Segment Order Preserving and Generational Garbage Collection for Prolog

Ruben Vandeginste[1], Konstantinos Sagonas[2], and Bart Demoen[1]

[1] Department of Computer Science, Katholieke Universiteit Leuven, Belgium
{ruben,bmd}@cs.kuleuven.ac.be
[2] Computing Science Department, Uppsala Universitet, Sweden
kostis@csd.uu.se

Abstract. We treat two important issues in heap garbage collection for WAM-based Prolog systems. First we describe a new method for preserving the order of heap segments in a copying garbage collector. Second, we deal with methods for (multi-)generational garbage collection; in particular we show the importance of precise maintenance of generation lines and propose different and novel ways for its implementation. All the methods are experimentally evaluated.

1 Introduction

High-level languages like Prolog rely heavily on heap garbage collection and a lot of work has already been done in this area: the DEC-10 Prolog system (mid seventies) had a garbage collector and the earliest published description of the algorithms used in an implementation is probably [3]. An excellent account of issues in Prolog heap garbage collection can be found in [4]. As a result, several Prolog systems do have a heap garbage collector and it might appear that the issue is solved completely and in a satisfactory way. However, this is not the case: to start with, traditional heap garbage collectors are of the sliding type as described in [3, 15]. Still, there is evidence that the characteristics of a copying collector suit Prolog better: indeed, it has been observed that the percentage of live data in the heap is usually quite small, which makes copying a better choice than sliding because of its intrinsic properties. On the other hand, the first proposed copying collector for Prolog ([5]) was not *segment order preserving* and making a copying collector also preserve the order of segments is not easy. A heap *segment* is a part of the heap delimited by the heap pointers in two consecutive choice points. Preserving the order of segments is desirable, as this permits instant reclamation of a heap segment on backtracking. [9] presents an algorithm for preserving the segment order in a copying collector: the method relies on a traversal of the stacks from old to young. Finally, although [3] describes generational garbage collection—actually a two-generational schema—some basic problems of this schema are not discussed and moreover, lifting the schema to multi-generational garbage collection has not been attempted nor experimented with. In principle, the method of [9] does not combine well with generations because of rejuvenation of heap data; see [16]. So it is worthwhile continuing research and development in the area of copying collectors that preserve the order of heap segments and that allow (multi-)generational collection.

We begin by proposing a new algorithm for preserving the segment order in a copying collector (Section 2). The characteristics of this algorithm are described and variants are discussed. An experimental evaluation is presented, by comparing it to a previously existing copying garbage collector in the context of ilProlog. We also show the advantage of preserving segment order over not doing so.

At first sight, the issues of preserving the segment order and generational garbage collection seem unrelated. However, for optimal generational collection, it is essential that generation order is preserved. In principle, generation order can be preserved without preserving segment order. A sliding algorithm keeps the order of heap cells, and *a fortiori* of segments and generations. In a copying context, preserving generation order poses the same problem as preserving the segment order. In fact, our new method for preserving segment order resulted from research in multi-generational copying collectors; see [16].

Section 3 reviews the basic two-generational schema of [3]. The problems with this approach are discussed, the need for maintaining generations more precisely is shown, and different ways to achieve this are developed. This section is largely independent of which segment order preserving method is used and thus also useful in the context of sliding collectors which are still more commonly used in Prolog systems. Small changes to the underlying abstract machine are described. This section also contains an evaluation of all the techniques and a separate account of the cost of the changes to the WAM. Section 4 concludes.

2 A new segment order preserving copying collector

In general, a copying collector does not preserve the order of useful data. This is partly due to the fact that the root set is not ordered according to the heap pointers they contain and thus heap cells are forwarded in an *a priori* unknown order. Also, copying happens typically breadth-first (based on [7]) or depth-first with a recursive copying algorithm; neither of these algorithms retain the order of cells. This implies that a copying collector might both improve or deteriorate the locality of reference of the mutator. On the other hand, copying has the very attractive property of being linear in the size of the data to be copied, or stated in the garbage collection context: linear in the live data. In its most basic form, a sliding collector (based on [15] for instance) is linear in the size of the heap. So it follows that for a heap with a low occupancy, a copying collector performs better, unless more sophisticated techniques are used as for example in [8].

Usually, copying collectors do not have a marking phase either: by allocating a *to-space* that is as large as the *from-space*, there is the guarantee that there will be enough space to contain the copy, even in the worst case that there is no garbage. This is based on the fact that the size of a copied term is usually not larger than the original.

