection whether the environment pointer on the trail is pointing to something valid and in which we duplicate trail entries are removed

We choose for the cautious approach: it is relatively easy to implement - although we got it wrong a couple of times first. Pointers of the kind (a) can be detected simply by a range check and deleted from the trail. Pointers of the kind (b) can point to cell that is not a (tagged) heap pointer and can also be removed. The kind (c) can result in redundant trail pointers, but they do not pose a problem otherwise.

Duplicate entries in the trail are a bit more difficult, because of the way the copying algorithm works and because we do not detect them during the marking phase: the environment cells are treated (as root set) before the trail. Root cells are in principle visited only once during the copying

phase and relocated immediately to the final destination of the cell they point to. So, a root cell never contains a forwarding pointer. We changed that so that after the trail is treated for the first time during the copying phase, environment cells pointed to by trail cells contain a forwarding pointer. In this way, a duplicate trail entry can be recognised as one pointing to a forwarding pointer, and set to null. Since we traverse the trail at this point from young to old, this removes all duplicates except the youngest, which is exactly what is needed. Then the environments are treated - of course taking into account the forwarding pointers they can contain. Afterwards, the trail is scanned from old to new, removing the null entries and compacting the trail, while at the same time changing the forwarding pointers in environments to their final value.

An extra bit for each environment cell, indicating whether it is pointed to by a trail entry or not would have also solved this problem, but our internal representation does not allow for an extra bit in the cell itself, and the environment cells we deal with here not even have a mark bit allocated for them because they belong to the non-collected generation.

### 5.3 Combining pages and generations

While trying to use the pages approach in a generational way we came across a few problems which hadn't been thought of before in [15]. These problems make the pages approach less attractive than originally anticipated.

We already explained the problem with structures crossing page boundaries in a previous section. Although not directly related to that, the problem is also caused by the fact that page boundaries can be anywhere.

When we work generational, every choice point should be able to act as a generational choice point. To be able to garbage collect the young generation, we need to keep track of all pointers from the old generation to the young one. As discussed in the segment approach, we record intersegment pointers on the trail. To know the segment boundaries, we look at the heap pointers in the choice points.

There is a big difference between the pages and the segments approach in what they will do with the heap pointers in the choice points. In the segments approach, after collection (whether segments have been collapsed or not) every choice point's heap pointer will split the heap in two parts so that all cells of a segment are either all above or all under that pointer. This means that we can find in the trail all pointers from the older part to the newer part. This works also with collapsed segments, because these are treated as one big segment. So in the case of a segment preserving garbage collector, the heap pointer in a choice point has the extra meaning of delimiting segments on the heap.

In the case of pages, this assumption is no longer true. A heap pointer in a choice point does not split the heap on (old) segment boundaries. When using pages, the meaning of a heap pointer in a choice point is just a pointer to a place on the heap to which one can safely reclaim heap space upon backtracking over to choice point. The same could be said in the case of a non segment preserving copying collector.

A choice point containing such a heap pointer cannot be used as a generational choice point. The reason is that such a heap pointer might have cells of the next segment under it and above it. So it's possible that a cell from that segment lies under the heap pointer and points to a cell above (newer than) that heap pointer. The trail does not contain a pointer to that cell, because this is in fact an intra-segment pointer. If that cell above the heap pointer is only accessible through that other cell, the marking phase will miss it.

We have not found any satisfying solution for these cross-page pointers. It also seems that

a solution would cause so much additional overhead that it's overall performance would be less than with the segments approach. In practice, it means that generations cannot be combined with pages, or to say the least: a choice point whose H field was adapted during a garbage collection with pages, can in general not serve as a delimiter between generations. In practice this means that if the pushed generational choice point is lost (by backtracking or cut) the next garbage collection will be major.

# 6 Evaluation

All measurements were performed on a Pentium III 866Mhz with 256Mb RAM and timings are given in milliseconds.

## 6.1 Comparing the different options

Table 1 shows some statistics about the simple benchmarks: the figures are given for respectively

- the old garbage collector which does not preserve segments and is non-generational (oldgc)
- the new garbage collector in a mode that preserves segments, but is non-generational (new)
- the new garbage collector in a mode that preserves segments and is generational (newgen)
- the new garbage collector in a mode that preserves segments, is generational and pushes generational choice points (gencp)
- the new garbage collector in a mode that preserves segments, is generational, pushes generational choice points and keeps precise information about the generational choice points (precise_gcp) as described in section 5.2.1
- the new garbage collector in a mode that preserves segments, is generational, pushes generational choice points, keeps precise information about the generational choice points and trails environment variables as described in 5.2.1 and 5.2.3 (precise_gcp-traills)

The statistics are about:

- totalmarked: the total number of marked cells during the run of the program
- heaprecovered: number of heap cells recovered by instant reclaiming
- trailrecovered: number of trail cells recovered during garbage collection
- maxls: high water mark of the environment stack
- maxtr: high water mark of the trail
- maxcp: high water mark of the choice point stack
- nrcollections: total number of collections
- totalmarktime: total time taken by marking
- totalgctime: total time taken by the collections

- totalruntime: total time taken in running the benchmark

The name of each benchmark is followed by the heap size with which this benchmark was run. No expansions were needed during these runs.

Since the granularity of timing is 10 milliseconds, it is clear that a timing of e.g. 90 milliseconds totalgctime for 155 collections is not very meaningful. In such cases, one should look rather at the totalruntime.

