UPMAIL Technical Report No. 87
August 1994
ISSN 1100–0686

A Simple and Efficient Copying Garbage Collector for Prolog

Johan Bevemyr Thomas Lindgren

Box 311, S-751 05 Uppsala, Sweden
Phone: +48−18−18 25 00
Fax: +46−18−51 19 25

Abstract

We show how to implement efficient copying garbage collection for Prolog. We measure the efficiency of the collector compared to a standard mark-sweep algorithm on several programs. We then show how to accomodate generational garbage collection and Prolog primitives that make the implementation more difficult.

The resulting algorithms are simpler and more efficient than the standard mark-sweep method on a range of benchmarks. The total execution times of the benchmark programs are reduced by 4 to 11 percent.

INTRODUCTION

Automated storage reclamation for Prolog based on Warren's Abstract Machine (WAM) [14] has several difficulties. Let us consider the architecture of a typical WAM: most data are stored on a global stack (also called the heap), while choice points and environments are stored on a local stack (also referred to as the stack). A trail stack records bindings to be undone on backtracking. We will not consider garbage collection of code space in this paper, atom tables or other miscellaneous areas. There are no pointers from such tables into the garbage collected areas.

The WAM saves the state of the machine whenever a choice point is created. Using this information, stacks can be reset and storage reclaimed cheaply. We can view the global stack as composed of several segments, delimited by the choice point stack. Creating a new choice point creates a new segment; backtracking removes segments, while performing a cut merges segments. Data are allocated in the topmost segment, while variable bindings, which are implemented as assigning a cell representing the variable, are recorded on the trail stack whenever the variable cell is not in the topmost segment. When two variables are unified, a pointer from one cell to the other cell is created. In general, pointer chains may arise which require dereferencing.

A garbage collector for Prolog might thus retain the segment ordering to enable fast storage reclamation on failure by deallocating a segment allocated after the topmost choice point was created. Furthermore, since there are primitives (such as `@</2`) that compare variables, e.g., by creation time, most systems elect to preserve the heap ordering of data after garbage collection.

Garbage collection is done by starting at a set of *root pointers*, such as registers and the local stack, and discovering what data are reachable from these pointers, or *live*. Memory is reclaimed by compacting the live data [9], copying them to a new area [6] or putting the dead data on a free list. Memory allocation can then be resumed.

RELATED WORK

Prolog implementations such as SICStus Prolog use a mark-sweep algorithm that first marks the live data, then compacts the heap. We take the implementation of Appleby et al. [1] as typical. This algorithm works in four steps and is based on the Deutsch-Schorr-Waite [12, 7] algorithm for marking and on Morris' algorithm [9, 7] for compacting.

1. All live data are marked through roots found in registers, choice points, environments, and value trail entries (entries in the trail where the old value have been recorded, e.g., as a result of using `setarg/3`). A live tree is marked using a nonrecursive pointer-reversing algorithm that does not require any extra space to operate.

2. The registers, choice points, environments, and the trail are examined for references into the heap. All such references are put into reallocation chains, with the heap cell as root, to be updated when the heap cell is moved.

3. The heap is scanned upwards and all upward references are put into reallocation chains in order to be updated when the cell they refer to is moved.

4. The heap is scanned downwards and all marked data are moved to their new locations. All references to a moved object are found through the reallocation chains and updated. All references downward are also put into reallocation chains so that they may be updated when the object further down the heap is moved.

Touati and Hama [13] developed a generational copying garbage collector. The heap is split into an old and a new generation. Their algorithm uses copying when the new generation consists of the top most heap segment, i.e., no choice point is present in the new generation, and no troublesome primitives have been used (primitives that rely on a fixed heap ordering of variables). For the older generation they use a mark-sweep algorithm. The technique is similar to that described by Barklund and Millroth [4] and later by Older and Rummell [10].

We show how a simpler copying collector can be implemented, how the troublesome primitives can be accomodated better and how generational collection can be done in a simple and intuitive way. However, our view is also more radical than theirs. Where Touati and Hama still wish to retain properties such as memory recovery on backtracking, we take a more radical approach: ease of garbage collection is more important than recovering memory on backtracking. We show below that memory recovery by backtracking is still possible, and that the new approach in practice recovers approximately as much garbage by backtracking as the conventional approach.