However, in the context of the WAM term representation, this is not true. A very simple example consists of the following goal: `Y = f(X)`, for which in the

WAM, the term `Y` occupies 2 cells in the heap; the self-reference `X` is the second one. The root set contains the variables `X` and `Y`. Assume `X` is copied first, and only later `Y`; then the total copy will contain 3 heap cells. [5] proposes a solution for this problem: start the collection with a traditional marking phase and during the copy phase forward contiguous blocks of marked heap cells together. That way, one avoids the undesirable situation that the copy is larger than the original.[1]

Since marking is needed, we might as well exploit it to our advantage. The method in [9] proposes to *count* the number of marked cells during the marking phase and use this number to reduce the memory needs: the to-space used for copying can actually be allocated only on demand and needs only be as large as to contain the surviving cells. Rather than swapping the two semi-spaces—which in many systems might not be possible as it might violate invariants (e.g., local stack and heap growing towards each other)—the to-space is immediately copied back to the original from-space and released after the copying phase. The copy-back is usually fast, and this method results in a memory usage that is in practice smaller than the factor two usually ascribed to copying collectors.

The experimental evaluation of all garbage collection schemes presented in this paper was performed on a Pentium III 866MHz with 256Mb RAM and timings are given in milliseconds. All memory sizes are given in number of cells: each cell is 4 bytes in this case. The timings include the gathering and printing of some statistics needed for producing the tables, but their impact is negligible.

2.1 The method: preserving the segment order by counting during marking

We make further use of the marking phase by *counting* the number of marked cells *in each segment*. To this end, before marking, we construct a list of segment counters with one counter for each segment, that are initialized to zero and a *segment list* of the heap addresses that delimit the heap segments: these addresses can be found in the choice points. Any time a heap cell is marked, a *lookup* in the segment list takes place in order to determine which segment counter needs incrementing. Thus, at the end of the marking phase, we know exactly how many surviving cells each segment contains. We can therefore allocate for each segment exactly the amount of space needed and during the copy phase copy each surviving cell to its corresponding to-space, possibly at the cost of another lookup. At the end, all *to-segments* are copied back to the original heap in the correct order.

We use a Cheney-like algorithm for copying; see [7]. This algorithm originally uses two pointers, named *scan* and *next*, which both point into the to-space. After the root set is forwarded, the copying continues until scan has caught up with next. The algorithm is easily adapted to copy segments to their corresponding to-space: each segment has now its own scan and next pointer. They are both initialized to the start of their to-space. After the root set has been forwarded, the scan pointer of each segment is made to catch up with its corresponding next pointer, at which point the copy is complete.

[1] No experimental evaluation of the extent to which this is a problem in practice exists.

Compared to traditional mark&copy, the overhead of preserving the segment order in the above method comes from looking up which segment a cell belongs to. The lookup function can become a bottleneck if there are many segments. By organizing the segment list as an ordered list, the cost is $O(log(n))$ where n is the number of segments. However, note that the lookup function is called once for each live cell during marking, but during copying, it is not called for every live cell: as explained in [5], contiguous blocks of marked cells are forwarded together, so the algorithm needs to call the lookup function only once for each contiguous block of marked cells during the copying. We have experimentally confirmed that one can observe this in practice. On the programs of our benchmark set, whose characteristics are presented in detail in Section 2.3, the number of lookups during copying is typically below 7% of the number of lookups during marking; being less than 1% quite often (in **browse**, **dnamatch**, **tsp**, and **muta**) and 19% only once (in **serial**).

