SEGMENT ORDER PRESERVING COPYING GARBAGE COLLECTION FOR WAM BASED PROLOG

Bart Demoen
Katholieke Universiteit Leuven
bimbart@cs.kuleuven.ac.be

Geert Engels
Katholieke Universiteit Leuven

Paul Tarau
Université de Moncton
tarau@info.umoncton.ca

Keywords: *memory management of logic programming languages, segment order preserving copying garbage collection, WAM based Prolog implementation.*

Abstract

We present a bottom-up copying garbage collector for WAM based Prolog. In contrast to the top-down copying garbage collector of Bevemyr and Lindgren, it preserves the order of the heap segments in the WAM like sliding collectors do, so that space can be reclaimed on backtracking. Moreover, the collector can move some old data to newer segments so that future instant reclaiming becomes better than with sliding collectors; this behavior can not be explained by *usefulness logic* and applies when the underlying Prolog machine implements *choicepoint trimming*. The algorithm is mostly explained informally through examples.

The garbage collector is implemented in the context of BinProlog, a production quality Prolog system working on various 32-bit and 64-bit architectures. It shows outstanding practical performance when compared with the high-quality sliding collector of SICStus Prolog.

1 Introduction

Garbage collection of the Prolog heap (also called global stack), has been described in a sequence of papers: [10], [2], [3]. In most Prolog implementations, the garbage collector is of the sliding type because the order of the segments is important for almost free (and constant time) de-allocation of a heap segment on backtracking (following [3], we will call this *instant reclaiming*) and for conditional trailing. The order of cells inside one segment is also critically used by most implementations for the realization of the compare/3 built-in predicate; however, this last issue has become obsolete due to the decision of [8] which does no longer impose a persistent order on unbound variables.

Compared to a sliding collector, a copying garbage collector is attractive because its theoretical complexity is linear in the size of the useful data, while a sliding collector has a component which is linear in the size of all the data (an improvement on this is described in [12]).

On the other hand, a copying collector needs a *to* and a *from* space, so that in a language with backtracking it can be called arbitrarily more often than a sliding collector, given the same total amount of memory.

Recently, [4] described a top-down copying collector (whose details are recalled in section 2) which has the drawback that the order of heap segments is not preserved. This prevents cheap instant reclaiming. [4] reports that this doesn't show up as a real problem in a set of benchmarks. However, this is highly program dependent and in fact, the behavior of an order-losing collector, can be arbitrarily worse than when order is preserved (see section 4). These considerations motivated our work on a practical copying collector in the context of a high performance (WAM-based) Prolog implementation that would at least preserve the order of the heap segments: it is known from literature [3] that bottom-up copying allows instant reclaiming and in fact, the MALI-implementation [11] contains a bottom-up copying part since many years. We will show that our bottom-up copying algorithm not only preserves instant reclaiming, but actually can improve it by moving objects to newer segments when this is allowed. MALI has the same property and potentially improves instant reclaiming more than our algorithm. However, this property of MALI was never recognized, not even by its authors. Also, due to its data representation, MALI does not lead to an efficient Prolog implementation, so it is worthwhile to study the algorithm in the context of WAM. Bottom-up copying is described in section 3.

Our collector was implemented in the BinProlog system [15]. More details can be found in section 5.

Section 6 reports on the performance of our implementation.

We assume knowledge of WAM [18], [1]. We use the following symbols, terminology and conventions:

- H: the top of heap pointer: new data is always created at the top of the heap
- HB: the saved heap pointer in the top choicepoint; it acts as a write-barrier: all changes to the portion of the heap that is younger than HB are recorded on the trail, which is the equivalent of the exception list in functional language implementations
- Areg[i]: the i-th argument register
- the heap grows downwards in pictures
- a *segment* is a piece of memory allocated (or created or become in use) between the creation of two successive choicepoints (or before the first or after the last choicepoint): we will refer to segments on the trail, the heap and the local stack; the argument registers always belong to the most recent segment; a choicepoint itself, although strictly created 'between' two segments, belongs to the older of the two segments; we number segments, starting with 1 for the oldest, and upward; note that trail segment 1 is always empty in WAM; a segment in WAM can be seen as a generation: each generation is delimited by a choicepoint and each set of generations up to a certain age has its own exception list, called a trail segment, where changes are recorded
- gc will be used as a shorthand for garbage collection or collector in running text; when gc appears as a goal

in Prolog code, it means an activation of the heap garbage collector

- a goal *use* with any arity, in Prolog code, indicates that its arguments are still alive at that point in the program
- heap cells have a *tag*: we need only the STRUCT-tag which is abbreviated to S; functors will be represented directly; references and unbound variables will be represented by (self-)pointers