Bekkers, Ridoux and Ungaro [5] describe an algorithm for copying garbage collector for Prolog. They observe that it is possible to reclaim garbage collected data on backtracking if copying starts at the oldest choice point (bottom-to-top). However, their method has several differences to ours.

- Their algorithm does not preserve the heap order, which means primitives such as `@</2` will work incorrectly. They do not indicate how this problem should be solved.

- Their algorithm (the version that incorporates early reset) copies data twice, while our algorithm visits data once and then copies the visited data. We think our approach leads to better locality of reference. However, we have not found any published measurements of the efficiency of the Bekkers-Ridoux-Ungaro algorithm.

- Variable shunting is used to avoid duplication of variables inside structures. This may introduce new variable chains, as shown in Appendix A. We want to avoid this situation.

Their algorithm does preserve the segment-structure of the heap (but not the ordering within a segment). Hence, they can reclaim all memory by backtracking. In contrast, our algorithm only supports partial reclamation of memory by backtracking. Our measurements indicate that this is sufficient: the copying algorithms we describe do not reclaim appreciably less memory on backtracking than the standard mark-sweep algorithm on the measured benchmarks.

Appel [2, 3] describes a simple generational garbage collector for Standard ML. The collector uses Cheney's garbage collection algorithm, which is the basis of our algorithm as well. However, his collector relies on assignments being infrequent. In Prolog, variable binding is assignment in this sense. Our algorithm handles frequent assignments efficiently.

Sahlin [11] has developed a method that makes the execution time of the Appleby et al. [1] algorithm proportional to the size of the live data. The main drawback of Sahlin's algorithm is that implementing the mark-sweep algorithm becomes more difficult, not to mention guaranteeing that there are no programming errors in its implementation. To our knowledge it has never been implemented. We also believe that Sahlin's algorithm is not as efficient as ours since it requires an extra pass over the live data, beyond the passes in the Appleby algorithm. Since our algorithm is almost 70 % faster than the Appleby algorithm even when the heap is filled with live data, it is unlikely that Sahlin's algorithm will be more efficient than ours.

ALGORITHM

We assume the standard term representation of WAM [14]. Our algorithm requires the existence of two tag bits for each cell on the heap, reserved for the use of the garbage collector. These tag bits may either be stored in each cell or in some separate area. One of the bits is used for marking copied cells as forwarded, the other is used for indicating that a cell appears inside a live structure.

Avoiding the heap ordering

Variables in the WAM are represented as self-referring heap cells. The WAM uses the location of a variable for deciding if trailing is required when binding the variable. Hence, variables should not move out of their heap segments. Since our algorithm does not preserve heap segments we must find another solution.

Our solution is simply to trail bindings of variables copied during the last garbage collection. The method to do this efficiently is described in Section "Recovering memory on backtracking". Bindings to the surviving variables from the topmost heap segment will be trailed unnecessarily (as compared to the

compacting approach), but other bindings will not be affected. The unnecessary trail entries are deleted by the next collection.

Mark-and-copy

The copying collector is a straightforward adaption of Cheney's algorithm [6] and works in three phases. The algorithm allows the standard optimizations of early reset. The old data reside in `fromspace` and are evacuated into `tospace`.

1. Mark the live data. When a structure is encountered, mark the functor cell and all internal cells. When a simple object is found, mark that cell only.
2. Copy the data using Cheney's breadth-first algorithm. When a marked cell is visited in `fromspace`, do the following:
   (a) Scan backward (towards lower addresses) until an unmarked cell is found.
   (b) Scan forward and evacuate marked cells into `tospace` until an unmarked cell is found. Overwrite the old cells with forwarding pointers to the corresponding cells in the copy.

   Thus, interior pointers are handled correctly. Several adjacent live objects may be evacuated at once. Continue until no cells remain to be evacuated.
3. Update the trail. If a trail entry does not refer to a copied cell (i.e., does not point at a forwarding pointer), it can be deleted. Implementing early reset is done by incorporating this step into the procedure that copies live data from the chain of choice points.

The collector thus visits (and writes) the data once, then writes the copy in `tospace`. We believe locality of reference to be quite good: in the second pass, the marked data will already reside in the cache if the data are sufficiently small.

