Heap Garbage Collection in XSB: Practice and Experience

Bart Demoen[1] and Konstantinos Sagonas[2]

[1] Department of Computer Science, K.U. Leuven, Belgium
e-mail: `bmd@cs.kuleuven.ac.be`
[2] Computing Science Department, Uppsala University, Sweden
e-mail: `kostis@csd.uu.se`

Abstract. Starting from a theoretical understanding of the usefulness logic of a logic programming system with built-in tabling, and from a collector that did not take the characteristics of a tabled abstract machine into account we have build two heap garbage collectors (one mark&slide, one mark&copy) for XSB on top of the CHAT implementation model for the suspension/resumption of consumers. Based on this experience we discuss implementation issues that are general to heap garbage collection for the WAM and also issues that are specific to an implementation with tabling: as such, this paper documents our own implementation and can serve as guidance for anyone attempting a similar feat. We report on the behaviour of the garbage collectors on different kinds of programs. We also present figures on the extent of internal fragmentation and the effectiveness of early reset in Prolog systems with and without tabling.

1 Introduction

In September 1996, we started developing a heap garbage collector for XSB. A mark&slide collector was written, closely following previous experience (see also [1]): it worked as long as tabling was not used. When trying to extend it for tabled programs, we failed to understand the *usefulness logic* (see [2]) of tabling systems. In particular, we could not get a grasp on *early reset* (also known as *virtual backtracking*; see e.g. [12, 2]) in the context of suspended computations. At that point we could have decided to make the garbage collector more conservative — i.e. leave out early reset and just consider everything pointer reachable as useful. This would have been completely acceptable from an engineering perspective but appeared to us very unsatisfactory from a scientific point of view. So we abandoned the work on the garbage collector and concentrated (more than a year later) on alternative ways for implementing suspension/resumption of consumers. This resulted in the 'Copying Approach to Tabling' (abbrv. CAT [7]): this implementation schema lead directly to a better understanding of the usefulness logic of logic programming systems with tabling (see [8]). Armed with this theoretical understanding we resumed work on a more accurate garbage collector in August 1998, finishing the sliding collector and at the same time implementing a copying collector as well: the copying collector uses the same marking phase as the sliding one (see [3] for why a copying collector in the WAM needs a marking phase) so it was relatively little additional work. Still, we struggled a lot with technical details and misunderstandings of the invariants of the tabling run time data structures. We finally integrated our garbage collectors in the XSB system in February 1999 and since June 1999 they are part of the XSB release.

This paper reports issues that came up, real problems that occurred, their solutions, decisions we took and why. These are the contents of Sections 3 and 4. Some of these issues may seem trivial (especially in retrospect) but most of them were learned the hard way, i.e. by debugging. We thus think that our experience on the practical aspects of building a garbage collector is of interest and use to other declarative programming language implementors and may even serve as a warning to anyone attempting to write a garbage collector for a system that was not designed to have one, and even more to anyone designing a system without thinking about its proper memory management. We include some performance figures about our collectors in Section 5. Finally, Section 6 discusses memory fragmentation both with and without early reset: to the best of our knowledge, figures related to these issues have never before been published for any Prolog system — let alone a tabled one — and as such they are of independent interest. We start by briefly reviewing the internals of tabled abstract machines.

2 Memory organization in tabled abstract machines

The implementation of tabling on top of the WAM [19] is complicated by the inherent asynchrony of answer generation and consumption, or in other words the support for a *suspension/resumption* mechanism that tabling requires. The need to suspend and resume computations is a main issue in a tabling implementation because some tabled subgoals, called *generators*, use program clauses to generate answers that go into the tables, while other subgoals, called *consumers*, resolve against the answers from the tables that generators fill. As soon as a generator depends on a consumer, the consumer must be able to suspend and work in a coroutining fashion with the generator, something that is not readily possible in the WAM because it reclaims space on backtracking. In short, in a tabled implementation, the execution environments of suspended computations must somehow be preserved and reinstated. By now, several alternatives for suspension/resumption exist: either by totally sharing the execution environments by interspersing them in the WAM stacks (as in the SLG-WAM [14]), or by a total copying approach (as in CAT [7]), or by a hybrid approach (as in CHAT [9]). In this paper, we stick to a CHAT implementation, and refer the interested reader to the above references for differences between these abstract machines [1].

Independently of the implementation model chosen for the suspension/resumption mechanism, tabling calls for sophisticated memory management. Indeed, tabling systems have inherently more complex memory models and their space requirements are different from (usually bigger than) those of plain Prolog systems. As advocated in e.g. [2], the accuracy of memory management is not related to the underlying abstract machine or the garbage collection technique; instead

[1] All relevant papers are accessible at `http://www.csd.uu.se/~kostis/Papers/`.

it is related to the *usefulness logic* of the run-time system: an abstraction of the operational semantics of the language, or in other words the ability to decide which objects are useful and which are garbage. In [8], we have described the usefulness logic of Prolog systems with tabling and how operations such as early reset can in principle be implemented with equal accuracy in an SLG-WAM or in a CAT-based abstract machine. As our aim is to discuss garbage collection issues in the context of tabled execution of logic programs, we only review information from [8, 9] which is necessary to make this paper reasonably self-contained.