A gc collects the useful or live data; this is the data accessible from a set of root pointers; for a WAM-like implementation, the root pointers are found in the argument registers and on the local stack: choicepoints and environments. The entries on the trail stack are not proper root pointers, but they must be relocated. One can visit the root pointers from more recently created (newer) frames to the older frames: such a traversal is referred to as *top-down*. Traversing the root pointers from old to new, is called *bottom-up*. We shall present the algorithm in an informal way through examples and pictures: for every Prolog code example, the query is ?- run. The algorithm is described more formally in the appendix.

Early reset [2] is a technique which during gc, undoes bindings no longer accessible in forward execution mode.

2 Garbage collection by top-down copying

[4] presents a garbage collector for the Prolog heap, by top-down copying. Two important issues are involved: the order on the heap is not preserved, and special care must be taken with the copying of arguments of structures. Both are illustrated with a small example:

```
run :- X = f(Y) , b(Y) , use(X) .
b(Y) :- Y = g(8) , gc , use(Y) .
b(Y) :- ...
```

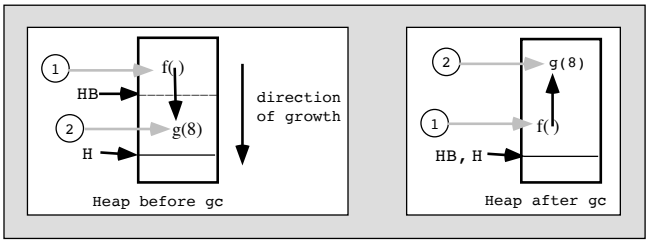

Figure 1: Losing the segments by top-down copying

HB is the H pointer from the choicepoint of b/1 (see figure 1). The (shaded) root pointers, carry the number of the segment from which they originate. Before gc, there were two heap segments; after gc, there is only one heap segment and H == HB, therefore, backtracking to the second alternative of b/1 will not reclaim the term g(8). The next example shows the problem of internal cells:

```
run :- b(X,f(X)).
b(A,B) :- gc , use(A,B).
```

Figure 2 shows that by copying Areg[1] first, one ends up with a copy which is larger than the original: this problem is solved by a marking phase introduced by [4]. Section 3.4 describes this marking phase and also our modification.

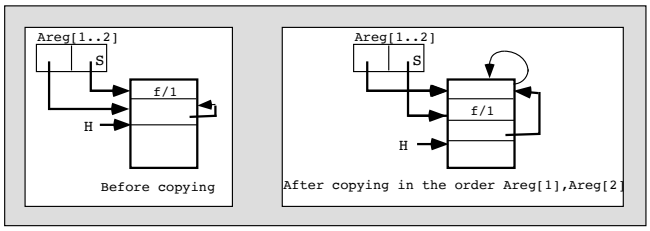

Figure 2: Copying twice an internal cell

3 Garbage collection by bottom-up copying

Let $seg(cell)_t$ denote the number of the heap segment to which *cell* belongs at time t. It is clear that if a garbage collector obeys:

$$seg(cell)_{after\_gc} \leq seg(cell)_{before\_gc} \quad (1)$$

it is correct as far as instant reclaiming is concerned. Top-down copying as in [4], has the property that for every cell, $seg(cell)_{after\_gc} = 1$, so that for most cells, strict inequality in (1) holds: this expresses the loss of instant reclaiming. A strictly heap segment order preserving collector has equality for all cells in (1). We will show in subsection 3.1 that a naive implementation of bottom-up copying not necessarily preserves the heap segment order. Our bottom-up copying method presented in subsection 3.2 does preserve segment order. Finally, given a state-of-the-art compiler (to WAM-like code), our algorithm is able to correctly reverse the inequality in (1): for some cells

$$seg(cell)_{after\_gc} > seg(cell)_{before\_gc} \quad (2)$$

i.e. the cell has moved from an older to a newer segment, meaning that this cell will be reclaimed (on backtracking) earlier than without garbage collection: our collector is unique in this respect amongst WAM-based implementations. Subsection 3.3 discusses this.

3.1 Naive bottom-up copying

The Prolog code for the example is the same as in the first example of section 2; figure 3 illustrates the point.