If we do not mark all cells that occur inside live data structures then duplication of cells could occur. Suppose we have both a reference to a variable in a structure and a reference to the structure, e.g., see Figure 1. Suppose we copy the variable before the structure; then we would introduce an extra reference, e.g., see Figure 2. This is undesirable since the result of doing garbage collection might then be that more space is required! To solve this, we mark all cells that occur in live data structures. When a marked cell is copied the enclosing structure is also copied by step 2.

Recovering memory on backtracking

A compacting collector preserves the heap segments (see Figure 3) and entire segments can be deallocated on backtracking. In general, a copying algorithm cannot recover memory on backtracking since the heap no longer preserves the

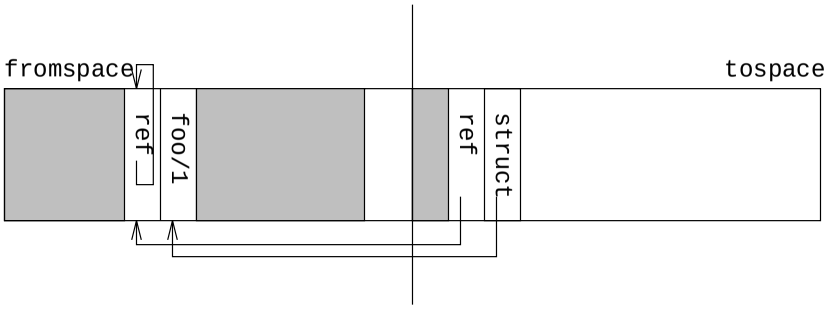


Figure 1: An internal cell is referred to twice, both directly by a variable and indirecly through a structure.

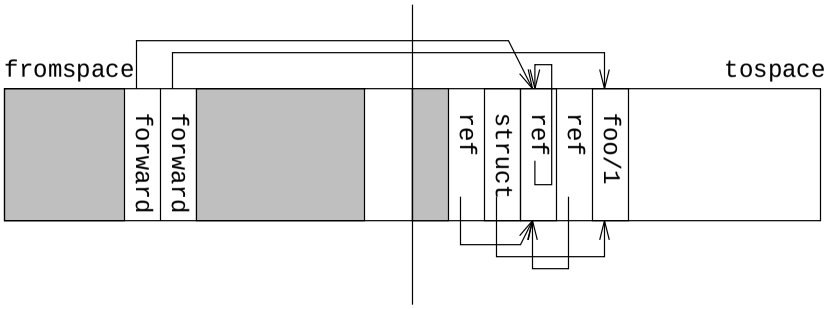


Figure 2: As a result of copying without marking internal cells some cells might be duplicated.

required stack ordering. However, our algorithm can still recover some garbage by resetting the heap pointer, just as in a standard WAM implementation.

We note that *between* collections, memory is allocated just as in a WAM. Using this observation, we can arrange to recover memory allocated after the last collection on backtracking. After a collection, we set the saved heap top of all choice points to the top of the heap space, making this one segment (see Figure 4). Terms allocated after this segment can be reclaimed upon backtracking. In this way, we retain most of the advantages of resetting the heap upon backtracking without having to tailor our runtime system and garbage collector to ensure this property at every point. Precisely the same test for trailing a binding can be used as in a standard WAM. Collection may also split a single segment into two, which leads to extra trailing. We measure the efficiency of our system below.

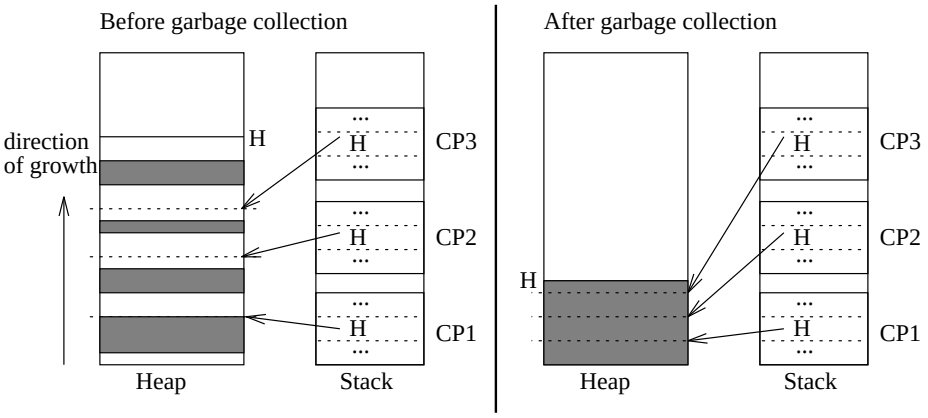


Figure 3: Saved heap top pointers (H) in choice points before and after **compacting** garbage collection. Heap segments are preserved.