We concentrate on the following typical scenario: The memory management policy has decided that the heap better be garbage collected. The garbage collection process then consists of 1) finding the set of useful data in the heap, and 2) move it appropriately while adapting all pointers to it. The second point is a matter of choosing an appropriate collection algorithm: we have written a sliding collector based on Morris' algorithm [11] and one based on the copying algorithm of Cheney [5]. In the context of Prolog, both need a marking phase and that is precisely the issue of the first point: how to approximate the usefulness of data.

Fig. 1 shows a rough picture[2] of a complete snapshot of the memory areas of a CHAT implementation. The left part of the picture, identified as 'Prolog', shows the usual WAM areas in an implementation that stores environments and choice points separately, such as in SICStus or in XSB; besides this, the only difference with the WAM is that the choice point stack contains possibly more than one kind of choice points: regular WAM ones, P_1, P_2, P_3 for executing non-tabled predicates, choice points for tabled generators, G_1, G_2, and choice points of consumers, C_1. The 'Prolog' part reduces to *exactly* the WAM if no tabled execution has occurred; in particular, the trail here is the WAM trail. The

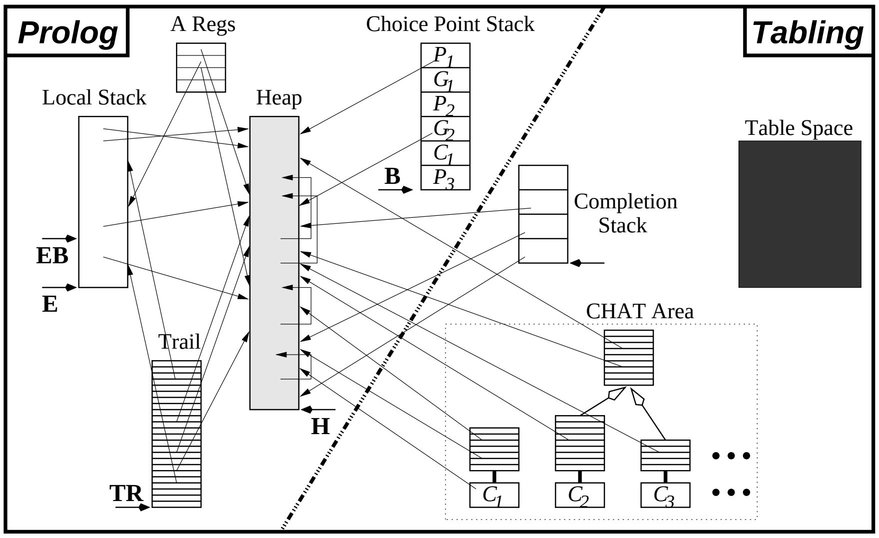

Fig. 1. Memory snapshot of a CHAT-based abstract machine when GC is triggered.

[2] In figures, relative size of memory areas is not significant. All stacks grow downwards.

right part of the picture, identified as 'Tabling', shows areas that CHAT adds when tabled execution takes place. The picture shows all memory areas that can possibly point to the heap; areas that remain unaffected by garbage collection such as the Table Space are not of interest here and are thus shown as a black box. For the 'Prolog' part, garbage collection techniques are standard and well-described in the literature; see e.g. [1, 3, 6]. We concentrate on the memory areas of the 'Tabling' part and the choice point stack that differs from the WAM.

As the memory snapshot shows, the evaluation involved consumers, some of which, e.g. $C_2, C_3, \ldots$ are currently suspended (appear only in the CHAT area; see below) and some others, like C_1, have had their execution state reinstalled in the stacks and are part of the active computation. Complete knowledge of CHAT is not required; however, it is important to see how CHAT has arrived in this state and, more importantly, how execution might continue after collection. We thus describe the actions and memory organization of a CHAT-based abstract machine viewed from a heap garbage collection perspective.

Choice points & Completion stack CHAT, much like the WAM, uses the choice point stack as a scheduling stack: the youngest choice point determines the action to be performed upon failure. A Prolog choice point, P, is popped off the stack when the last program clause of the associated goal is triggered. A generator choice point, G, is always created for a new tabled subgoal and behaves as a Prolog choice point with the following exception: before being popped off the stack, G must resume all consumers with unresolved answers that have their execution state protected by G (as explained below). Only when no more such consumers exist, is G popped off the stack. As far as the choice point stack and the heap are concerned, resumption of a consumer means: 1) re-installation of the consumer choice point C immediately below G and 2) setting C[H] — the H field of C — to G[H] so that C does not reclaim any heap that G protects (e.g. in Fig. 2, G_2 and C_1 protect the same heap: from the top till the dotted line). Finally, a consumer choice point C is pushed onto the stack either when the consumer is reinstalled by a generator, or the first time that the consumer is encountered. The consumer is popped off the stack and gets suspended whenever it has resolved against all answers currently in the table. Besides the H fields of choice points, pointers from the choice point stack to the heap exist in the argument registers stored in choice points used for program clause resolution (the darker areas above G's and P's in Fig. 2).