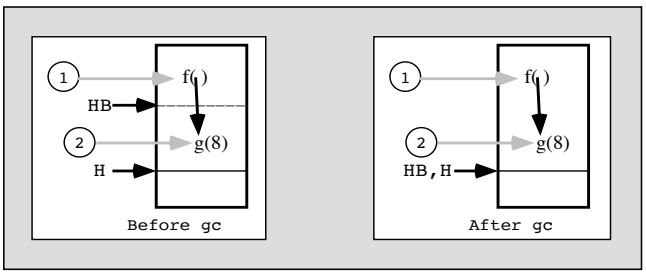

Figure 3: Losing segments by naive bottom-up copying

Copying (everything reachable from) segment 1 first, does not guarantee that no cells move from segment 2 to segment 1: while copying the object accessible from X in the clause for run/0, the term g(8) has moved to segment 1.

3.2 Using the trail during bottom-up copying

The same code example as before is used; however, the picture is completed to include the trail entry: see figure 4.

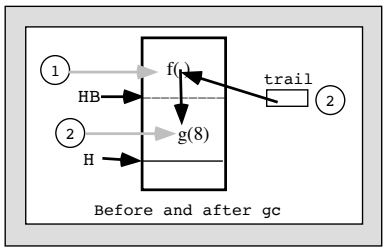

Figure 4: The trail records forward inter-segment pointers

Trail segment 2 contains the pointer to the cell with the pointer from heap segment 1 to heap segment 2: the copying of this cell is postponed (because it is a forward inter-segment pointer) until (trail) segment 2 is treated.

The key idea of the algorithm[1] is as follows:

> if during the treatment of segment i, a cell c is encountered, which contains a reference to an object o in segment $(i+n)(n>0)$, then o is not copied to segment i; c is necessarily pointed to by a trail entry belonging to segment $(i+n')(n \leq n')$; o is copied at the latest when segment $(i+n')$ is treated

The check whether a pointer crosses a segment boundary is cheap since we are only interested in pointers from an old to a newer segment: one comparison is enough.

The example and the verbal description of the algorithm hide some complexity: the object o can itself contain a pointer that crosses a segment boundary in either direction and so on. The full algorithm takes care of this.

3.3 Improving instant reclaiming at no extra cost

The previous discussion leads to the following question: is for any cell inequality (2) possibly correct? The question amounts to: is it possible that an object was created in segment i and is only reachable from segment $(i+n)(n>0)$? The answer in WAM is negative; the basic reason is that an object created in segment i becomes accessible to segment $(i+1)$ through choicepoint i, so that it is also accessible to segment i. Nevertheless, the following example shows that there is no fundamental necessity for making objects always available through the choicepoint:

```
run :- X = f(9) , b(X) .
b(Y) :- gc, use(Y) .
b(_) :- ...
```

Note the void variable in the second clause of `b/1` and how HB in the right part of the picture has moved up. Usually, WAM implementations save the value of Areg[1] in the choicepoint of `b/1`, so that there is a reference to the term `f(9)` from the choicepoint. However, the compiler can figure out easily that there is no need to save Areg[1] in the choicepoint, as this argument is not needed after backtracking to the choicepoint of `b/1`. This situation has been noted by

[1] see Appendix

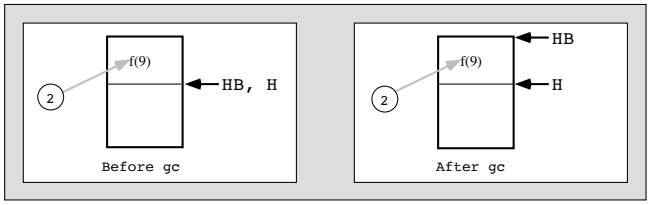

Figure 5: Moving an object to a newer segment

several implementors (e.g. [20]) and was named *choicepoint trimming* in [9]. For the above example, the state of execution just before and just after gc is shown in figure 5, where the term `f(9)` is only accessible from segment 2, i.e. from the variable Y in the environment of the first clause of `b/1`. It is clear that the situation after gc is still consistent, i.e. all useful data is still reachable, and after backtracking to the second alternative of `b/1`, the term `f(9)` will have been de-allocated. Our algorithm achieves moving objects to a newer segment without any extra effort; it is in fact costly to avoid this behavior.

Choicepoint trimming is more often applicable than one might think; even the member/2 predicate lends itself perfectly to choicepoint trimming: see [9] for more details.