Preserving the order of segments allows instant reclamation of heap upon backtracking. Losing this ability can in principle lead to a considerably larger number of calls to the garbage collector. Indeed, while the segment order preserving collector might allow enough instant reclaiming to avoid a future heap overflow, the non-preserving collector blocks garbage in an older segment and a later heap overflow can be provoked. The following artificial Prolog program shows this:

```
go :- mklist(650000,L1), cp(L1).

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

use(_).

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

Starting ilProlog with an initial heap size of 2M, triggers copying garbage collection twice when segment order is not preserved and only once when it is. Garbage collection figures are shown in Table 1. For comparison, we also give figures for performing garbage collection in SICStus 3.8.6 and Yap 4.3.0, just to show that our collectors are generally speaking of similar quality. Both SICStus and Yap are based on a sliding algorithm and started with initial heap such that they perform only one garbage collection during this small program.

However note that the effect of loss of instant reclaiming is countered partly by garbage collection itself: garbage trapped in a segment by not preserving the segment order during garbage collection, will be freed in the next (major) collection. In a copying collector, this comes virtually for free, as garbage is not looked at.

	m&c_gc	***sop_gc***	SICStus	Yap
total gc time	370	260	250	390
heap recovered	142147	2140138		

Table 1. Effect of recovery on backtracking on garbage collection time.

2.2 An optimization: collapsing segments

If there are many segments at the moment of the garbage collection, it might not be worthwhile to preserve all of them: when some small segments are collapsed into one segment, the loss of instant reclaiming is expected to be small as well. Since by collapsing segments the number of effective segments in the segment list decreases, the lookup function becomes faster. Collapsing segments can be implemented in several ways, but it is most easily visualized by merging two consecutive entries in the segment list when their combined size is less than a certain threshold.

Our implementation allows to specify this threshold at runtime or to opt for the default collapsing strategy which puts the threshold at the (initial) heap size divided by 64.

In [16] we describe another method for collapsing segments and in [18] evaluate its usefulness experimentally; more information can be found there.

2.3 Experimental evaluation

With the experimental evaluation of this section we aim at measuring the benefits and drawbacks of preserving the segment order and with that of Section 3.6 of different schemes for generational garbage collection. In this paper, we seek neither to compare our collectors with those of other Prolog systems, nor compare collection algorithms (e.g. sliding versus copying). The system chosen to implement our new garbage collection schemes is ilProlog, a Prolog system that is heavily used for machine learning applications; see e.g. [6]. ilProlog is WAM-based; see [19, 1]. Its main deviation from the WAM is that free variables are always put on the heap; see for instance [10]. Another deviation, like in many other Prolog systems, is to have separate stacks for the environments and the choice points. So we expect that our experimental results will also hold in other WAM-based systems. The advantage in doing all the experiments in the same system is that for sure the results cannot be affected by abstract machine differences, like for example term representation.

Prior to this research, ilProlog had a copying collector *ála* [5] which is not segment order preserving; we refer to it as ***m&c_gc***. All other collectors we implemented are based on ***m&c_gc***, in that they use basically the same marking and copying algorithm. The ilProlog compiler generates live variable maps, so the collector has precise information about which environment slots are live. No extra initialization of environment variables (as in SICStus for instance) is done nor needed. In fact, one of our starting principles was that we would not change the underlying Prolog engine to cater for garbage collection. In Section 3.4 we will deviate, albeit only slightly, from this principle.

Also, in this paper we will refrain from investigating garbage collection *policy* issues, and we will therefore *a priori* fix a (reasonable) policy: In all experiments, the ilProlog system was started with its default heap size (which is 256K cells, K = 1024) and with the size of the other stacks set high enough so that they do not need expansion. In several benchmarks the heap needs expanding: the heap size at the end of the run, together with the number of heap expansions, is indicated in the tables just below the name of the benchmark. The heap expansion policy doubles the heap size whenever more than 70% of the heap consists of live data after garbage collection. The policy must of course also take into account the amount of heap space that the caller of the collector requests to be free after the collection: this is relevant for built-in predicates like `findall/3`.[2] Expansion based on the requested amount of heap space only occurs in the **muta** benchmark. The ***m&c_gc*** collector collapses all segments, i.e., after a collection, there is only one non-empty segment. All other collectors are approximately segment order preserving, in that they collapse by default segments smaller than 4K cells with enough neighbouring segments so as to make the collapsed segments just larger than the threshold, but the order of the (collapsed) segments is preserved.

The small benchmarks are taken from [13].[3] The **muta** benchmark is much larger than the others and performs a typical run of an inductive learning tool written in ilProlog: without garbage collection, this application would not be able to run (this can also be seen by the number of collected garbage cells). The program uses abstract machine extensions that efficiently support query packs (see [6], also for more detail on this application) and thus does not run without changes in standard Prolog systems.

The first new collector we evaluate is referred to as ***sop_gc*** and implements the segment order preserving schema as described above. Table 2 shows data about ***m&c_gc*** and ***sop_gc***, in particular for each benchmark, we show: the total number of heap cells marked during the run of the benchmark, the number of heap cells reclaimed instantly on backtracking, the number of collected heap cells; the number of collections, the total marking time, the total garbage collection time, and the total runtime of the benchmark. As mentioned, the heap size at the end of the run is also shown: it indicates the extent to which heap expansion was needed.

From the figures in Table 2 we observe the following:

- The number of collections is the same in ***m&c_gc*** and ***sop_gc***: the potential increase in the number of garbage collections in ***m&c_gc*** seems not to occur in this benchmark set.
- The number of collected cells is most often the same; when it is different, the difference is small. Since garbage collections do not need to occur at the same moment during the execution of the program, such differences are difficult to predict or reason about. The total number of marked cells shows the same behaviour.
- The amount of heap recovered by instant reclaiming varies widely from 40% to 1% and is always larger for ***sop_gc*** in the small benchmarks; surprisingly, in the large benchmark, it is slightly smaller for ***sop_gc***. One should not

[2] The size of the answer list to be copied back to the heap is known before the copying starts.