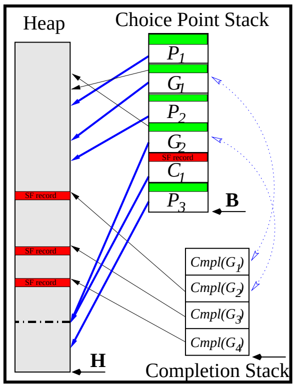

Fig. 2. Detail of choice point & completion stack.

The implementation of tabling becomes more complicated when there exist mutual dependencies between subgoals. The execution environments of the associated generators, $G_1, \ldots, G_n$, which reside in the WAM stacks need to be preserved until all answers for these subgoals have been derived. Furthermore, memory reclamation upon backtracking out of a generator choice point G_i cannot happen as in the WAM; trail and choice point stack can be reclaimed but e.g. the heap cannot: $G_i[H]$ may be protecting heap space of still suspended computations; not just its own heap. To determine whether space reclamation can be performed, subgoal dependencies have to be taken into account. The latter is the purpose of the *Completion Stack*. In Fig. 2, the situation depicted is as follows: subgoals associated with generators G_3 and G_4 cannot be completed independently of that of G_2. G_3 and G_4 have exhausted program clause resolution and the portion of the choice point associated with this operation can be reclaimed; however, there is information about G_3 and G_4 that needs to survive backtracking and this information has been preserved in the completion stack. In other words, generators consist of two parts: one in the choice point stack and one in the completion stack; for G_1 and G_2 that are still in the choice point stack the association between these parts is shown by dotted lines in Fig. 2.

Substitution factors Subgoals and their answers usually share some subterms. For each subgoal only the substitutions of its variables need to be stored in the table to reconstruct its answers. XSB implements this optimization through an operation called *substitution factoring*. On encountering a generator G, the dereferenced variables of the subgoal (found in the argument registers of G) are stored in a substitution factor record SF. For generators, CHAT stores substitution factor records on the heap. The reason: SF is conceptually part of the environment as it needs to be accessed at the 'return' point of each tabled clause (i.e. when a new answer is potentially derived and inserted in the table). Thus, SF needs to be accessible from a part of G that survives backtracking: in CHAT a cell of each completion stack frame points to SF; for consumers the substitution factor can be part of the consumer choice point as described in [13]; see Fig. 2.

CHAT area The first time that a consumer gets suspended, CHAT protects its execution state as follows: a *CHAT freeze* mechanism gets invoked which modifies H and EB fields in some choice points in a way that ensures that parts of the heap and local stack that the consumer might need for its proper resumption are not reclaimed on backtracking (see [9] for further explanation). As it is not possible to protect the consumer choice point C and the trail needed for resumption of the consumer using the same mechanism, these areas are saved using copying to the *CHAT area*. The copying of C (together with its SF record) is immediate, while the relevant entries of the trail (and their values in the heap and local stack) that C requires for its resumption are copied incrementally. In this way, trail portions common to a set of consumers are shared between them. For example, consumers C_2 and C_3 in Fig. 3 share the CHAT trail area CTR_4 while they each also have a private part of trail. The same figure shows which are the pointers from the CHAT sub-areas to the heap that need to be followed for marking and pos-

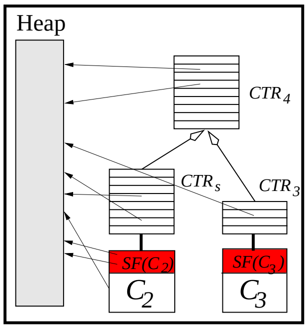

Fig. 3. CHAT area detail.

sible relocation: they are the trail values of the *CTR* sub-areas, substitution factors of suspended consumers and the C[D] field (the value of delay register as saved in each choice point; see [15]). The following things are important to note here: 1) the CHAT sub-areas are allocated dynamically and in non-contiguous space; how this influences garbage collection is described in Section 4.3, and 2) in CHAT, suspended computations have parts of their execution state saved in a private area while other parts are either shared or interspersed in the WAM stacks together with the state of the active computation.

3 Implementing heap garbage collection: general issues

We first discuss issues that are relevant to heap garbage collection in any WAM-based implementation: some or maybe all are folklore, but have not been published before as far as we know [3].

3.1 Dealing with unitialized environment variables

The WAM does not need to initialize permanent variables (in the local stack) on allocation of an environment, because its instruction set is specialized for the first occurrence of a variable. On the other hand, some Prolog systems (e.g. SICStus; see also [4]) do initialize some permanent variables just for the sake of garbage collection. This makes the marking phase more precise; the alternative is a conservative marking schema which follows cautiously all heap pointers from an environment whether from an active permanent variable or not. Indeed, XSB, as most Prolog systems (with MasterProlog a notable exception), have no explicit information about which permanent variables are alive at a particular moment in the computation: this information is only implicit in the WAM code. We thus faced a choice between the following options, given in increasing order of difficulty of implementation:

1. initialize environment variables (e.g. to unbound or to an integer)
2. write a more cautious and conservative marking phase
3. introduce more preciseness about the liveness of permanent variables.