3.4 The adaptation to the marking phase

[4] introduced a marking phase before starting the copy phase: some form of marking is necessary anyway for doing early reset and it does not alter the complexity of the total algorithm. Then, before copying a cell c, [4] makes a scan upward of the heap, starting at c, until an unmarked cell is found; then the whole block of marked cells is copied at once; this serves the purpose of preventing an extra cell in the copy for internal cells. Since we want to move as many objects as possible to newer segments, we cannot always copy such a contiguous block of marked cells: we can only copy the immediately enveloping structure of c. Consider as example:

```
run :- X = f(A1,A2,...,An) , gc , use(An,...,A2,A1) .
```

Assume that the argument registers are copied in the order Areg[1] to Areg[n], then on copying Areg[i], a scan is made upwards to detect whether the object Ai belongs to a marked structure or not; since it doesn't (the functor cell f/n is not reachable), we copy only the object Ai. The repeated scanning introduces in this way a quadratic worst-case complexity.

To remedy this, we have introduced an extra bit per heap cell, which distinguishes between cells internal to a marked structure and other marked cells. The extra work for setting this bit is negligible (in particular, it does not change the complexity of the marking) and is done during the top-down marking phase.

4 The need to keep the order of segments

It is easy to mimic the effect of a top-down copying gc with any collector (sliding or not), by putting HB and the saved H pointers in all the choicepoints equal to the value of H immediately after gc. We do not challenge the findings of [4]

on a set of benchmarks, namely that the loss of instant reclaiming (and extra trailing) due to non-order preserving gc is small: we have performed the above adaptation in an existing (sliding) collector and the systematic increase in total gc time and number of calls to the collector when the order is not preserved, is indeed not prohibitive. However, a simple example shows that generational gc without preserving the order of segments, is arbitrarily worse than generational gc which does preserve the order:

```
run :- X = f(9) , gc , use(X) , fail .
run :- run .
run .   % prevents the choicepoint
        % of run/0 from vanishing
```

GC without keeping the order, makes instant reclaiming of the term `f(9)` on backtracking to the second clause of `run/0` impossible, and since this term belongs to the already collected generation, it will not be collected by the next generational gc: the heap keeps growing and a major collection is eventually needed. In contrast, any order preserving (even generational) gc, allows the program to run in constant heap space (there is actually nothing to collect). The example exhibits also an ever growing local stack, but it is easily adapted to show that the heap can grow arbitrarily faster than the local stack. It probably shows the worst case for generational non-order preserving gc but it also indicates that in general, one must expect more major collections in a non-order preserving collection schema. This, together with the insight this paper brings that the cost of preserving (and even improving) the segment order is negligible, shows that there is no need to lose the order of segments.

5 The implementation in BinProlog

The algorithm described informally in section 3, has been the basis of an implementation of a garbage collector in BinProlog [15]. The choice for a copying collector is even more justified in the context of BinProlog which is based on continuations and therefore keeps the equivalent of WAM environments on the heap: these structures are not recoverable on forward execution, so that the ratio garbage/useful data is higher than in a straight WAM implementation. It then becomes even more attractive to have a collector which behaves linearly in the size of the useful data. The BinProlog compiler does not implement choice point trimming, so we had to introduce some low level hacks and source transformations to effectively observe that our collector indeed does move objects to newer segments. BinProlog implements a different data representation than usual WAM: it employs tag-on-data instead of tag-on-pointer and uses a data compression technique resembling cdr-coding, called last argument overlapping (LAO) [16]: it complicates slightly the copy phase of the gc. BinProlog has some other non-standard features: a blackboard, a global hash table which can share data with the heap, backtrackable destructive assignment and a copy-once-splitting-heap implementation of `findall/3` [13]: our collector is adapted appropriately. Also generational gc is catered for and we have made adaptations to the copying algorithm of Cheney [6] related to the necessity of keeping the reference chains and the preservation LAO. A full discussion of all this, is beyond the scope of this paper. More details can be found in [7]. The garbage collector is now an integral part of BinProlog.

6 Performance evaluation

Establishing the quality of a new gc is best done through a comparison with an existing gc whose quality is beyond doubt; we therefore choose to evaluate our copying collector against the sliding gc in the high quality SICStus Prolog implementation [5]. Its emulator is of similar quality as the BinProlog emulator. Details about its gc can be found in [2]. Direct comparison with the implementation of [4] would have been less significant, because the underlying Prolog machines are too different.

In order to make the comparison between these two gcs fair and meaningful, we made sure that BinProlog (version 3.85) and SICStus Prolog (release `2.1#9`) have the same memory management policy with respect to the heap, i.e. they can be started with exactly the same initial heap space, which is never increased nor decreased during the running of the program. Overflow is checked for in both systems at the entrance of a predicate (at the so called CALL-port). The margin at the end of the heap which is protected is equal. Both systems compact the trail during garbage collection and do early reset.