Handling troublesome primitives

Touati and Hama recognized that the generic comparison operators, such as `@</2`, required that variables preserve their relative ordering (since variable ordering is usually the comparison criterion when two unbound variables are compared). Their method was either to disable copying collection when these primitives were used and revert to using a compacting collector, or to generate identifiers for variables when needed. The identifiers were to be kept in a hash table to be updated when variables were relocated.

Others have proposed to associate a creation time with each variable. Our experience is that timestamps add a runtime overhead of approximately 5% in an emulator-based WAM implementation.

Our solution retains the use of copying collection, while requiring a small modification to the runtime system. When variables are compared, we arbitrarily order them if unordered. This is done by binding unordered variable cells to new variable cells on a small *compared-variable* stack (cv-stack), see Figure 5. Once this is done, we can just compare addresses. The binding is not trailed. (An unbound variable is unordered if it does not reside on the cv-stack and ordered if it is on the cv-stack.)

Once a variable is ordered it resides on the cv-stack. Subsequent unordered variables will be 'greater' than ordered variables when compared, since they are pushed on the cv-stack (where the ordering is kept) when the comparison occurs. Using compacting collection on the cv-stack retains the variable ordering (though the copying collector must take care not to migrate variables residing on the cv-stack), while dead variables disappear.

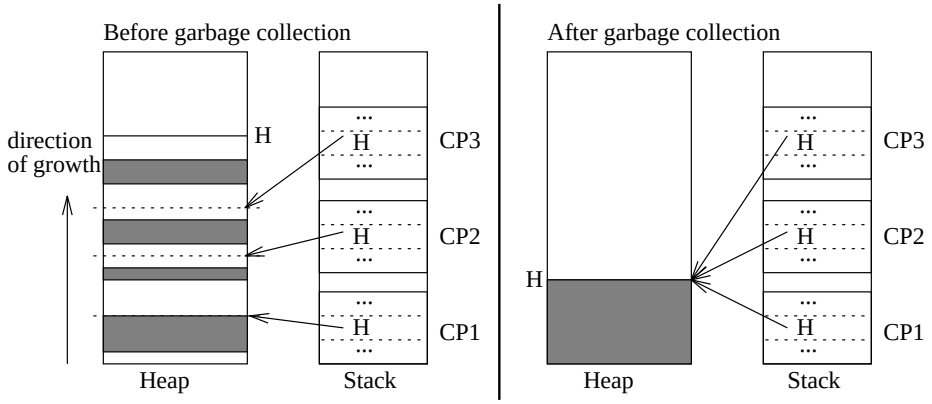


Figure 4: Saved heap top pointers (H) in choice points before and after **copying** garbage collection. After garbage collection all segments have been merged into one single segment. Note that only the active heap is shown here—the live data have been copied from the old heap to the new heap.

The space cost is proportional to the number of unbound variables compared by generic comparison operations. Naturally, if most of the live unbound variables have been compared in this way, the collector will have to spend more time in compacting the cv-stack. We believe this situation to be rare.

INTRODUCING GENERATIONAL GARBAGE COLLECTION

Generational garbage collection [8, 3] relies on the observation that newly created objects tend to be short-lived. Thus, garbage collection should concentrate on recently created data. The heap is split into two or more generations, and the most recent generation is collected most frequently. When the youngest generation fills up, a collection spanning more generations is done, and the survivors move to the oldest of these generations. Frequently, implementations have two generations, and we will assume so from now on.

The difference from standard copying collection is that the collection roots also include the pointers from the older to the younger generation. In languages such as SML, most objects are immutable, and assignments that may cause cross-generational pointers can be compiled to special code that registers such a pointer if it appears.