[3] Some benchmarks used there are excluded because their memory usage makes them uninteresting; see [18].

benchmark		*m&c_gc*	*sop_gc*
boyer	total marked	72155851	72155851
8190K/5	heap recovered	7180443	11259478
	heap collected	67955085	67955085
	num collections	68	68
	total mark time	4470	5990
	total gc time	13630	16780
	total run time	29880	33070
browse	total marked	1380517	1380517
510K/1	heap recovered	14605833	14885921
	heap collected	968767	968767
	num collections	5	5
	total mark time	70	90
	total gc time	270	330
	total run time	9070	9160
dnamatch	total marked	4302012	4302012
254K/0	heap recovered	196609	198429
	heap collected	26915147	26915147
	num collections	120	120
	total mark time	230	330
	total gc time	710	770
	total run time	6140	6290
qsort	total marked	5308921	5308921
510K/1	heap recovered	1950205	2246796
	heap collected	5657071	5657071
	num collections	22	22
	total mark time	460	490
	total gc time	1110	1280
	total run time	2130	2340
serial	total marked	18209426	18209424
16382K/6	heap recovered	20272313	21720525
	heap collected	32104202	32104205
	num collections	9	9
	total mark time	1650	1830
	total gc time	3850	4310
	total run time	21730	22210
tsp	total marked	26909982	26909982
254K/0	heap recovered	236075	259553
	heap collected	271212807	271212807
	num collections	1146	1146
	total mark time	1210	1800
	total gc time	2690	3890
	total run time	65600	66810
muta	total marked	217118144	217120159
8190K/4	heap recovered	325879486	325879485
	heap collected	817187753	817187404
	num collections	267	267
	total mark time	14760	21490
	total gc time	46840	56340
	total run time	665060	672100

Table 2. Segment order preserving vs. non-segment preserving garbage collection.

draw general conclusions from these figures, as the amount of heap which is recovered by instant reclaiming is highly dependent on the initial heap size and the exact backtracking behaviour of the program.
- The mark time in ***m&c_gc*** is always smaller than in ***sop_gc***: this is caused by the overhead of the segment lookup and the counting per segment.
- The total garbage collection time in ***m&c_gc*** is always smaller than in ***sop_gc*** and the difference shows that this is not only due to the lookup cost. Indeed, we have measured for **boyer** for instance that the lookup function is called about 27 times as often during the marking phase than during copying, while the marking phase is responsible for only half of the speed loss of ***sop_gc*** compared to ***m&c_gc***. The extra overhead is in the set up of the segment lists at each garbage collection and the maintenance of and switching between the scan and next pointers for the different to-spaces.

Additionally, we note that in ***m&c_gc***, there is potentially more trailing, because segments are effectively made as old as the oldest one. However, we have hardly noticed this in the benchmark set used here; see also [18].

We can conclude that our segment order preserving method shows a varying, but non-negligible overhead over a plain mark&copy garbage collector. Preliminary attempts to speed up the lookup function further by applying heuristics show already good results, but given the philosophy of the method, counting per segment, we cannot hope for zero overhead. Still, the price is worth paying because preserving the segment order is the basis for a good generational schema that will be the subject of Section 3.

2.4 Comparison with other methods

As far as we know, there is only one other garbage collection method for the WAM heap that preserves the segment order during major collections: it was implemented in the BinProlog system [9]. The method starts by marking in the usual way from new to old. Then the copying phase visits segments in the root set from older to younger: in this way heap segments are copied also from old to new. The algorithm delays copying recursively a heap cell that points to a newer segment, until the moment it is encountered on the trail: it will necessarily appear there because of the trailing mechanism of the WAM. It is possible that the algorithm copies a heap cell that is not reachable from an older segment, but reachable from a newer segment, to a newer segment than it belonged to originally. This makes the method of [9] only approximately segment order preserving, but this phenomenon called *rejuvenation of data* is considered good: at no extra cost, the collector manages to make some data available for instant reclaiming sooner than normally. In the context of generational garbage collection however, the same algorithm might also move data from an older to a newer *generation* and thereby make the data subject to more frequent collections, which is exactly what generational garbage collection tries to avoid. Especially in view of the fact that this rejuvenating behavior of [9] 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, and thus impose overhead.

3 Generational garbage collection

Generational garbage collection [14] exploits the dynamic property exhibited by many programs that most objects die young while a small percentage lives 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 garbage collection schemes to properly work, it should be possible to find all references from objects in old generations (those areas 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*[4] 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. In imperative or functional programming languages, this results in a non-negligible additional runtime cost for supporting generational scavengers. See for instance [12] for more details about generational garbage collection.

3.1 Generational garbage collection in Prolog

The most commonly used virtual machine for Prolog execution, 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 makes generational garbage collection well-suited for WAM-based Prolog and 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 Prolog [3] which is two-generational.

Despite being well-suited for the WAM, besides segmented collection, only simple variants of generational copying garbage collection have been studied

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

before: Touati and Hama [17] 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 two-generational scheme with an immediate promotion policy which performs well on a set of three small benchmarks [5]. The experience from another two-generational scheme, having the additional constraint that the region to collect must fit in the cache, is similar; see [13].

The introduction of a two-generational garbage collection scheme to an existing Prolog collector is attractive and, especially in the context of a sliding collector, at first sight seems easy: one just has to be able to distinguish between the old and new generation. It has been noted already in [3] and later in [5] that the most recent choice point at the moment of garbage collection can be used as a barrier between generations. In such a scheme, the trail naturally acts as an exception list and the barrier is in fact a write-barrier. This is the scheme implemented in SICStus and we will henceforth refer to this scheme of delimiting generations as ***last_cp***.

3.2 A problem with this scheme: determinism

Note that in a segment preserving collector, the ***last_cp*** scheme is imprecise in remembering generations: after a collection, the heap possibly contains data on the top segment that will be subject to collection again in the next minor collection. The problem is potentially severe in the case where a heap-allocating deterministic computation follows after garbage collection: in this case no choice point is created and data in the new generation is repeatedly collected.

This can be very simply illustrated with a program that calls the garbage collector explicitly, as in:

```
go(N) :- mklist(N,L), b(L), use(L).

b([_|R]) :- garbage_collect, b(R).

use(_).

mklist(N,L) :-
    ( N =:= 0 ->
        L = []
    ;   Term = f(1,2,3,4,5,6,7,8,9,10),
        L = [Term|R], M is N - 1, mklist(M,R)
    ).
```

All collections are now major collections, and the total running time (including garbage collection time of course) of queries `?- go(`N`)` is $O(N^2)$, while without garbage collection, the execution of this program would be $O(N)$.

Although the above example is artificial, it clearly shows the inefficiency of ***last_cp***. This inefficiency has been identified as a real problem in practice. The example also shows that without a modification, ***last_cp*** cannot perform what we consider proper generational collection: data is repeatedly collected and never promoted to the old generation.

A simple-minded approach to avoid this phenomenon is to set the H pointer of the most recent choice point and the **HB** register to point to the top of heap after each minor garbage collection, and perform a similar action for the environment stack. The correctness of doing so is based on the fact that any choice point can be used as a generation delimiter: in the WAM it is correct to have a topmost sequence of choice points protect the local stack and heap after garbage collection. This solution results in increased trailing, but that is acceptable as it is a necessity in any generational scheme. However, it also over-approximates segments (the topmost segments are collapsed) and consequently, when non-determinism enters the picture, some opportunity for instant reclaiming is lost. A better way to deal with this inefficiency of ***last_cp*** is inspired by the following observation: the above example would run in linear time even with ***last_cp*** if the definition of `b/1` were slightly adapted to:

```
b([_|R]) :- ( garbage_collect ; fail ), b(R).
```

That is, remembering the last choice point at the moment of collection, gives precise information about the corresponding generation line. The next section exploits this idea further.

3.3 An initial solution: naive pushing of generational choice points

If one is interested in preserving instant reclaiming, another scheme that avoids multiple minor collections of the same generation is to always push a *generation delimiter choice point* after each garbage collection. For the current purpose, the alternative field (the failure continuation) of the generational choice point can be set to a special instruction that on backtracking just removes the choice point and fails, but which is distinguishable from any other possible alternative. Because of this property, one can easily avoid pushing more than one such choice point immediately after each other at the cost of collapsing collected generations; this is not an issue in a two-generational garbage collector. From the choice point chain at the moment of the next collection, it is now easy to recognize the choice point(s) which delimit consecutive generations. We will refer to this scheme of delimiting generations as ***push_cp***.

We note that this solution to the determinism problem of generational garbage collection is folklore and used to be implemented in ECLiPSe but was later switched off because it seemed to cause problems.[5] It has recently been adopted in SICStus Prolog (introduced in version 3.8.5). To the best of our knowledge, the ***push_cp*** technique has nowhere been described nor systematically investigated. Our aim was to fill this gap and in doing so, we indeed identified some problems. We begin by describing them below; their solutions are then presented in the following sections.

In a straightforward implementation, the ***push_cp*** scheme of delimiting generations potentially has the following problems: A generational choice point blocks the environment that is current at the moment of garbage collection—even if this environment belongs to an otherwise deterministic clause that would

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

be deallocated by last-call optimization—and keeps variables in this clause alive. We have observed that in such a naive implementation, these effects can result in a much larger local stack and can also worsen marking time as environments will be repeatedly traversed during the marking phase; see [18] for an example. A solution that avoids this problem is presented in Section 3.5. A second problem is that when garbage collections occur frequently (e.g. in a long-running application), a long chain of generational choice points might be created. Besides increased choice point stack consumption, it is crucial that these choice points do not result in additional complexity problems. The issue is also discussed further in Section 3.5. The last problem is related to generational choice points disappearing due to backtracking or the Prolog cut; this introduces impreciseness in remembering collected generations and subtly undermines the ***push_cp*** scheme. We address this issue first.

3.4 Precise schemes of maintaining generation lines

In order to have all information about which segments have been collected and how often, we maintain a *generation line list.* In practice this means that we keep a list of the contents of the pushed generational choice points together with their survival rate. This list also contains the topmost choice point at the moment of garbage collection. Based on this list, we can make a decision on which part of the heap to collect at the next garbage collection. This generation line list is also maintained on a major collection even though we do not yet exploit it for multi-generational garbage collection. As mentioned previously, backtracking (through execution of a trust instruction or one of its variants) can remove a generational choice point from the choice point stack. Instead of forgetting completely the generation delimited by this choice point, at the moment of executing the trust, we replace the corresponding information in the generation line list, by the choice point that now becomes the topmost. This means that trust must:

check whether the backtracked over choice point is indeed at the tail of the generation line list and if so:

- replace the generation line list tail by the new topmost choice point information, or
- if now the two most recent remembered generation lines are the same, pop one.

Another way to deal with disappearing generation delimiters is to employ an alternative stealing method similar in spirit to the *CHAT*-trick [11] or the actions performed in Muse [2]. More specifically, on backtracking over a generational choice point, we can promote the now top (Prolog) choice point to a generational choice point by stealing its failure continuation. We make it point to a new 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 list. The situation is slightly different from that in e.g. [11] 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 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 solution is slightly more complicated than in an emulator.

As mentioned, the execution of a Prolog cut (!/0) can also remove a generational choice point from the choice point stack. We consequently decided to make the cut operation check the generation line list and adapt it appropriately.

Note that both changes to the implementation of the WAM—the change of the trust instruction and of cut—put a small overhead on the abstract machine which is moreover at most linear in the number of created choice points and executed cuts, thereby not changing the complexity of any execution.

Two more methods for maintaining precise generation lines seem also attractive; both must be combined with pushing a generational choice point:

1. Define choice points to have an extra field, named e.g. *gc_info*; in this field, one can store the information on whether the segment just younger than the choice point, has been collected and how often. At garbage collection time, one then just scans the choice point stack to find which is the most recent generation delimiter. This scanning imposes only a constant overhead to the collection.
2. Keep in each heap segment 1 or 2 words for similar information; these two words should be the last two of a segment and can be installed easily only for collected segments. This method is easy in a copying collector; it is slightly more complex in a sliding context.

Both methods are quite simple, but the first one increases the size of all choice points. The second one is attractive because the price is only paid at garbage collection time—except for the small space overhead in each collected heap segment.

We have implemented the methods that change the implementation of trust and cut: their combined overhead is reported in Table 4. The garbage collector using this scheme is referred to as ***precise_cp***. Its implementation (as well as those of all other generational schemes) takes into account the issues described in the next section.