We have opted for the first solution because it was the quickest to implement by extending the allocate instruction: the extra cost is usually quite low.

3.2 The test for heap overflow

XSB, like many other logic programming systems, relies on software tests for overflow of all its stacks: heap overfow was checked on every increase of the **H** register. A more efficient way to check for overflow is at call ports, either by extending the call and execute instructions or by introducing a new instruction, test_heap, which the compiler generates as the entry point of every predicate. Its two arguments are: 1) the arity of the predicate which is needed for the garbage collection marking phase, and 2) the margin of heap that has to be available before proceeding: this margin depends on how much heap the predicate can maximally consume during the execution of its heads and first chunks.

[3] We would be glad to give credit to people who have originally noted similar things.

For the first cut of the garbage collector, the XSB compiler was adapted so as to spit out the instruction test_heap with a fixed margin. A related change is needed in the implementation of built-in predicates: they should not commit any changes until they are sure to have enough heap space and if not, call the collector. Also, if built-in predicates that consume heap are inlined, a variant of the test_heap instruction might be needed in the body of clauses; marking needs to have access to the length of the current environment. A similar instruction is needed before inlined built-ins that create choice points. In XSB such an inlined built-in, called `negation_suspend`, was used for the implementation of tabled negation. As its inlining was not time critical, we have simply disabled it.

3.3 H fields in choice points

In a plain Prolog garbage collector, after a segment of computation — starting from the E and CP fields of a choice point — has been marked, as well as the trail, the H field of the choice point can be treated. At that moment, the H field can point to a cell which was marked already and no further action is required. Otherwise, H points to a non-marked cell and then two ways of dealing with this situation are common practice: (1) mark the cell and fill it with an atomic object; (2) decrease the H field until it points to a marked cell. The first method is simple, has constant time cost, and can waste at most a number of heap entries equal to the number of active choice points. The second method wastes no space, but in the worst case adds a time cost to the garbage collection that is linear in the size of the heap. We prefer the first method and in our implementation used the tagged integer 666 as the atomic object. The correctness of this operation in plain WAM is based on the observation that the action `if (!marked(B[H])) {*B[H] = tagged_int(666); mark(B[H]);}` can be performed as soon as it is sure that *B[H] will not be marked later.

Note that the H fields in the consumer choice points in the CHAT area need not be considered during heap garbage collection: indeed, when the consumer is reinstalled, its choice point will get its H field from the scheduling generator. Following [8] the suspended computations are marked after the active computation. Since the substitution variables are *never* reachable from the active computation (only from the frames in the completion stack), in a CHAT garbage collector the above action needs to be postponed until after the substitution variables are marked. Instead of trying to find the earliest possible moment to perform the action, we have opted for postponing it until the very end of the marking, i.e. to perform it after all other marking has been finished.

3.4 Trail compaction during garbage collection

Trail compaction is possible whenever early reset is performed (cf. Section 4.1). When trail compaction is omitted, one must ensure that trail cells that could have been removed point to something "reasonable". One can reserve a particular cell on the start of the heap for this purpose; i.e. all trail entries that are subject to early reset can be made to point to this heap cell [4]. Since reserving such a cell

[4] This trick seems folklore and was once described in `comp.lang.prolog` by R.A. O'Keefe.

would involve changes elsewhere in XSB, we have chosen to mark the cell that was reset; i.e. consider it as non-garbage. Although this diminishes somewhat the effect of early reset in the actual implementation, it has no influence on the results of Section 6 as in our measurements we counted these cells as garbage.

3.5 Chain bits for dynamically allocated root pointers

The XSB cell representation in one machine word cannot cater for the extra bits needed by the garbage collector. In particular, in the mark&slide collector we were faced with the problem where to store the chain bit for the root pointers: as long as these are in easily recognizable and contiguous areas — like the WAM stacks — one can allocate a parallel bit array and the translation from a root pointer to its corresponding bit is straightforward. For small sets of root pointers, just copying them to the top of the heap (as is the treatment of e.g. the argument registers) is also a good solution, but for a large number of root pointers that are not necessarily in a contiguous area, the solution must be more generic. In CHAT we encounter such a situation as the CHAT trail chunks are allocated dynamically. Section 4.3 discusses in more detail how we dealt with this issue.

3.6 The copying collector

The copying collector was implemented following ideas of [3]. We deviate slightly from the usual two-space schema of Cheney [5], since after the marking phase we know exactly how large the *to*-space needs to be to hold the copy so we allocate at that moment just this amount. After having copied the non-garbage to the *to*-space, we copy it as a block back to the original *from*-space and release the allocated *to*-space. This has a principled reason: this scheme uses memory resources more economically because usually, the *to*-space can be quite a bit smaller than the *from*-space. It has however also a pragmatic reason: in the current memory model of XSB, the heap and local stack are allocated contiguously and are growing towards each-other; putting the heap in another region that is not just above the local stack would break some invariants of XSB. Copying back the *to*-space to the original heap has low cost, as was observed already in [6].