In Prolog, there is a high incidence of assigning already created objects, so such a solution is likely to be expensive. Variable bindings involve assignment. In fact, we can arrange so that only *trailed* bindings may be cross-generational, by setting the limit where trailing occurs appropriately, see Figure 6. Usually, this limit is the start of the topmost heap segment, but this is not required: we can set it in the interior of the topmost segment at the cost of doing unnecessary

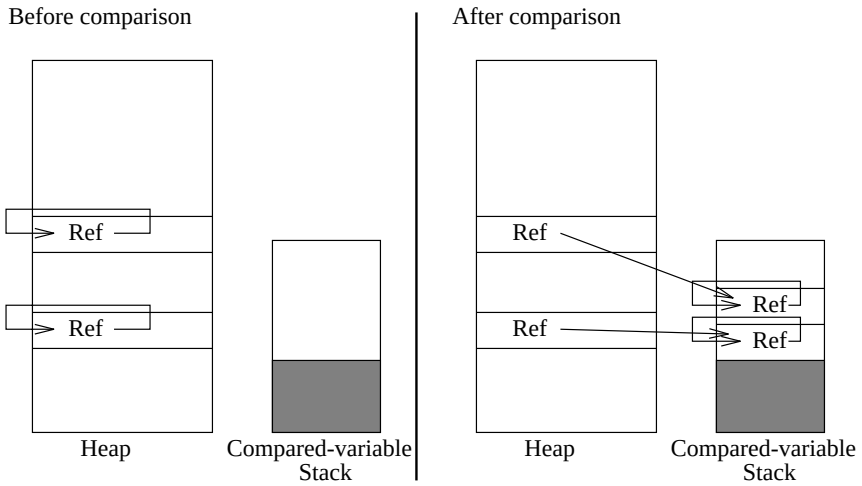


Figure 5: Compared variables are bound to variables on a compared-variable stack. This way their relative order is preserved.

trailings. This may be useful if the computation is deterministic but still has a large amount of live data.

We assume that objects are tenured (moved to the old generation) if they survive a collection of the new generation. Now, we can find cross-generation pointers by examining all new trail entries since the last garbage collection. These pointers point out root pointers from the older generation. Since the tenuring threshold is one, the old part of the trail need not be scanned; it can refer only to the old generation. If we were to allow several minor collections before tenuring, the cross generation pointers must be recorded for subsequent use, to avoid scanning the trail repeatedly.

In other languages it is usually necessary to add a *write barrier*, code that detects cross generational bindings and record them on a stack. This result in a runtime cost for using generational garbage collection. In Prolog this overhead is already present in the form of trail tests and there is no extra runtime penalty for using generational collection.

EVALUATION

We have implemented a standard mark-sweep algorithm [1] and compared it to our copying algorithms. All garbage collection algorithms have been implemented in the same system, a sequential version of Reform Prolog. All algorithms implement early reset.

The TSP program implements an approximation algorithm for the Travel-

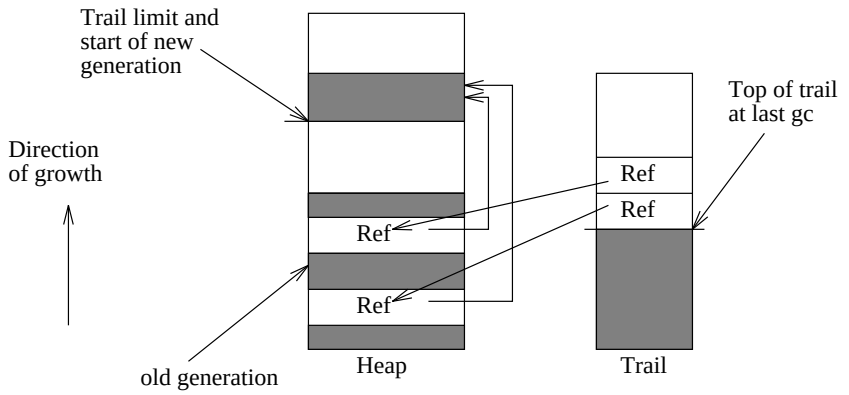


Figure 6: The limit for trailing is set in such a way that all cross-generational references are recorded on the trail. In the presence of a choice point in the new generation the trail limit is set as usual.

ling Salesman Problem. A tour of 60 cities was computed. The MATCH program implements a dynamic programming algorithm for comparing, e.g., DNA-sequences. One sequence of length 32 was compared to 100 other sequences. BOYER is the Boyer-Moore theorem prover adapted by Evan Tick to Prolog. This program is part of the Berkeley benchmark suite.

BIG is a program that allocates a large data structure and then forces garbage collection 1000 times while the data structure is still live. The program is intended to compare the copying collectors with mark-sweep when the heap is filled with live data, i.e., when copying and compaction are working on a similar amount of memory and data.