We have switched off the "variable shunting" pass of SICStus and its segmented (or generational) aspect, in order to focus entirely on the difference in performance between a sliding and a copying collector.

Both systems implement backtrackable destructive assignment using the trail and their gcs treat such trail entries accordingly; however, none of the benchmarks makes use of destructive assignment.

Both Prolog systems are based on WAM [18, 1], however, BinProlog is continuation based, i.e. instead of managing a stack of environments, it puts continuations on the heap. Therefore, BinProlog consumes more heap than SICStus for ordinary programs. However, a binary program does not need environments in WAM and no continuations on the heap, so that by binarizing the benchmark programs, both systems behave very similar in terms of heap usage. Binarization introduces metacalls which are expensive in SICStus because the module is resolved at run-time, which consumes heap, so we have replaced metacalls in the binarized programs by a specialized user predicate with zero heap consumption.

While SICStus (as usual in WAM) optimizes lists (i.e. compound term with principal functor `./2`), BinProlog treats all functors in a uniform way, so we have changed the original programs by replacing all occurrences of `./2` by an arbitrary functor with arity 2.

Two differences between BinProlog and SICStus could not be eliminated: BinProlog implements LAO; this optimization is difficult to switch off and accounts for its smaller heap consumption on some benchmarks. On the other hand, the superior indexing strategy of SICStus results in a heap consumption which is sometimes smaller, because less choicepoints means less reachable data in general.

Both collectors were written in C and were compiled with the same gcc options (-O2 -msupersparc -fomit-frame-pointer) on SUN SPARCclassic (SUN 4/15N-32-P43) with 32Mb RAM. SICStus was compiled without support for binary decision diagrams which could have affected the gc time. As the performance of emulated SICStus is close to emulated BinProlog, we have used emulated mode in both systems.

We made a big effort to make the heap management policy of both systems identical and their heap usage very close: we believe that the remaining differences are so small that the measurements are meaningful.

The benchmark programs were partly taken from [4]: boyer, tsp and match; we added the program allperms which computes deterministically a list of all permutations of a given

p	s	h	gcs	t	u
match	Bin	1000	35	510	820
match	SICS	1000	39	3810	899
match	Bin	500	73	980	1725
tsp	Bin	1000	10	50	88
tsp	SICS	1000	10	880	54
tsp	Bin	500	20	110	176
boyer	Bin	1000	11	780	1238
boyer	SICS	1000	10	1550	1056
boyer	Bin	500	26	1760	3008
queen3	Bin	150	56	1310	1495
queen3	SICS	150	82	3590	2711
queen3	Bin	75	152	2380	3202

Figure 6: Performance comparison I

p	s	g	u/t	g/t	gain
match	Bin	34020	1.61	66.71	3.0
match	SICS	37923	0.24	9.95	-4.5
match	Bin	34441	1.76	35.14	1.1
tsp	Bin	9866	1.76	197.32	4.8
tsp	SICS	9899	0.06	11.25	-5.7
tsp	Bin	9732	1.60	88.47	4.6
boyer	Bin	9711	1.59	12.45	-5.7
boyer	SICS	7945	0.70	5.13	-8.3
boyer	Bin	9872	1.71	5.61	-14.9
queen3	Bin	6647	1.14	5.07	-5.3
queen3	SICS	9213	0.76	2.57	-15.1
queen3	Bin	7500	1.35	3.15	-14.5

Figure 7: Performance comparison II

list from [14], and queen3, a 3-dimensional variant of the N-queens problem, used by [19] for the evaluation of a parallel gc for an OR-parallel Prolog implementation.

In the figures 6 and 7, we give for each test the total number of garbage collections, the total time of gc and the space collected (both garbage and useful data). The meaning of the columns in figure 6 and 7 is as follows.

- p = program
- s = system (Bin or SICS)
- h = available heap size (in Kb)
- gcs = number of garbage collections
- t = total time of the gcs (in ms)
- u = total useful data after collection (in Kb)
- g = total garbage recovered (in Kb)
- u/t = amount of useful data after gc per sec (in Mb/s)
- g/t = amount of recovered garbage during gc per sec (in Mb/s)
- gain = speedup of program execution time due to gc; in percentage, i.e (time_without_gc - time_with_gc) * 100 / time_without_gc

The most relevant columns are total gc time (t) and useful data recovered per time unit (u/t).