4 Tabling-specific issues of heap garbage collection

The usefulness logic of tabled evaluation [8] dictates that both the current computation (together with its backtracking states) and the suspended computations in the CHAT area should be used as a *root set*. As argued in [8] for the SLG-WAM, one can perform the marking phase of garbage collection *without* reinstalling the execution states of the suspended computations on the stacks. The same reasoning applies to CHAT. Moreover, the marking phase of a suspended consumer C does not need to consider the part of the heap that is older than the generator G up to which C has its execution environment CHAT-protected, and neither any choice point between **B** (WAM top of choice point stack) register and G; see [8] on why this scheme is correct in an abstract machine that preserves consumer choice points by copying. It follows that multiple traversal of the same areas is not needed and that garbage collection can be implemented efficiently in the tabled abstract machines we consider here.

Armed with this theoretical understanding, we proceeded with the actual implementation of our garbage collectors only to stumble very quickly on technical issues the theory did not immediately cater for. The remaining part of this section presents these issues, together with their solutions as implemented. The first two issues are related to the order of marking the current and the suspended computations in the presence of early reset. The last two we encountered for the first time in the tabling context, but they are more general.

4.1 Performing early reset when trail chunks are shared

The idea of early reset in the WAM [12, 2] is that a trailed heap or local stack entry which is not reachable for the forward continuation of the active computation (but might be on backtracking) can be set to unbound during garbage collection. The situation is recognized during marking, and it is essential that the continuation is marked before the future alternatives. Early reset in the context of tabling is more complicated: the suspended computations must also be taken into account. [8] describes in detail why it is better to mark the consumers in the CHAT area *after* the marking of the current computation.

However, even the order of marking and performing early reset among suspended consumers matters ! In the WAM, the trail is segmented according to choice points and trail chunks are not shared: the trail is a stack, not a tree. As Fig. 3 shows, in CHAT trail entry chunks are possibly shared between more than one suspended consumer choice points. The same is true in both SLG-WAM and CAT. In such a situation it is wrong to treat suspended consumers separately, i.e. by marking and early resetting from one suspended consumer completely before the other. Instead, the correct treatment of suspended consumers consists in: for each C mark the reachable environments and heap; only *after* this operation is finished for each C mark and early reset the trail of C. This is because it is quite possible to have e.g. two suspended consumers which share some part of the trail (as in Fig. 3) and some trailed variable being reachable in the forward computation of one but not of the other.

4.2 Marking of substitution factors & from the completion stack

As mentioned, a substitution factor record contains the variables in the subgoal. These variables have to be accessed when a new answer is generated (i.e. at the return point of each clause) in order for the answer substitution to be inserted in the table. Without proper compiler support[5], it is quite easy for substitution factoring to become incompatible with the implementation of the garbage collector. Indeed, the compilation scheme for tabled predicates described in [14, 15] does not reflect the usefulness logic of tabled evaluations and the only alternative to changing it, is to impose strong restrictions on the order of marking. The following example illustrates the issue:

[5] As a general comment, usually garbage collection requires compiler support: cf. the missing initialization of local variables in the WAM and the story in this subsection.

Consider the execution of a query `?- test.` w.r.t. the tabled program given below. Here and in the sequel, we use the predicate `gc_heap/0` to indicate the point in the computation where garbage collection is triggered. Then marking as performed by a Prolog garbage collector would consider the heap value of variable `X` as not useful. In a tabled abstract machine, the binding of `X` to `[a]` is trailed as tabled predicates always create a choice point (cf. [14]). In such a situation, a Prolog garbage collector would invoke early reset of `X`. This is correct, as in the usefulness logic of Prolog, `X` is not used in the forward continuation of the computation. However, note that the usefulness logic of tabled evaluation is different: `X` also appears in the substitution factor record and its binding needs to be accessed at the return point of the corresponding tabled clause. Otherwise, a wrong answer is inserted in the table. Dealing with this issue by looking at the code in the forward continuation is complicated by the fact that the XSB abstract code for `t/1` (as shown above) was not designed to reflect the usefulness logic of tabling: the first argument of the new_answer instruction contains the arity of the procedure and the second a pointer to the subgoal frame in the table space; the substitution factor is accessed only indirectly (cf. [14]).

XSB program	XSB abstract code
`test :- t(_).`	tabletrysingle 1 ...
	allocate 2 2
`:- table t/1.`	getVn v2
`t(X) :-`	call 3 p/1
`p(X),`	call 3 gc_heap/0
`gc_heap.`	new_answer 1 r2
	deallocate
`p([a]).`	proceed

To overcome this particular problem compiler support is desirable but not strictly required: an alternative is to mark variables in the substitution factor records of all generators *before* marking anything from the active computation. In other words, marking in a CHAT implementation should start by considering as root set pointers to the heap from the completion stack — this is where CHAT keeps a pointer to the substitution factor record of generators (cf. [9] and Fig. 2). In this way, problems caused by this kind of premature early reset are avoided.