Table 2 contains similar figures for the larger benchmarks.

- new recovers more heap space on backtracking than oldgc; however, although that in some benchmarks (boyer, tsp) the difference is noticeable, it doesn't have any influence on totalmarked or nrcollections; a reason for this is that the benchmarks don't do much backtracking
- new is slower than oldgc, mainly because of the segment lookup function; in most benchmarks the totalgctime is about 20% slower, however the benchmarks qsort and serialgc show some strange behaviour: totalgctime is smaller with new than with oldgc; we can't find a profound explanation for this, but it seems that cache effects play a role here
- newgen is mostly slower than new; most of the benchmarks are deterministic and only have a choice point near the start of the heap, the benefit of newgen here is not-collecting that small part (it will become the old generation); this gain isn't enough to justify the overhead of keeping the generation list; because of this most benchmarks have a higher totalgctime with newgen, only boyer does really benefit from the generational approach which can be seen in the difference of totalmarked
- gencp (and also precise_gcp and precise_gcptraills) have in some cases more collections than the other collectors; the reason herefore is that in deterministic applications other collectors will be forced to collect the whole 'deterministic' segment, while gencp will artificially split up the segment in smaller parts and only collect the most recent parts; by doing this gencp will possibly keep more cells alive (it has a bigger old generation) and this results in more collections
- gencp and precise_gcp won't necessarily have the same amount of collections because they use different choice points as generational choice points; precise_gcp will move its generational choice point on backtracking or cut, while gencp will jump back to the previously recorded generational choice point; as a result precise_gcp will have a bigger old generation; after a few collections some cells in these segments will die; in the case of precise_gcp these won't be cleaned up because they sit in the old generation, but they will be cleaned up with gencp; then gencp has more free space in its heap will have fewer collections
- gencp (and also precise_gcp and precise_gcptraills) will have a bigger maxcp; this is expected because they push a new choice point on the stack after each collection; on deterministic benchmarks with many collections (tak, tsp) however this can blow up the use of choice point stack space; we also expect the trail usage to be higher because of the higher number of choice points; in qsort and serialgc though, maxtr is lower, this is something we can't explain
- precise_gcptraills uses more trail space than precise_gcp, but it also recovers more trail space; this can be seen in boyer and all realistic benchmarks; the most remarkable here is muta_model_1: while its trailrecovered almost doubles, the amount of maxtr becomes only slightly bigger; this is probably because there are many duplicates in the trailed environment cells