The allperms benchmark computes a list of all permutations only part of which is kept for further computation; the part which is kept and which is collected as useful by the gc, is

s	h1 (Kb)	h2 (Kb)	t (ms)
Bin	4627	2755	1450
SICS	4788	2915	2270
Bin	4627	2411	1270
SICS	4788	2550	2040
Bin	4627	2066	1090
SICS	4788	2186	1800
Bin	4627	1722	910
SICS	4788	1822	1570
Bin	4627	1377	720
SICS	4788	1457	1340
Bin	4627	1033	540
SICS	4788	1093	1110
Bin	4627	689	360
SICS	4788	728	870
Bin	4627	344	180
SICS	4788	364	630
Bin	4627	0	0
SICS	4788	0	400

Figure 8: Copying vs. sliding collector on variable useful data

variable (h2 in figure 8), while the size of the heap at the moment of gc (i.e. when all permutations have been computed), is constant (h1). This benchmark illustrates the complexity of both garbage collectors. The difference for the figures for h1 and h2 between Bin and SICS, is to be attributed to LAO only and is small: 3.5% and 5.8% respectively, smaller than the relative differences in the timings.

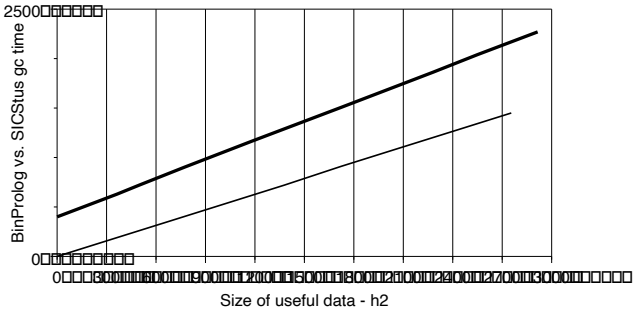

Figure 9: GC time as a function on the size of useful data

The figure 8 and its graphical representation (with the lower line representing BinProlog) figure 9 show that, given a fixed amount of heap, both collectors perform linear in the size of the useful data (`h2`); while the copying collector is independent of the size of the actual heap (`h1`) (illustrated by the fact that `t` = 0 for `h2` = 0), one sees that the sliding collector of SICS has a fixed extra cost which in general is linear in `h1`, and which is constant here, because `h1` is the same for each test. The fact that the sliding collector is linear in the size of the useful data, given a constant heap, reflects its phases: mark (linear in useful data) and compact (linear in total heap). Our collector outperforms the SICStus collector systematically.

Also figures 6 and 7 show clearly that our copying gc performs quite well compared to the high quality sliding collector in SICStus Prolog. In fact, our gc takes always less time for the same heap size, and almost always less even

if only half the heap space is given to our copying collector, which is then activated twice (or more) as often: this answers the often heard argumentation (e.g. [4]) that a fair comparison between sliding and copying collection must give only half the space to the copying collector as the sliding collector works in-place; this argumentation is also debatable, because on modern operating systems like Solaris or Windows NT, having a large address space is zero-cost as far as the space is not actually accessed. Moreover, in our implementation, the *to* space is effectively freed after gc, so that the extra space consumption is limited to the duration of the gc.

7 Related work

Copying gcs for Prolog were known before from [11] and [17] (see [3] for an almost complete account of gc for Prolog) and more recently from [4]. In the context of WAM-based Prolog systems, our garbage collector is the first to employ copying for the whole heap while keeping the segment order. It is also the first garbage collector for Prolog which was shown to improve instant reclaiming. The bottom-up and segment preserving gc in MALI does also improve instant reclaiming (and might do it better), but MALI makes assumptions on the representation of heap structures which do not hold in (Bin)WAM and which are even unrealistic in an optimized implementation of Prolog. Moreover, the authors of MALI have never realized that their gc does in fact improve instant reclaiming: this is not surprising, since the notion of choicepoint trimming is more recent than the conception of MALI. The improvement in instant reclaiming is probably not of the order to make a significant difference for most programs and requires compilers to implement choicepoint trimming. However, we have clearly shown that there is no reason to implement a copying gc which loses the order of the segments as in [4].