4.3 A chain bit in cells of the CHAT area

As Fig. 3 shows, cells in the CHAT area can contain references to the heap. The CHAT areas however, are dynamically allocated and so it is natural to cater for the chain bits in the CHAT sub-areas themselves. We implement a sequence of N words that all need a chain bit, as a number of groups of $(S+1)$ words, where the first S are some of the N words and the $(S+1)^{th}$ word, contains the S chain bits — we actually use a byte for each chain bit. We make sure that each group of $(S+1)$ words is aligned on a $(S+1)$-boundary. S is chosen as `sizeof(Cell *)` that is the size of the pointer type used for cells of the WAM stacks. A pointer p to a CHAT object, can now be translated to a pointer to its chain byte as follows: let $i = ((((\texttt{int})p)/S)\%(S+1))$, then $pointer\_chain\_byte = (\texttt{char}\ *)(p+S-i)+i$.

5 Performance evaluation

Putting our garbage collectors in perspective Prolog systems have usually opted for a sliding collector since traditionally the order of segments is considered important for cheap reclamation of heap on backtracking and for preserving the semantics of the @-family of compare built-ins. However, the ISO Prolog standard has removed the latter reason, and [3] and [6] argue in different ways against the former reason. So, after having implemented the sliding collector, we implemented a copying collector starting from the marking phase that is common for both collectors and following [3]. The fact that our copying collector is not segment-preserving, makes the interpretation of the results of tests that include backtracking not always clear cut. Few other Prolog systems have more than one garbage collector: [3] reports that Reform Prolog also had a copying and a sliding collector. BinProlog gave up its sliding collector in favor of copying. Touati and Hama report on a partially copying collector (for the most recent segment only) that is combined with a sliding collector: see [18]. In addition, XSB is currently the only system that has tabling implemented at the engine level. So, in tabled programs we can at most compare our collectors with each other and only for plain Prolog execution with collectors from other systems. An extensive comparison of a sliding and a copying collector in the context of a functional programming language can be found e.g. in [16] and its results carry over to Prolog; see also [3]. Two points are worth noting: accurate marking is the difficult part of both collectors and the copying phase is much easier to implement and maintain than the sliding phase. On the other hand, a copying collector may be more difficult to debug due to motion sickness: heap objects can change order.

Performance in programs without tabling We felt that there was no good reason to do lots of testing for programs without tabling as the relative merits of copying and sliding have been discussed in the literature at length. We just present one measurement that places our garbage collectors in context. The test program used (shown as part of Table 1) builds two lists that are interleaved on the heap [6]. Note that length of the second list is 1/10 the length of the first one. The two queries represent the following two situations: either most of the data survives garbage collection, or only a small fraction does. The Prolog systems were started with enough initial heap so that no collection occurred, except the explicitly invoked one. We measured the time (in milliseconds on an Intel 686, 266MHz running Linux) for performing the one garbage collection during the queries `?- q1.` and `?- q2.` In XSB this was done with the two collectors; in ProLog_by_BIM 4.1 (now named MasterProlog) and SICStus 3.7 using the sliding collector; in BinProlog 6.84 with its segment-preserving copying collector. Table 1 provides some evidence that the collectors of XSB are similar in performance to those of commercially available systems. It also shows that (in the absence of backtracking) copying is a reasonable alternative to sliding.

Performance in programs with tabling To get an idea of how our garbage collectors perform on programs with tabling, we took the programs from the benchmarks in [9] and gave them just enough heap and local stack so that expansion of these areas was not necessary: in all cases this meant that the garbage collection

[6] The particular form of `mklists/3` was chosen so as not to disadvantage BinProlog because of its binarization.

	gc-q1	gc-q2
XSB slide	1337	316
BIM slide	1270	430
SICStus slide	1426	434
XSB copy	890	126
BinProlog copy	974	214

```
mklists(0,L1,L2) :- !, L1 = [], L2 = [].
mklists(N,[1,2,3,4,5,6,7,8,9,0|R1],[1|R2]) :-
      M is N - 1, mklists(M,R1,R2).
q1 :- mklists(100000,L1,L2),gc_heap,use(L1,L2).
q2 :- mklists(100000,L1,L2),gc_heap,use(_,L2).
use(_,_).
```

Table 1. Performance of sliding and copying GC on an artificial Prolog program.

was called at least once. In Table 2, we indicate, for each test the `-m` option for XSB (`-m13` allocates an area of 13000 cells of heap and local stack), the number of times garbage collection was called and the number of garbage cells reclaimed (both using copying and sliding), the time spent in garbage collection, and this time as a percentage of the total time for the benchmark. The runtime without garbage collection is given in the last row of the table. All times are again in milliseconds but this time on a Ultra Sparc 2 (168 MHz) under Solaris 5.6. We also note that in XSB a cell is represented using one machine word.