| benchmark | what | oldgc | new | newgen | gencp | precise_gcp | precise_gcp traills |
|---|---|---|---|---|---|---|---|
| boyer | totalmarked | 52079509 | 52079509 | 38870130 | 52079509 | 29194142 | 29216960 |
| 6M | heaprecovered | 7260877 | 11334108 | 10856703 | 11334123 | 10856703 | 10877938 |
| | trailrecovered | 10791464 | 10791747 | 10869262 | 10791753 | 10869278 | 11078459 |
| | maxls | 459 | 459 | 459 | 459 | 459 | 459 |
| | maxtr | 813310 | 813306 | 813306 | 813306 | 813306 | 827890 |
| | maxcp | 235 | 235 | 235 | 235 | 242 | 235 |
| | nrcollections | 20 | 20 | 21 | 20 | 21 | 21 |
| | totalmarktime | 3470 | 4190 | 3220 | 4220 | 2590 | 2890 |
| | totalgctime | 10440 | 12060 | 9330 | 11990 | 7350 | 7510 |
| | totalruntime | 26660 | 28320 | 25550 | 28440 | 24460 | 24450 |
| browse | totalmarked | 1400475 | 1400475 | 1400461 | 850630 | 850630 | 850630 |
| 500K | heaprecovered | 14485089 | 14765177 | 14765192 | 14770942 | 14770942 | 14770942 |
| | trailrecovered | 24044 | 24048 | 24048 | 24026 | 24026 | 24026 |
| | maxls | 60 | 60 | 60 | 60 | 60 | 60 |
| | maxtr | 16805 | 16805 | 16805 | 16805 | 16805 | 16805 |
| | maxcp | 86 | 86 | 86 | 96 | 96 | 96 |
| | nrcollections | 5 | 5 | 5 | 5 | 5 | 5 |
| | totalmarktime | 60 | 110 | 120 | 70 | 80 | 60 |
| | totalgctime | 260 | 320 | 350 | 220 | 200 | 200 |
| | totalruntime | 9020 | 9060 | 9140 | 9020 | 9250 | 9440 |
| dnamatch | totalmarked | 4563750 | 4563750 | 4562882 | 277188 | 277188 | 277188 |
| 250K | heaprecovered | 179723 | 181477 | 181492 | 191484 | 191484 | 191484 |
| | trailrecovered | 103312 | 102667 | 102667 | 103281 | 103281 | 103281 |
| | maxls | 67 | 67 | 67 | 67 | 67 | 67 |
| | maxtr | 1549 | 1549 | 1549 | 1549 | 1549 | 1549 |
| | maxcp | 68 | 68 | 68 | 1145 | 1145 | 1145 |
| | nrcollections | 127 | 127 | 127 | 155 | 155 | 155 |
| | totalmarktime | 210 | 370 | 440 | 40 | 0 | 20 |
| | totalgctime | 680 | 860 | 960 | 70 | 30 | 80 |
| | totalruntime | 6180 | 6320 | 6390 | 5610 | 5770 | 5970 |
| qsort | totalmarked | 5685001 | 5685001 | 5684861 | 2741869 | 2741869 | 2741869 |
| 500K | heaprecovered | 1836037 | 2134925 | 2134940 | 2199177 | 2199177 | 2199177 |
| | trailrecovered | 1738706 | 1738685 | 1738685 | 1729324 | 1729324 | 1729324 |
| | maxls | 183 | 183 | 183 | 183 | 183 | 183 |
| | maxtr | 99373 | 99355 | 99355 | 94599 | 94599 | 94599 |
| | maxcp | 68 | 68 | 68 | 236 | 236 | 236 |
| | nrcollections | 23 | 23 | 23 | 25 | 25 | 25 |
| | totalmarktime | 950 | 530 | 550 | 310 | 280 | 260 |
| | totalgctime | 1640 | 1400 | 1420 | 720 | 680 | 710 |
| | totalruntime | 2720 | 2490 | 2490 | 1750 | 1790 | 1830 |
| serialgc | totalmarked | 23520742 | 23520734 | 23520713 | 19409637 | 19409637 | 19409637 |
| 10M | heaprecovered | 16932749 | 17357475 | 17357490 | 23177546 | 23177546 | 23177546 |
| | trailrecovered | 6537512 | 6537562 | 6537562 | 5117693 | 5117693 | 5117693 |
| | maxls | 118 | 118 | 118 | 118 | 118 | 118 |
| | maxtr | 2295067 | 2295001 | 2295001 | 2092710 | 2092710 | 2092710 |
| | maxcp | 68 | 68 | 68 | 75 | 75 | 75 |
| | nrcollections | 6 | 6 | 6 | 6 | 6 | 6 |
| | totalmarktime | 3160 | 2500 | 2520 | 2170 | 2240 | 2210 |
| | totalgctime | 6370 | 6100 | 6130 | 5690 | 5710 | 5700 |
| | totalruntime | 24210 | 23890 | 24020 | 23740 | 24450 | 24350 |
| tak | totalmarked | 58633 | 58633 | 49995 | 74702 | 74702 | 74702 |
| 10K | heaprecovered | 1144 | 1172 | 1187 | 5939 | 5939 | 5939 |
| | trailrecovered | 16945 | 0 | 0 | 23149 | 23149 | 23149 |
| | maxls | 256 | 256 | 256 | 256 | 256 | 256 |
| | maxtr | 35 | 2 | 2 | 57 | 57 | 57 |
| | maxcp | 68 | 68 | 68 | 12493 | 12493 | 12493 |
| | nrcollections | 1237 | 1237 | 1237 | 1776 | 1776 | 1776 |
| | totalmarktime | 90 | 20 | 20 | 50 | 110 | 20 |
| | totalgctime | 120 | 70 | 60 | 160 | 220 | 80 |
| | totalruntime | 3590 | 3600 | 3610 | 3750 | 3790 | 3800 |
| tspgc | totalmarked | 28389147 | 28389147 | 28380705 | 670143 | 670143 | 670143 |
| 250K | heaprecovered | 8517 | 31983 | 31998 | 199889 | 199889 | 199889 |
| | trailrecovered | 52278 | 45296 | 45296 | 50868 | 50868 | 50868 |
| | maxls | 75 | 75 | 75 | 75 | 75 | 75 |
| | maxtr | 204 | 204 | 204 | 2844 | 2844 | 2844 |
| | maxcp | 68 | 68 | 68 | 10202 | 10202 | 10202 |
| | nrcollections | 1209 | 1209 | 1209 | 1448 | 1448 | 1448 |
| | totalmarktime | 1430 | 2070 | 2130 | 460 | 640 | 80 |
| | totalgctime | 3230 | 4130 | 4170 | 690 | 850 | 250 |
| | totalruntime | 65800 | 67240 | 67700 | 63990 | 63770 | 65900 |