3.5 Refinements: careful handling of generational choice points

Pushing generational choice points can introduce significant overheads when repeated collections make a chain of generational choice points, which delimit very small segments. The problem with such behaviour is that the choice point stack can grow very large and at each garbage collection it gets traversed 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 a 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 for a chain of generational choice points (pushed by the garbage collector), on the condition that the segments they delimit have already reached tenuring age. One could then decide to *update* an existing 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. Currently, all our collectors that push a generational choice point, do this unconditionally, but as mentioned before, too small segments are collapsed.

In Section 3.3 we mentioned problems with a naive implementation of pushing generational choice points w.r.t. marking from the environment stack. However, one can observe that a generational choice point has no forward continuation. This means that the environment and continuation pointer in a generational choice point will never be used during execution, so it is correct to also *ignore* the generational choice points during garbage collection. During the marking phase this fits in quite easily and during the copying phase, it is also 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 is 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 (tops) pointer the same tops as in the previous choice point. In this way, the check for last-call optimization in the deallocate instruction, which looks at the tops in the current choice point, will not fail unnecessarily. This solution does not introduce any extra problems and in ilProlog is implemented in all collectors that push generational choice points (i.e., both in ***push_cp*** and in ***precise_cp***). However, 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.

3.6 Experimental evaluation

The experimental setting is as in Section 2.3 with the following additions. All generational collectors have been implemented to be multi-generational; i.e., they preserve generation information even after a major collection. The policy is kept fixed: data which has survived two collections is promoted to the part of the heap that is not collected during minor collections. This policy avoids tenuring cells that are too young at the point of a minor collection and were not yet given a proper chance to die. Thus, the collectors effectively operate in a three-generational mode. The expansion policy after a major collection is unchanged (i.e., as in the non-generational collectors). If a minor collection has recovered enough heap space for execution to continue, expansion does not occur. However, if the live data in the collected heap after a minor collection is more than 70%, the next collection will be a major one. This is done so as to avoid the case where a minor collection which does not recover enough heap space is immediately (i.e., in the same call to the collector) followed by a major collection; a scheme that visits data in the new generation twice. An alternative to forcing a major collection in the next cycle is to take advantage of the information about multiple generations and make an (informed) decision on the part of heap to collect at runtime. However, this is a policy issue, and not what we seek to investigate here.

Table 3 contains the obtained results. The measured items are as in Table 2 with the exception that heap reclaimed instantly is not shown as it is not interesting here, the number of collections is shown in the form Major+Minor, and we also present the maximum size of the choice point stack because extra choice points are pushed in some of the collectors. From the figures we observe that:

- The ***last_cp*** scheme indeed suffers from the problems mentioned in Section 3.2; the number of collected cells is almost always similar and the marked data is more than those marked by ***sop_gc*** (**boyer** is the only exception). This is partly explained by programs being mostly deterministic, but the lesson learned is that the use of an extra generational choice point is almost a necessity so as to get the behaviour expected from a generational garbage collector.
- Despite having the bigger number of collections, the ***push_cp*** and ***precise_cp*** schemes have the smaller mark and total garbage collection times and often the difference with those of ***last_cp*** is significant (cf. **qsort**, **tsp**, and especially **dnamatch** where the difference of marked cells is striking). Observe that this is the same set of benchmarks where these schemes result in increased choice point consumption.
- Disregarding increased choice point consumption, ***precise_cp*** is the overall winner. Compared with ***push_cp***, it manages to lower the number of major collections and the time needed for them (cf. **boyer** and **muta**). Although the figures do not show a uniform improvement of ***precise_cp*** over ***push_cp***, we definitely prefer ***precise_cp*** because it has better properties and an improved implementation of the same idea—perhaps using the alternative schemes suggested in Section 3.4—will probably beat ***push_cp*** consistently. Maintaining generational information precisely pays off.
- One benchmark, **serial**, reacts differently to the series of garbage collectors than the other benchmarks: ***sop_gc*** is faster than the generational schemes. The reason is twofold: First of all, **serial** is deterministic,[6] but this determinacy is obtained through cuts in the `split/4` predicate. Secondly, these cuts cut away almost all generational choice points as well as the last choice point that could have been used by ***last_cp***. This means that the generational schemes cannot lower the number of major collections. Indeed alternating minor and major collections take place that do not recover enough garbage; note the number of collections, expansions, and the final heap size. Since we have already put some overhead in the cut operation, a possible improvement here might be to make cut preserve any pushed generational choice points. This needs further investigation.
  The same phenomenon does not occur in **qsort** because the `partition/4` predicate does not provoke a garbage collection just before the cut.

[6] **serial** is actually a version of the quick sort algorithm transforming a list into a binary search tree.

			generational collectors		
benchmark		***sop_gc***	***last_cp***	***push_cp***	***precise_cp***
boyer	total marked	72155851	39053376	39053376	29020524
8190K/5	heap collected	67955085	65710207	65710207	65710207
	max cp	235	235	235	235
	num collections	68+0	15+57	15+57	7+65