	cs_o	cs_r	disj_o	gabriel	kalah_o	peep	pg	read_o
-m	11	12	11	15	17	110	39	187
copying GC #	183	107	10	77	17	11	8	4
cells collected	43951	33100	4113	18810	9680	13519	8802	14462
GC time	90	77	0	77	21	800	179	439
% GC time	29	16	0	34	15	67	64	44
sliding GC #	86	57	3	40	5	8	5	2
cells collected	12143	11644	1050	16332	5265	12288	8138	13584
GC time	66	62	0	150	22	1319	129	410
% GC time	23	13	0	50	15	77	56	42
runtime (no GC)	219	400	130	151	121	400	100	560

Table 2. Performance of sliding and copying GC on a set of benchmarks with tabling.

In all cases the sliding collector gets invoked less frequently than the non segment-preserving copying collector and collects less garbage. In some cases the sliding collector can spend less time than the copying collector. The reason for this last behavior could be that the effect of the loss of reclamation on backtracking can be much worse when tabling is used and several consumers have frozen the heap than when using plain Prolog code. However, this effect is not uniformly visible in the tested programs. Also, in a backtracking system, a copying collector can be called arbitrarily more often than a sliding collector. On the other hand, note that because most data remains useful (this can be deduced from the low figures of collected garbage) the copying collector is disadvantaged in this benchmark set. A generational schema can in all cases improve the figures.

6 Measuring fragmentation & effectiveness of early reset

Prolog implementors have paid little attention to the concept of internal fragmentation: as far as we know, measurements of internal fragmentation have never been published before for any Prolog system. Still this notion is quite important, as it gives a measure on how effectively the total occupied space is used by the memory allocator, and in the memory management community it is a recurring topic (see e.g. [10]). It is also surprising that although early reset is generally implemented in Prolog garbage collectors, its effectiveness has not been reported in the literature with [17] being a notable exception. We will combine both measurements. Once one has a collector, these figures can in principle be obtained quite easily. It is important to realise that the results tell something about the memory allocator, not about the collector !

According to [10], internal fragmentation can be expressed in several possible ways. To us, the most informative seems the one that compares the maximal space required by the allocator (i.e. the minimal heap size required if no garbage collection were present) with the minimal amount of dead memory during the running of the program. In contrast to [10], we prefer to express fragmentation as the ratio of these quantities: it represents the percentage of wasted memory. E.g. a fragmentation of 75% means that without garbage collection the allocator uses four times more heap space than minimally needed to run the program.

To this effect, we have conducted two experiments: one using a set of typical Prolog benchmarks and another using the same set of tabled programs as before (as well as tboyer: a tabled version of the boyer program). To measure fragmentation reasonably accurately, we have forced the marking phase to be invoked every 100 predicate calls. At each such moment, the number of marked (useful) cells is recorded. Garbage is not collected at these moments. After the run, the two quantities above are computed and their ratio is reported in Tables 3 and 4 with and without performing early reset. Additionally, the tables contain the number of times there was an opportunity for early resetting a cell, the number of predicate calls (in K) and the maximum heap usage (in K cells). Note that one early reset operation can result in more than one cell becoming garbage.

fragmentation	boyer	browse	chat_parser	reduce	simple_analyser	zebra
with early reset	61.2	46.4	11.7	91.17	49.9	41.9
without early reset	61.2	46.3	6.4	91.15	47.9	16.5
# early resets	6	9	64	7	60	38
predicate calls (K)	865	599	74	30	16	14
maximum heap (K)	144	11	1	20	5	0.2

Table 3. Fragmentation in a set of Prolog programs with and without early reset

In most cases, the figures show a surprisingly high fragmentation: remember that the fragmentation gives an upper limit for the amount of *avoidable* waste given a perfect allocator. The fragmentation would be even higher if XSB would trim its environments and have a more sophisticated determinism detection.

fragmentation	cs_o	cs_r	disj_o	gabriel	kalah_o	peep	pg	read_o	tboyer
with early reset	52.3	51.7	63.4	62.9	83.6	67.3	73.8	53.9	0.44
without early reset	41.4	41.1	57.4	58.3	77.7	65.9	69.8	53.4	0.15
# of early resets	21.5	19.2	12.9	18.9	31.8	40.0	124.6	49.1	123
predicate calls (K)	72	138	40	47	45	134	34	169	6.6
maximum heap	1.1	1.2	0.8	2.1	4.8	28	12	43	67

Table 4. Fragmentation in a set of programs with tabling with and without early reset

The fragmentation with early reset is higher than without, because when performing early reset fewer cells are marked as useful. The effect of early reset can of course be very much dependent on the characteristics of a program, but given the range of programs here, it seems safe to conclude that one should not expect more than 10% gain in memory efficiency for most realistic programs. We also note that the experience reported in [17] is similar. On the other hand, the cost of early reset is extremely small: the test whether a trail cell points to a marked heap cell happens anyway, and the extra cost consists in resetting the (unmarked) heap cell and redirecting the trail entry. So it appears that early reset is worth its while both in Prolog as well as in tabled execution.