Table 1: Comparing garbage collections on smallish benchmarks

| benchmark | what | oldgc | new | newgen | gencp | precise_gcp | precise_gcp traills |
|---|---|---|---|---|---|---|---|
| comp 260K | totalmarked | 425145 | 425145 | 425145 | 425145 | 425094 | 425104 |
| | heaprecovered | 3423966 | 3536358 | 3536373 | 3536373 | 3536373 | 3536386 |
| | trailrecovered | 212055 | 212102 | 212102 | 212102 | 212102 | 224939 |
| | maxls | 6255 | 6255 | 6255 | 6255 | 6255 | 6255 |
| | maxtr | 44796 | 44796 | 44796 | 44796 | 44796 | 47687 |
| | maxcp | 5917 | 5917 | 5917 | 5917 | 5917 | 5917 |
| | nrcollections | 6 | 6 | 6 | 6 | 6 | 6 |
| | totalmarktime | 40 | 70 | 60 | 50 | 60 | 40 |
| | totalgctime | 110 | 130 | 120 | 120 | 120 | 130 |
| | totalruntime | 2540 | 2560 | 2570 | 2510 | 2570 | 2710 |
| muta_model_1 300K | totalmarked | 3957027 | 3940411 | 3880039 | 3904746 | 3625855 | 3544412 |
| | heaprecovered | 277587743 | 277627758 | 277611466 | 277606016 | 277683262 | 277590406 |
| | trailrecovered | 193439 | 193455 | 193360 | 193816 | 193166 | 356861 |
| | maxls | 3925 | 3925 | 3925 | 3925 | 3925 | 3925 |
| | maxtr | 18560 | 18413 | 18413 | 18417 | 18429 | 18938 |
| | maxcp | 1636 | 1636 | 1636 | 1636 | 1636 | 1636 |
| | nrcollections | 118 | 118 | 120 | 118 | 124 | 125 |
| | totalmarktime | 190 | 370 | 350 | 320 | 290 | 440 |
| | totalgctime | 650 | 890 | 950 | 830 | 790 | 830 |
| | totalruntime | 57670 | 59220 | 59400 | 57770 | 59250 | 60930 |
| muta_nomodel_1 5M | totalmarked | 370409112 | 370421395 | 372673278 | 370421395 | 374100790 | 370783244 |
| | heaprecovered | 325879386 | 325879872 | 325877716 | 325879884 | 325879850 | 325880563 |
| | trailrecovered | 33768107 | 33765016 | 33765022 | 33766804 | 33788773 | 34596489 |
| | maxls | 9295 | 9295 | 9295 | 9295 | 9295 | 9295 |
| | maxtr | 828300 | 828284 | 828284 | 828284 | 828284 | 828364 |
| | maxcp | 7495 | 7495 | 7495 | 7502 | 7495 | 7495 |
| | nrcollections | 239 | 239 | 240 | 239 | 241 | 241 |
| | totalmarktime | 22120 | 35640 | 35930 | 35630 | 36100 | 35920 |
| | totalgctime | 76940 | 94880 | 97050 | 95020 | 95530 | 96700 |
| | totalruntime | 696440 | 716630 | 719700 | 715840 | 712720 | 728340 |

Table 2: Comparing garbage collections on more realistic benchmarks

- gencp and precise_gcp clearly offer the best performance on the smallish benchmarks; on the realistic benchmarks they're still the best segment preserving collectors, but oldgc is overall better there; especially for muta_nomodel_1 oldgc performs remarkably better, a better policy for the generational collector might help here: precise_gcp which has the second best totalruntime, actually marks more cells than oldgc does; the reason for this is that it keeps cells alive by never doing a major collection; another remark about muta_nomodel_1 is that it doesn't benefit much from a generational approach, the old generation is all times only 1% of the total heap size;

- in some cases precise_gcp is slower than gencp in totalruntime; mostly when this is the case, totalmarked will be the same for the two; the reason for the difference in totalruntime is the runtime overhead with precise_gcp

- there are few benchmarks which really benefit from precise_gcptraills; mostly totalruntime is a little slower than with precise_gcp; the causes for this is that (a) (like already mentioned) there are many duplicates on the trail and while they will be removed out of the trail, they still cause overhead in at least one collection cycle and (b) the trailing of environment cells causes an additional runtime overhead; it seems that the win of traills isn't enough to win back its cost

### 6.2 Comparing different ways of collapsing/approximating segments

Tables 3 and 4 show the effect of different approximations to segments. In the columns, *newgc none* means that no segments are merged (this amounts to setting the minimally allowed segment size to 1); *newgc 8K* means the minimally allowed segment size is 8K cells; *newgc H/64* sets it to the heap size divided by 64. A similar explanation is valid for *pages*, but $H/64$ is always rounded to the nearest power of 2.

The figures show some performance advantage for the pages, but the number of segments in almost all benchmarks is very low. This seems to be inherent in the way Prolog programs are written: backtracking is usually not deep and choice points are discarded quickly.

### 6.3 The overhead of precise generational choice point maintainance and of environment cell trailing

At the start, one of our guiding principles was that no changes should be made to our WAM, and that execution of programs that do not need garbage collection should not be slowed down. Some of the implemented alternatives do not adhere to that principle, in particular the precise generational choice point maintenance and the introduction of environment cell trailing. So it is worthwhile measuring the overhead of each of these mechanisms independently of garbage collection .

Note first that precise generational choice point maintenance does not affect space, and adds a cost to the total of the execution which is linear in the number of pushed generational choice points: complexity is not affected. The cost of environment cell trailing is a constant factor for some instructions (getpvar for instance), but has an unbounded worst case space behaviour in the absence of tidy trail at cut as was shown by the example in 5.2.3.

Table 5 shows for our set of benchmarks the overhead of both mechanisms.

The overhead seems small and acceptable for precise generational choice point maintenance. Trailing local stack variables has a slightly higher cost.

## 7 Policy

As mentioned before, the aim of the paper was not to evaluate the policy of the memory manager. Still, some choices had to be made in the implementation. Here is a short overview of them without any justification.

- our tenuring policy is that a cell moves to the older generation after it has survived two garbage collections
- we allow at most 300 entries in the generation list: when this overflows, we merge every two adjacent entries into one, so that we keep 150
- a major collection is performed when the marking of the minor collection detects that not more than 30% of the heap will be free after the minor collection
- the default for merging segments is that segments are merged (newgc H/64 in tables 3 and 4), wiyth the minimum segment size depending on the heap size