When executing the TSP, MATCH, and BOYER programs we gave each version of the emulator (mark-sweep and copy) an equal amount of memory. This meant that the emulator using copying garbage collection used a heap that was half the size of the mark-sweep heap. One might argue that since modern computers have virtual memory it is reasonable to let the copy version temporarily allocate twice as much space as the mark-sweep version. However, we have found that even the size of the virtual memory (actually the size of the swap space) can be a limiting factor.

In our measurements of the generational collector, the new generation is given half the remaining free memory in the semi-space, after a minor collection. When the size of the new generation is less than 20K, a major collection is done.

When we executed the BIG program we gave the copying collectors twice the memory of the mark-sweep collector so that the collected data structures could have the same size. The from-space and mark-sweep heap thus were of the same size. This was done in order to compare the timing of a single collection, rather

than the entire execution.

All times are in seconds user time.

Program	Heap size	Mark-Sweep gc	Mark-Sweep run time	Copy gc	Copy run time	Gen. Copy gc	Gen. Copy run time
MATCH	750K	12.0%	22.02	2.6%	20.01	0.6%	19.82
	1500K	11.2%	21.89	1.4%	19.70	0.3%	19.64
	2250K	10.4%	21.43	0.7%	19.96	0.1%	19.87
	3000K	10.0%	21.64	0.6%	19.89	0.05%	19.61
TSP	750K	5.3%	54.73	0.9%	51.99	0.1%	51.11
	1500K	4.8%	54.62	0.4%	52.12	0.1%	51.81
	2250K	4.7%	53.75	0.3%	51.96	0.08%	51.48
	3000K	4.6%	55.38	0.2%	51.77	0.07%	51.65
BOYER	750K	15.8%	4.61	15.7%	4.51	8.4%	4.38
	1500K	8.7%	4.25	7.1%	4.23	5.6%	4.09
	2250K	12.4%	4.58	5.9%	4.04	4.9%	4.06
	3000K	0%	4.04	3.0%	3.98	4.9%	4.02
BIG	1500K	100%	11.22	100%	6.34	—	—

The next table shows the difference in execution times for the three algorithms (using 1500K memory). The improvement of the total execution time when using a copying collector ranges from 4 to 11 percent.

We also measured how many times the different garbage collectors were invoked.

Program	Heap size	Number of garbage collections performed Mark-Sweep	Copy	Gen. Copy
MATCH	750K	25	51	130
	1500K	12	25	48
	2250K	8	16	30
	3000K	6	12	22
TSP	750K	22	46	113
	1500K	11	22	51
	2250K	7	14	33
	3000K	5	11	25
BOYER	750K	3	8	30
	1500K	1	3	7
	2250K	1	2	4
	3000K	0	0	3

All times are in milliseconds user time.

Program	GC time/run time		
	Mark-Sweep	Copy	Generational Copy
MATCH	2460/21890	280/19700	50/19640
TSP	2600/54620	220/52120	50/51810
BOYER	370/4250	300/4230	230/4090

Note that the total execution times are sometimes shorter when garbage collection is performed. We believe this to be due to improved data locality due to copying.

Program	Memory allocated	Memory reclaimed on backtracking		
		Mark-Sweep	Copy	Generational
MATCH	17770K	793 (4%)	793 (4%)	793 (4%)
TSP	18895K	0 (0%)	0 (0%)	0 (0%)
BOYER	4187K	1798 (43%)	1798 (43%)	1798 (43%)

Approximately the same amount of data is reclaimed on backtracking with all three algorithms. We believe the reason for this is that most of the memory is reclaimed during shallow backtracking.

We also measured the amount of extra trailing imposed by the copying collectors.

Program	Number of trail entries			Ratio	
	Mark-Sweep	Copy	Gen.	Copy/Mark	Gen/Mark
MATCH	112582	112767	112864	1.0016	1.0025
TSP	10560	10568	10568	1.0008	1.0008
BOYER	108352	108450	108441	1.0009	1.0008

Clearly, the extra trailing performed by the copying algorithms is insignificant compared with the total amount of trailing in our benchmark programs. The above measurements were made using a heap size of 750K bytes.

CONCLUSION

We have described a method for adapting conventional copying garbage collection to Prolog and how to add generational collection to this algorithm. Three problems have been solved, leading to efficient copying and generational copying collectors.