The fact that bottom-up copying can move older data to newer segments, so that this data becomes earlier available for instant reclaiming, shows that the notion of *usefulness logic* [3] has its limitations: usefulness logic explains the phenomenon of early reset and variable shunting, but since it maps data to {garbage, useful}, it cannot explain improved instant reclaiming. Our algorithm, as well as the algorithm in [3], distinguishes implicitly between {garbage, useful(1), useful(2), ...} where useful(N) means: useful for forward execution after backtracking to choicepoint N. For the youngest segment this means that effectively a distinction is made between data needed for the forward execution and data still needed after backtracking: data only needed for forward execution is moved to the top heap segment. The algorithm does not need to inspect the program for this: the current state of the execution is sufficient. We believe that usefulness logic should be reconsidered taking this realization into account and that a different approach to gc (and in particular to the age of an object) is required.

8 Conclusion

The collector was implemented and tested on top of BinProlog[2], a production quality continuation passing Prolog system working on various 32-bit and 64-bit RISC and Intel architectures. The garbage collector shows outstanding practical performance compared with the high-quality sliding collector of SICStus Prolog and shows the theoretical linear complexity of a copying gc. The actual implementation also deals with BinProlog's compressed term representation while supporting the memory management requirements of Prolog extensions like backtrackable destructive assignment, *copy once* all-solution predicates [13], multiple logic engines and blackboards, linear and intuitionistic implication. Our work shows that segment order preserving copying garbage collection is practical for WAM based Prolog systems and competitive with the best existing GC technology.

[2] Available by ftp from clement.info.umoncton.ca.

9 Acknowledgements

The authors wish to thank an anonymous referee for help with related work and Thomas Lindgren for his benchmark programs. Bart Demoen thanks the Belgian Ministry (DWTC) for support through project IT/IF/4. Paul Tarau thanks for support from NSERC (grant OGP0107411), from the FESR of the Université de Moncton, and from K.U.Leuven through Fellowship F/93/36.

References

[1] H. Aït-Kaci. *Warren's Abstract Machine: A Tutorial Reconstruction*. MIT Press, 1991.

[2] K. Appleby, M. Carlsson, S. Haridi, and D. Sahlin. Garbage collection for Prolog based on the WAM. *CACM*, 6(31), 1988.

[3] Y. Bekkers, O. Ridoux, and L. Ungaro. Dynamic memory management for Sequential Logic Programming Languages. In Y. Bekkers and J. Cohen, editors, *the Proceedings of WMM'92, Rennes*, pages 82–102. LNCS 637, 1992.

[4] J. Bevemyr and T. Lindgren. A simple and efficient copying garbage collector for Prolog. In *the Proceedings of PLILP'94, Madrid, M. Hermenegildo, J. Penjam*, pages 88–101. LNCS 844, 1994.

[5] M. Carlsson, J. Widen, J. Andersson, K. B. S. Andersson, H. Nilsson, and T. Sjöland. SICStus Prolog User's Manual. Technical Report T91:11B, SICS, Kista, Sweden, 1991.

[6] C. J. Cheney. A nonrecursive list compacting Algorithm. *CACM*, 11(13):677–678, 1970.

[7] B. Demoen, G. Engels, and P. Tarau. Issues in (copying) garbage collection for (Bin)WAM. Technical Report CW-202, K.U. Leuven, 1994.

[8] Int. Org. for Standardization. ISO/IEC 13211-1:1995(E). Information technology - Programming languages - Prolog - Part 1: General core. Geneva, Switzerland, 1995.

[9] J. Noyé. *Elagage de contexte, retour arrière superficiel, coupure et autres: une étude approfondie de la WAM.* Thèse de doctorat, 1275, Université de Rennes II, 1994.

[10] E. Pittomvils, M. Bruynooghe, and Y.D. Willems. Towards a real-time garbage collector for Prolog. In *Proceedings of 2nd Symposium on Logic Programming, IEEE*, 1985.

[11] O. Ridoux. MALIv06: Tutorial and Reference Manual. Publication Interne 611, IRISA, 1991. ftp://ftp.irisa.fr/local/lande.

[12] D. Sahlin. Making Garbage Collection Independent of the Amount of Garbage. SICS, 1987. Research Report SICS/R-87/87008.

[13] P. Tarau. Ecological Memory Management in a Continuation Passing Prolog Engine. In Y. Bekkers and J. Cohen, editors, *Memory Management International Workshop IWMM 92 Proceedings*, number 637 in Lecture

Notes in Computer Science, pages 344–356. Springer, Sept. 1992.

[14] P. Tarau. Language Issues and Programming Techniques in BinProlog. In D. Sacca, editor, *Proceedings of the GULP'93 Conference*, Gizzeria Lido, Italy, June 1993.