Also, the main underlying principle in the implementation of generations with pushing a generational choice point has been that at **every** collection, such a choice point would be pushed and

| benchmark<br>benchmark | what | newgc<br>none | newgc<br>8K | newgc<br>H/64 | pages<br>8K | pages<br>H/64 |
|---|---|---|---|---|---|---|
| boyer | totalmarked | 52079514 | 52079509 | 52079509 | 52079509 | 52079509 |
| 6M | heaprecovered | 11334130 | 11334120 | 11334108 | 11325938 | 11268579 |
| | trailrecovered | 10791547 | 10791610 | 10791747 | 10791652 | 10791725 |
| | maxls | 459 | 459 | 459 | 459 | 459 |
| | maxtr | 813296 | 813304 | 813306 | 813310 | 813310 |
| | maxcp | 235 | 235 | 235 | 235 | 235 |
| | nrcollections | 20 | 20 | 20 | 20 | 20 |
| | totalmarktime | 4450 | 4420 | 4330 | 3440 | 3510 |
| | totalgctime | 12370 | 12530 | 12120 | 10880 | 10820 |
| | totalruntime | 28560 | 29010 | 28540 | 27270 | 27340 |
| browse | totalmarked | 1400475 | 1400475 | 1400475 | 1400475 | 1400475 |
| 500K | heaprecovered | 14765196 | 14765189 | 14765177 | 14757007 | 14761088 |
| | trailrecovered | 24048 | 24048 | 24048 | 24048 | 24048 |
| | maxls | 60 | 60 | 60 | 60 | 60 |
| | maxtr | 16805 | 16805 | 16805 | 16805 | 16805 |
| | maxcp | 86 | 86 | 86 | 86 | 86 |
| | nrcollections | 5 | 5 | 5 | 5 | 5 |
| | totalmarktime | 130 | 140 | 350 | 90 | 80 |
| | totalgctime | 330 | 320 | 570 | 300 | 270 |
| | totalruntime | 9170 | 9110 | 9350 | 9080 | 9040 |
| dnamatch | totalmarked | 4563750 | 4563750 | 4563750 | 4563750 | 4563750 |
| 250K | heaprecovered | 181496 | 181489 | 181477 | 179976 | 180030 |
| | trailrecovered | 102667 | 102667 | 102667 | 102667 | 102667 |
| | maxls | 67 | 67 | 67 | 67 | 67 |
| | maxtr | 1549 | 1549 | 1549 | 1549 | 1549 |
| | maxcp | 68 | 68 | 68 | 68 | 68 |
| | nrcollections | 127 | 127 | 127 | 127 | 127 |
| | totalmarktime | 390 | 390 | 290 | 180 | 190 |
| | totalgctime | 830 | 820 | 780 | 680 | 710 |
| | totalruntime | 6270 | 6310 | 6360 | 6170 | 6180 |
| qsort | totalmarked | 5685001 | 5685001 | 5685001 | 5685001 | 5685001 |
| 500K | heaprecovered | 2134944 | 2134937 | 2134925 | 2130648 | 2134729 |
| | trailrecovered | 1738685 | 1738685 | 1738685 | 1738685 | 1738685 |
| | maxls | 183 | 183 | 183 | 183 | 183 |
| | maxtr | 99355 | 99355 | 99355 | 99355 | 99355 |
| | maxcp | 68 | 68 | 68 | 68 | 68 |
| | nrcollections | 23 | 23 | 23 | 23 | 23 |
| | totalmarktime | 550 | 510 | 550 | 290 | 310 |
| | totalgctime | 1340 | 1390 | 1360 | 1190 | 1210 |
| | totalruntime | 2440 | 2480 | 2430 | 2280 | 2270 |
| serialgc | totalmarked | 23520734 | 23520734 | 23520734 | 23520486 | 23521442 |
| 10M | heaprecovered | 17357494 | 17357487 | 17357475 | 17349057 | 17324216 |
| | trailrecovered | 6537562 | 6537562 | 6537562 | 6537606 | 6537541 |
| | maxls | 118 | 118 | 118 | 118 | 118 |
| | maxtr | 2295001 | 2295001 | 2295001 | 2294994 | 2295143 |
| | maxcp | 68 | 68 | 68 | 68 | 68 |
| | nrcollections | 6 | 6 | 6 | 6 | 6 |
| | totalmarktime | 2530 | 2500 | 2500 | 1830 | 1890 |
| | totalgctime | 6140 | 6070 | 6070 | 5600 | 5560 |
| | totalruntime | 23980 | 23900 | 23920 | 23270 | 23320 |
| tak | totalmarked | 58633 | 58633 | 58633 | 58633 | 58633 |
| 10K | heaprecovered | 1191 | 1156 | 1172 | 1159 | 1164 |
| | trailrecovered | 0 | 16945 | 0 | 16945 | 1393 |
| | maxls | 256 | 256 | 256 | 256 | 256 |
| | maxtr | 2 | 35 | 2 | 35 | 17 |
| | maxcp | 68 | 68 | 68 | 68 | 68 |
| | nrcollections | 1237 | 1237 | 1237 | 1237 | 1237 |
| | totalmarktime | 20 | 30 | 80 | 20 | 10 |
| | totalgctime | 50 | 50 | 100 | 70 | 70 |
| | totalruntime | 3610 | 3600 | 3610 | 3540 | 3570 |
| tspgc | totalmarked | 28389147 | 28389147 | 28389147 | 28389147 | 28389147 |
| 250K | heaprecovered | 32002 | 31995 | 31983 | 23813 | 29942 |
| | trailrecovered | 45296 | 45296 | 45296 | 45296 | 45296 |
| | maxls | 75 | 75 | 75 | 75 | 75 |
| | maxtr | 204 | 204 | 204 | 204 | 204 |
| | maxcp | 68 | 68 | 68 | 68 | 68 |
| | nrcollections | 1209 | 1209 | 1209 | 1209 | 1209 |
| | totalmarktime | 2110 | 1930 | 2060 | 1370 | 1450 |
| | totalgctime | 4230 | 4090 | 4050 | 3280 | 3810 |
| | totalruntime | 67590 | 67400 | 67260 | 66640 | 67030 |