The first problem is interior pointers, which can lead to duplication of data if copied naively. Our method correctly handles interior pointers by marking, then copying data.

The second problem is that copying collection does not preserve the heap ordering. In theory, this means memory cannot be reclaimed by backtracking, and that bindings in the copied area must always be trailed (rather than occasionally).

Our collector exploits that data allocated since the last collection still retain the desired heap ordering. Hence, memory allocated after the last collection can still be reclaimed by backtracking. Our measurements show that our copying algorithm recovers as much memory by backtracking as a conventional ("perfect") mark-sweep algorithm on a range of realistic benchmarks.

We have also measured the amount of extra trailing due to losing the order of the heap. This was negligible: less than one-quarter of a percent of the total number of trailings at most. We conclude that copying collection is a viable alternative to the conventional mark-sweep algorithm for Prolog.

Finally, we also showed how to extend the copying algorithm to generational collection. The crucial insight is that pointers from the old generation (in a two-generation system) can be found by scanning the trail. By adapting the trailing mechanism, we get an almost-free write-barrier. The only extra cost is some unnecessary trailings in certain situations. This cost is again negligible for our benchmarks.

ACKNOWLEDGMENT

We thank Ulrich Neumerkel for his comments on this paper, in particular on whether our method to deal with troublesome primitives was correct. We also thank Mikael Pettersson and the anonymous referees for their comments.

REFERENCES

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

2. A.W. Appel, A runtime system, *Lisp and Symbolic Computation*, 3(4), 1990.

3. A.W. Appel, Simple generational garbage collection and fast allocation, *Software—Practice and Experience*,19(2):171–183, 1989.

4. J. Barklund, H. Millroth, Garbage cut for garbage collection of iterative Prolog programs, *3rd Symposium on Logic Programming*, Salt Lake City, September 1986, IEEE.

5. Y. Bekkers, O. Ridoux and L. Ungaro, Dynamic Memory Management for Sequential Logic Programming Languages, *Proceedings of the International Workshop on Memory Management 92*, LNCS 637, Springer-Verlag, Berlin, 1992.

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

7. J. Cohen, Garbage Collection of Linked Data Structure, *Computing Surveys*, 13(3):341–367, September 1981.

8. H. Lieberman, C. Hewitt, A real-time garbage collector based on the lifetimes of objects, *Communications of the ACM*, 26(6):419–429, June 1983.

9. F. Morris, A Time- and Space- Efficient Compaction Algorithm, *Communications of the ACM*, 12(9):662–665, August 1978.

10. W.J. Older and J.A. Rummell, An Incremental Garbage Collector for WAM-Based Prolog, *Proceedings of the Joint International Conference and Symposium on Logic Programming*, MIT Press, Cambridge, Mass., 1992.

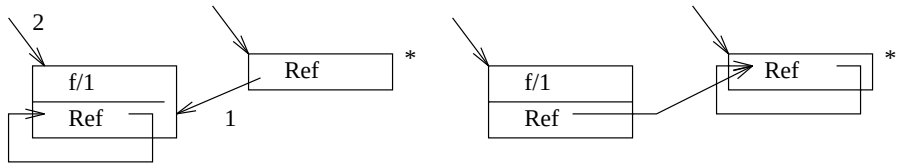
11. D. Sahlin, Making garbage collection independent of the amount of garbage, Research Report R87008, Swedish Institute of Computer Science, 1987.

12. H. Schorr and W.M. Waite, An Efficient Machine-Independent Procedure for Garbage Collection in Various List Structures, *Communications of the ACM*, 10(8):501–506, August 1967.

13. H. Touati, T. Hama, A light-weight prolog garbage collector, *Proceedings of the International Conference on Fifth Generation Computing Systems*, 1988.

14. D.H.D. Warren, An Abstract Prolog Instruction Set, SRI Tech. Note 309, SRI International, Menlo Park, Calif., USA, 1983.

APPENDIX A



Before copying

After copying

Variable shunting collapses a chain of pointers into a single cell when they are all in the same segment. Assume that pointer (1) in the figure is copied first. The chain of two pointers is collapsed into a single cell $*$ due to variable shunting. Subsequently, reference (2) copies the structure, which yields the situation shown after copying. The number of cells remains constant, but a new reference chain has appeared.