[15] P. Tarau. BinProlog 4.00 User Guide. Technical report, Département d'Informatique, Université de Moncton, 1995. Available by ftp from clement.info.umoncton.ca.

[16] P. Tarau and U. Neumerkel. A Novel Term Compression Scheme and Data Representation in the BinWAM. In *the Proceedings of PLILP'94, Madrid, M. Hermenegildo, J. Penjam*, pages 73–87. LNCS 844, 1994.

[17] H. Touati and T. Hama. A light-weight Prolog garbage collector. In *Proceedings of the International Conference on Fifth Generation Computing Systems*, pages 344–356, 1988.

[18] D. Warren. An Abstract Prolog Instruction Set. Technical report, SRI International, Artificial Intelligence Center, Aug. 1983.

[19] P. Weemeeuw. *Two Parallel Garbage Collection Algorithms for Prolog.* Department of Computer Science, PhD thesis, K.U. Leuven, May 1993.

[20] N.-F. Zhou. On the scheme of passing arguments in stack frames for Prolog. In *Proceedings of the Eleventh International Conference on Logic Programming*, pages 159–174. Santa Margherita Ligure, June 1988.

Appendix: Segment Preserving Copying GC Algorithm

The whole gc consists of a marking phase, a copy phase from *from* space, to *to* space, followed by a copy back from *from* to *to*. The first phase is performed top down (because of early reset) and is otherwise straightforward. The latter phase is just a block move (implemented by calling memcpy) because the copy phase makes sure that all pointers in the heap have been relocated already. We therefore only describe in some detail the copy phase itself.

Our algorithm distinguishes for each cell the boolean attributes: *reachable*, *internal* and *forwarded*. *Reachable* and *internal* are set during the marking phase, so 2 bits are needed. Everything which is internal is also reachable. The *forwarded* attribute is only relevant during the copy phase and of course only for reachable cells: it distinguishes cells that have been copied already from still to copy cells. After the *forwarded* attribute is set, the *reachable* and *internal* attribute can be turned off: the *reachable* or *internal* attribute of forwarded cells is never inspected. This means that forwarded can be made equivalent to not-reachable-nor-internal, so that 2 bits is enough for the three attributes. The additional advantage is that since every reachable cell is eventually forwarded, the reachable and internal bits are back to their original value after the copy phase: this is essential to achieve the theoretical linear complexity of a copying gc. Whether these bits are allocated in cells themselves or in a separate bit array is immaterial: the bit array (which must have size proportional to the size of the allocated heap) can be allocated (and set to all zero) once and (re)used during the life time of the heap.

The algorithm uses one global variable: `new_h`, which is the top of the new heap. It is initially set to the start of the *to* space.

```
copy()
{
  for each segment s from bottom to top do
    for each root pointer p in the local stack of s do
      if (! forwarded(p)) copy_cell(p) ;
      relocate(p) ;
      for each trail pointer p in s do
      if (! forwarded(p)) copy_cell(p) ;                (*)
      else
        if (pointer(*p) && to_newer_heap(*p)
                        && (! forwarded(*p)))
            { copy_cell(*p) ;
              relocate(*p) ;
            }
      relocate(p) ;
}

/* this is an adaptation of Cheney's algorithm */

copy_cell(p)
{ local scan ;
  scan = new_h ;
  copy_envelop(p) ;
  while (scan < new_h)
    { if pointer(scan)
        { if ((! forwarded(scan) &&
              (! to_newer_heap(scan)))
            copy_envelop(scan) ;
          relocate(p) ;
        }
      scan++ ;
    }
}

copy_envelop(p)
{ local l, u ;
  l = lower(p) ;
  u = upper(p) ;
  while (l < u) do
    { *new_h = *l ;
      setforwarded(l) ;
      l++ ; new_h ++ ;
    }
}
```

- `lower(p)` and `upper(p)`: determine the boundaries of the enveloping term of p which must be copied in one block
- `setforwarded(l)`: erases the mark bits for l and leaves behind a forwarding pointer as in Cheney's algorithm
- `forwarded(l)`: checks the mark bits
- `pointer(p)`: is true if the cell is a (possibly tagged) pointer to the heap
- `relocate(p)`: uses the forwarding pointer to give p its final value

`(*)` Note that the algorithm could do even better in terms of moving older data to a newer segment at this point: the copying of p could be postponed until the segment where p is accessible is treated; this would however involve too big a cost, because the trail entry itself would have to be moved; however, this is not in contradiction with the claims that our algorithm does move data to newer segments.