Table 3: Collapsing and approximation of segments for smallish benchmarks

| benchmark<br>benchmark | what | newgc<br>none | newgc<br>8K | newgc<br>H/64 | pages<br>8K | pages<br>H/64 |
|---|---|---|---|---|---|---|
| comp | totalmarked | 425145 | 425145 | 425145 | 425145 | 425145 |
| 260K | heaprecovered | 3536389 | 3536370 | 3536358 | 3528198 | 3534327 |
| | trailrecovered | 212090 | 212102 | 212102 | 212099 | 212092 |
| | maxls | 6255 | 6255 | 6255 | 6255 | 6255 |
| | maxtr | 44796 | 44796 | 44796 | 44796 | 44796 |
| | maxcp | 5917 | 5917 | 5917 | 5917 | 5917 |
| | nrcollections | 6 | 6 | 6 | 6 | 6 |
| | totalmarktime | 70 | 60 | 30 | 50 | 20 |
| | totalgctime | 150 | 140 | 100 | 120 | 90 |
| | totalruntime | 2550 | 2560 | 2550 | 2540 | 2540 |
| muta_model_1 | totalmarked | 3939492 | 3940411 | 3940411 | 3957868 | 3939356 |
| 300K | heaprecovered | 277609440 | 277622298 | 277627758 | 277545957 | 277569532 |
| | trailrecovered | 193298 | 193487 | 193455 | 193571 | 193483 |
| | maxls | 3925 | 3925 | 3925 | 3925 | 3925 |
| | maxtr | 18413 | 18421 | 18413 | 18448 | 18426 |
| | maxcp | 1636 | 1636 | 1636 | 1636 | 1636 |
| | nrcollections | 118 | 118 | 118 | 118 | 118 |
| | totalmarktime | 400 | 330 | 360 | 240 | 310 |
| | totalgctime | 870 | 830 | 880 | 800 | 830 |
| | totalruntime | 59610 | 59220 | 59300 | 58720 | 58010 |
| muta_nomodel_1 | totalmarked | 370418927 | 370421395 | 370421395 | 370406992 | 370406489 |
| 5M | heaprecovered | 325880698 | 325879884 | 325879872 | 325879362 | 325879589 |
| | trailrecovered | 33764946 | 33765015 | 33765016 | 33764925 | 33765010 |
| | maxls | 9295 | 9295 | 9295 | 9295 | 9295 |
| | maxtr | 828276 | 828284 | 828284 | 828297 | 828297 |
| | maxcp | 7495 | 7495 | 7495 | 7495 | 7495 |
| | nrcollections | 239 | 239 | 239 | 239 | 239 |
| | totalmarktime | 42140 | 35500 | 38650 | 27860 | 27740 |
| | totalgctime | 101830 | 95110 | 99120 | 85840 | 85640 |
| | totalruntime | 726800 | 718630 | 1951710 | 709410 | 711320 |

Table 4: Collapsing and approximation of segments for realistic benchmarks

that even on a major collections, we keep the generations. This leads to anomalies in some programs: tak is a good example of that. We have described some techniques for keeping the number of generational choice points low, but have not fully experimented with them.

The benchmarking is not really influenced by some of the above decisions: if there are too few generational choice points, the strategy for dealing with too many such choice points is immaterial, and major collections occur rarely during the benchmarks.

# 8 Conclusion and future work

As a compromise between the efficiency as shown by the benchmarks and the convenience of implementation, we favour the following choices in the implementation:

| benchmark | newgc | precise_gcp | traills |
|---|---|---|---|
| boyer | 2000 | 2010 | 2170 |
| browse | 1550 | 1560 | 1590 |
| dnamatch | 2190 | 2150 | 2340 |
| qsort | 1060 | 1050 | 1060 |
| serialgc | 1800 | 1830 | 1880 |
| tak | 3290 | 3360 | 3490 |
| tspgc | 3080 | 3170 | 3300 |
| comp | 2200 | 2200 | 2290 |

Table 5: The overhead of dealing with retracted generational choice points and environment cell trailing

- keep information about multiple generations as precise as possible, i.e. *precise_gcp*
- merge segments and do not make very small new generations
- avoid large number of generational choice points - possibly a clean up of the choice point stack must be considered or a different architecture which allows for an extra generational choice point stack
- no environment cell trailing

The latter requires a bit more explanation: the worst case of scanning the complete environment stack on every garbage collection is as bad as doing non-generational garbage collection . However, in practice it turns out that local stack consumption is not that large: maintaining TRO is more important.

There are many issues that we have not touched in this study and that are worth exploring:

- an adaptive policy which changes according to the observed occupancy or fragmentation - see for instance [17]
- using a policy that takes into account the cache - see for instance [19]
- adopting some form of the policy of [12], which moves the heap limit during the computation

We intend to design and implement a garbage collector for GNU-Prolog [10]: in that context, we will also have to deal with a value trail. In ilProlog, we had to deal with a non-backtrackable form of *setarg*/3 already: this gives rise to heap cells that are trailed multiple times and they are dealt with in essentially the same way as multiple trailed local stack cells (see 5.2.3)). The issue of a value trail has been dealt with in other contexts as well and no special problems are anticipated when the collector is truly generational.