	total mark time	5990	3310	3300	2570
	total gc time	16780	9300	9060	7080
	total run time	33070	25540	25410	23880
browse	total marked	1380517	1380496	854528	854528
510K/1	heap collected	968767	968767	955002	955002
	max cp	86	86	96	96
	num collections	5+0	2+3	2+3	2+3
	total mark time	90	110	70	80
	total gc time	330	350	200	210
	total run time	9160	9140	8990	9270
dnamatch	total marked	4302012	4301186	278518	278518
254K/0	heap collected	26915147	26915147	26965789	26965789
	max cp	68	68	1082	1082
	num collections	120+0	2+118	2+144	2+144
	total mark time	330	310	10	30
	total gc time	770	780	70	80
	total run time	6290	6270	5590	5690
qsort	total marked	5308921	5308781	3424126	3424126
510K/0	heap collected	5657071	5657071	5602942	5602942
	max cp	68	68	257	257
	num collections	22+0	2+20	5+23	5+23
	total mark time	490	480	280	280
	total gc time	1280	1270	820	760
	total run time	2340	2350	1930	1880
serial	total marked	18209424	29955995	29338655	29338655
16382K/6	heap collected	32104205	36600502	31395072	31395072
	max cp	68	68	138	138
	num collections	9+0	7+7	7+7	7+7
	total mark time	1830	2750	2740	2720
	total gc time	4310	6900	6970	6770
	total run time	22210	24630	25290	25890
tsp	total marked	26909982	26901974	635085	635085
254K/0	heap collected	271212807	271212807	271249819	271249819
	max cp	68	68	9544	9544
	num collections	1146+0	2+1144	2+1352	2+1352
	total mark time	1800	1880	440	400
	total gc time	3890	3980	580	580
	total run time	66810	66990	63720	63470
muta	total marked	217120159	220557488	215405117	215103780
8190K/4	heap collected	817187404	823263687	823263644	823185989
	max cp	7495	7495	7495	7495
	num collections	267+0	22+250	22+250	7+265
	total mark time	21490	21850	21390	21190
	total gc time	56340	58220	56780	56530
	total run time	672100	675640	674980	668680

Table 3. Comparison of different generational copying garbage collectors.

benchmark	***sop_gc***	***precise_cp***
boyer	2000	2040
browse	1590	1540
dnamatch	2100	2140
qsort	1010	1010
serial	1850	1820
tsp	3020	3170

Table 4. Overhead of maintaining precise generational information.

– **muta** does not benefit significantly from the generational schemes. The reason is that **muta** has very small old generations, namely in the order of 1% of the total heap or even less, meaning that most effort put in maintaining generations and in performing a generational collection is wasted.

Finally, to get an idea of the overhead in the mutator due to maintaining the generation lines precisely, the six small benchmarks were run with enough initial heap so as not to trigger garbage collection (and some of them with a smaller input so as to achieve this) in ***sop_gc*** and ***precise_cp***. Results are shown in Table 4. The difference in execution time ranges from -3% to +5% which is close to noise level. However, it is safe to conclude that the overhead of maintaining precise generational information is small and in any case acceptable in practice.

4 Conclusion

We presented a new algorithm for preserving the order of heap segments in a copying garbage collector. Within the same context, we showed how a collector can be made generational and how pushing generational choice points and maintaining precise generational information is necessary for proper generational collection and leads to good performance. Most of these techniques are new and even the rare one (***push_cp***) which is folklore has not, to the best of our knowledge, been investigated systematically. It is clear that alternative schemes are possible: For example, by devoting a new field in every choice point just for garbage collection, we can associate with each segment its survival count and in this way maintain precise generation information even more easily, at the cost of increased choice point stack usage and a slightly more costly choice point creation. Whichever method is chosen, it does not seem reasonable to expect that it incurs no overhead during the execution of the mutator if one wants optimal performance. Also a formal model of generational garbage collection in the presence of backtracking seems needed, in order to provide a basis for adaptive garbage collection policies. We intend to investigate these issues.

References

1. H. Aït-Kaci. *Warren's Abstract Machine: A Tutorial Reconstruction.* The MIT Press, 1991. See also: http://www.isg.sfu.ca/~hak/documents/wam.html.
2. K. A. M. Ali and R. Karlsson. The Muse approach to OR-parallel Prolog. *International Journal of Parallel Programming*, 19(2):129–162, Apr. 1990.

3. 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.
4. 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.
5. 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.
6. H. Blockeel, L. Dehaspe, B. Demoen, G. Janssens, J. Ramon, and H. Vandecasteele. Executing query packs in ILP. In J. Cussens and A. Frisch, editors, *Proceedings of the 10th International Conference on Inductive Logic Programming*, number 1866 in LNAI, pages 60–77, Springer, July 2000.
7. C. J. Cheney. A nonrecursive list compacting algorithm. *Communications of the ACM*, 13(11):677–678, Nov. 1970.
8. Yoo C. Chung, Soo-Mook Moon, Kemal Ebcioglu, and Dan Sahlin. Reducing sweep time for a nearly empty heap. In *Proceedings of the ACM SIGPLAN Symposium on Principles of Programming Languages*, pages 378–389. ACM Press, Jan. 2000.
9. 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.
10. B. Demoen, P.-L. Nguyen. *So many WAM variations, so little time.* In J. Lloyd, editor, *Proceedings of Computational Logic - CL2000, First International Conference*, number 1861 in LNAI, pages 1240–1254, Springer, July 2000.
11. B. Demoen and K. Sagonas. CHAT is Θ(SLG-WAM). In H. Ganzinger, D. McAllester, and A. Voronkov, editors, *Proceedings of the 6th International Conference on Logic for Programming and Automated Reasoning*, number 1705 in LNAI, pages 337–357. Springer, Sept. 1999.
12. R. Jones and R. Lins. *Garbage Collection: Algorithms for automatic memory management.* John Wiley, 1996.
13. X. Li. Efficient memory management in a merged heap/stack Prolog machine. In *Proceedings of the 2nd ACM SIGPLAN Conference on Principles and Practice of Declarative Programming (PPDP'00)*, pages 245–256. ACM Press, Sept. 2000.
14. 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.
15. F. L. Morris. A time- and space-efficient garbage compaction algorithm. *Communications of the ACM*, 21(8):662–665, Aug. 1978.
16. 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. See http://www.cs.kuleuven.ac.be/publicaties/rapporten/cw/CW303.abs.html
17. 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.
18. R. Vandeginste and B. Demoen. The implementation of a new segment preserving and/or (multi-)generational copying garbage collection for the WAM and its approximation. K. U. Leuven CW report 319, July 2001. See also: http://www.cs.kuleuven.ac.be/publicaties/rapporten/cw/CW319.abs.html
19. D. H. D. Warren. An abstract Prolog instruction set. Technical Report 309, SRI International, Menlo Park, U.S.A., Oct. 1983.