Finally, in principle we have implemented everything needed to deal with any number of generations. In practice, we deal only with two generations: when a minor collection does not free enough space (this is actually detected after a marking phase already), our policy is to do a major collection. We could instead attempt to collect a larger part of the heap but not the complete heap. However, the larger applications (ACE) indicate that older generations (i.e. sets of segments that have survived several collections) are very small. If this is true for the typical application written in Prolog, there is little point in maintaining multiple generations. Also here, adaptive strategies should be investigated.

### Acknowledgements

We are grateful to the machine learning team of the department of computer science of the K.U.Leuven for providing us the ACE benchmarks.

## References

[1] Hassan Aït-Kaci. *Warren's Abstract Machine: A Tutorial Reconstruction.* The MIT Press, Cambridge, Massachusetts, 1991. See also: http://www.isg.sfu.ca/~hak/documents/wam.html.

[2] 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.

[3] 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.

[4] 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.

[5] H. Blockeel, L. Dehaspe, B. Demoen, G. Janssens, J. Ramon, and H. Vandecasteele. Executing query packs in ILP. Inductive Logic Programming, 10th International Conference, ILP2000, London, UK, July 2000, Proceedings (J. Cussens and A. Frisch, eds.), Lecture Notes in Artificial Intelligence, vol. 1866, Springer, 2000, pp. 60-77.

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

[7] Vitor S. Costa Private communication.

[8] 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.

[9] B. Demoen and K. Sagonas. CHAT is Θ(SLG-WAM). Proceedings of the 6th International Conference on Logic for Programming and Automated Reasoning; pp. 337–357; Sept. 1999, Tbilisi, Georgia

[10] D. Diaz and P. Codognet. *GNU Prolog: beyond compiling Prolog to C* Proceedings of the Second International Workshop, PADL 2000, Boston, MA, USA, January 2000. LNCS 1753, pp. 81-92 See also http://gprolog.inria.fr

[11] 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.

[12] 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.

[13] 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.

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

[15] K. Sagonas and B. Demoen. From (multi-)generational to segment order preserving copying garbage collection for the WAM. K.U.Leuven, CW report 303, October 2000

[16] P. M. Sansom. Combining copying and compacting garbage collection or Dual-mode garbage collection. In R. Heldal, C. K. Holst, and P. Wadler, editors, *Functional Programming*, Workshops in Computing, Glasgow, Aug. 1991. Springer-Verlag.

[17] Yoshikawa Takahide and Chikayama Takashi. An Adaptive Generational GC Scheme that Dynamically Adjusts the Young Generation Size Preprint

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

[19] Paul R. Wilson, Michael S. Lam, Thomas G. Moher. Caching considerations for generational garbage collection. Proceedings of the 1992 ACM Conference on Lisp and Functional Programming, San Francisco, California, pp. 32-42, June 1992.

# A Some extreme examples

Extreme examples serve the purpose of showing a particular feature of algorithms.

## A.1 Deterministic execution and garbage collection .

This example is meant to show clearly the problem of non-generational garbage collection for deterministic programs: in program below, the list L2 is constructed incrementally and a non-generational collector will marked and copy repeatedly all parts, leading in fact to a quadratic algorithm, instead of a linear. With generations, only the last increment of L2 is marked and collected at any collection.

```
go(N) :-
        mklist(N,L1,[]),
        b(L1,L2),
        use(L1-L2).

b([_|R],L) :-
        mklist(10000,L,S),
        garbage_collect,
        b(R,S).

use(_).

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

The query ? – *go*(100). yields the timings (only garbage collection ) reproduced in table 6

| oldgc | gencp | SICStus | Yap |
|---|---|---|---|
| 34.560 | 800 | 23.330 | 29.880 |

Table 6: True generational version non-generational garbage collection in deterministic code

Note that the collectors in SICStus and Yap are of the sliding type and that there is no garbage: this is the reason why sliding beats the non-generational copying collector in oldgc.

## A.2 Quasi-deterministic execution and garbage collection

In Prolog, execution can become deterministic by a cut: programmers have different styles in using the cut, which can also be hidden in an if-then-else construct. The following program builds a list of 100K integers and then repeatedly collects it: the program does have a choice point which is cut away either early or late. The reason for having this program is that a similar situation occurs in the benchmark *boyer*.

```
go(N) :-
        mklist(100000,L1),
        mklist(N,L2),
        a(L2),
        use(L1-L2).

%% this version of a/1 has a late cut
a([_|R]) :-
        (x(R), a(R) ; true), !.

%% alternative definition of a/1 with an early cut
%% a([_|R]) :-
%%         (x(R), !, a(R) ; true).

x([_|_]) :- garbage_collect.

use(_).

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

The query ? – $go(100)$ results in the figures in table 7

| | gencp | SICStus | Yap |
|---|---|---|---|
| late cut | 120 | 4.430 | 5.870 |
| early cut | 4.240 | 4.380 | 5.870 |

Table 7: Effect of cutting away generational choice points

Note that this shows a shortcoming in the way we are dealing with cut away generational choice points.

### A.3 Recovery on backtracking

It is clear that instant reclaiming has an overal beneficial effect, but it also affects garbage collection itself; although it costs time to retain the order of the segments for our copying collector, instant reclaiming pays for it more than this cost. The following program shows that - note that it will not benefit from pushing generational choice points, as the point of garbage collection is always backtracked over.

```
go :-
        mklist(650000,L),
        cp(L).

cp(_) :-
        mklist(320000,_),
        mklist(80000,L2),
        use(L2),
        fail.
cp(L) :-
        mklist(340000,L3),
        use(L3),
        fail.
use(_).

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

The query ? – *go* yields the following timings (only garbage collection ) and amount of heap recovered by instant reclaiming (in heap cells). Both SICStus 3.8.6b and Yap 4.3.0use a sliding collector and retain the full capacity of instant reclaiming.

| | oldgc | new | SICStus | Yap |
|---|---|---|---|---|
| gctime | 370 | 260 | 250 | 390 |
| recovered | 142147 | 2140138 | | |

Table 8: Effect of recovery on backtracking on garbage collection time

## A.4 Generations and environments

When running a system with a generational collector, one can also impose the same generation structure on the environment stack as indicated in section 5.2.3. In the context of ilProlog, this means that environment cells also need (conditional) trailing. One can also consider the whole local stack as belonging to the generation to be collected. The following is an artificial example showing the effect of choosing either: first 1M environments are created - without any local variable - and then 100 garbage collections are triggered.

```
go :-
        mkenv(1000000).

mkenv(N) :-
        (N == 0 ->
            choice,
            go(100)
        ;
            M is N - 1,
            mkenv(M),
            tail
        ).

choice.                 % a pity we need this choice point
choice :- fail.         % the result of tos/e in gencp

tail.

go(N) :-
        N > 0,
        garbage_collect,
        M is N - 1,
        go(M).
```

Table 9 shows the timings on different versions of our collectors and on SICStus and Yap.

| | oldgc | traills | SICStus | Yap |
|---|---|---|---|---|
| gctime | 9.970 | 310 | 25.640 | 14.910 |

Table 9: Effect of avoiding to traverse all environments

## A.5 Naive generational choice points can keep dead objects alive

Consider the following deterministic and tail recursive predicate:

```
p :- read(X), X \== end_of_file, write(X), p.
```

The structure of this predicate is typical for many applications. When there is enough input to read, eventually a garbage collection will taken place during the call to *read*/1. For simplicity, we rewrite the clause so as to make the garbage collection and the pushing of the generational choice point explicit.

```
p :- read(X), gencp, X \== end_of_file, write(X), p.

gencp :- garbage_collect.
gencp :- fail.
```

It is clear that only the term just read is useful. The generation choice point has now as effect that a choice point is pushed, which keeps the environment of $p/0$ alive and which also keeps alive the read term that would have become garbage after it is written out.

We had not anticipated this side effect of generational choice points. We have dealt with it as described in section 5.2.1.

However, the issues are different in different versions of the collector: first of all, in no case should the E-CP fields in a generational choice point pushed by the collector be used as a starting point for marking. Further,

- in the version that traverses the whole local stack on garbage collection , just putting the tos field of the generational choice point to the tos of the previous choice point will allow deallocation in deterministic mode
- in the version *traills* there is another issue: the tos field in a choice point is used not only for deallocation of environments, but also for conditional (environment cell) trailing; this means that environment cells younger than this tos might not get trailed, so these environments must be traversed during garbage collection ; this means that also the E field of the genrational choice point must be set to the same value as the tos: that is the reason we needed the extra choice point in A.4
-

There seems to be some inherent problem here: the price for not being able to deallocate environments is having to traverse them during garbage collection . Even in a system that has environment cell trailing from the start, this problem occurs. This requires further investigation.

## A.6 Why reverse/2 is a bad garbage collection benchmark

The reverse/2 predicate basically works like a copylist/2 predicate and it is slightly easier to explain the reasoning on copylist/2.

```
copylist([],[]).
copylist([X|R],[X|S]) :- copylist(R,S).
```

called as in $? - mklist(L), copylist(L, NewL)$. where only NewL is non-garbage at the end of the computation.

From the point of view memory management it can be rewritten as:

```
copylist([],[]).
copylist(In,Out) :-
        In = [X|R],
        forgetlocations(In,X,R),      % the values of X and R are in registers now
                                      % and In is no longer needed
        allocatelocations(Out,X,S),
        Out = [X|S],
        copylist(R,S).
```

It is clear that the size of the useful data is constant and equal to the size of the input list. Also, the oldest cells of the input list become garbage while newer list cells in the output become alive. Clearly, at every non-generational garbage collection , as much data as that size needs to be touched. Now suppose the heap size is just large enough to contain $size+1$ cells, then the number if garbage collections will be in the order of $size$ and we get a quadratic process. Any non-generational schema suffers from this problem. And also in a generational schema, the problem remains the same, as long as the policy is to have a fixed heap size and only perform garbage collection when that heap is full, i.e. when no more cells can be allocated at the top of the heap. For this type of program, a better policy is offered by [12], which (simplified for deterministic computations) consists in

- assuming a virtually unlimited heap
- performing a generational garbage collection after a certain number of heap cells have been newly allocated (since the last garbage collection )

It is clear that such policy will perform garbage collection work linear in the size of the input list, at the cost of needing an effective heap large enough to contain both the input and output list.

The policy of [12] is attractive, however, a policy that does not deal with major collections at all, is unrealistic at the moment.