7 Concluding remarks

In this paper we bridged the usual gap between the theoretical understanding of a memory management principle and its actual implementation. We addressed the practical aspects of heap garbage collection in Prolog systems with or without tabling and reported our experience and the main lessons learned from implementing two garbage collectors in XSB. Furthermore, we addressed methodological issues of implementing and benchmarking garbage collectors: this is only rarely reported in the literature. We hold that by making implementation choices concrete and documented — even those that are considered folklore — this paper can be of significant value to implementors that consider garbage collection in a system with tabling or in a system that is similar in certain respects.

It is the case that relatively little has been published on garbage collection in logic programming systems. The WAM — on which many logic programming systems are based — has the reputation of being very memory efficient; still, it provides no support (in the instruction set for instance) for doing precise garbage collection. Also, the WAM has a fixed allocation schema for data structures (allocation on the top of the heap), about which there is relatively little empirical knowledge: the figures we presented show a high fragmentation and thus suggest that the WAM is not particularly good at using the heap very efficiently. Finally, most often people have been interested almost solely in the efficiency of *garbage collection-less* execution. Consequently, implementors have not been inclined to give up some efficiency for better memory management and some Prolog systems have even lived for quite a while without garbage collection at all. For sure research in logic programming implementation has not focussed on considering alternative memory management schemas, neither on *accurate* identification of useful data. This contrasts sharply with the attention that the functional programming community has given to memory management. Similarly to the story of fragmentation about which there seem no figures available in literature, no one has hard data on the effectiveness of early reset: our admittedly small set of benchmarks indicates how effective one can expect it to be in realistic programs. It is clear that a continuous follow-up on such issues is needed as new allocation schemas and extensions of the WAM emerge: such new allocation schemas will be the subject of our future research. More directly practical, we will also investigate incremental and generational variants of the current collectors in the XSB system.

Acknowledgements

We are grateful to members of the XSB implementation team for helping us understand code they designed or wrote ages ago: in particular, we thank David S. Warren, Terrance Swift, and Prasad Rao. The second author was supported by a K.U. Leuven fellowship.

References

1. K. Appleby, M. Carlsson, S. Haridi, and D. Sahlin. Garbage collection for Prolog based on WAM. *Communications of the ACM*, 31(6):719–741, June 1988.
2. Y. Bekkers, O. Ridoux, and L. Ungaro. Dynamic memory management for sequential logic programming languages. In *IWMM'92: International Workshop on Memory Management*, number 637 in LNCS, pages 82–102, Sept. 1992. Springer.
3. J. Bevemyr and T. Lindgren. A simple and efficient copying garbage collector for prolog. In M. Hermenegildo and J. Penjam, editors, *Proceedings of the 6th PLILP*, number 844 in LNCS, pages 88–101, Madrid, Spain, Sept. 1994. Springer-Verlag.
4. M. Carlsson. *Design and Implementation of an Or-Parallel Prolog Engine*. PhD thesis, The Royal Institute of Technology (KTH), Stokholm, Sweden, Mar. 1990.
5. C. J. Cheney. A nonrecursive list compacting algorithm. *Communications of the ACM*, 13(11):677–678, Nov. 1970.
6. B. Demoen, G. Engels, and P. Tarau. Segment preserving copying garbage collection for WAM based Prolog. In *Proceedings of the 1996 ACM Symposium on Applied Computing*, pages 380–386, Philadelphia, Feb. 1996. ACM Press.
7. B. Demoen and K. Sagonas. CAT: the Copying Approach to Tabling. In C. Palamidessi, H. Glaser, and K. Meinke, editors, *Principles of Declarative Programming*, number 1490 in LNCS, pages 21–35, Pisa, Italy, Sept. 1998. Springer.
8. B. Demoen and K. Sagonas. Memory management for Prolog with tabling. In *Proceedings of ISMM'98: ACM SIGPLAN International Symposium on Memory Management*, pages 97–106, Vancouver, B.C., Canada, Oct. 1998. ACM Press.
9. B. Demoen and K. Sagonas. CHAT: the Copy-Hybrid Approach to Tabling. In G. Gupta, editor, *Practical Aspects of Declarative Languages (PADL)*, number 1551 in LNCS, pages 106–121, San Antonio, Texas, Jan. 1999. Springer.
10. M. S. Johnstone and P. R. Wilson. The memory fragmentation problem: Solved? In *Proceedings of ISMM'98: ACM SIGPLAN International Symposium on Memory Management*, pages 26–36, Vancouver, B.C., Canada, Oct. 1998. ACM Press.
11. F. L. Morris. A time- and space-efficient garbage compaction algorithm. *Communications of the ACM*, 21(8):662–665, Aug. 1978.
12. E. Pittomvils, M. Bruynooghe, and Y. D. Willems. Towards a real time garbage collector for Prolog. In *Proceedings of Symp. on Logic Progr.*, pages 185–198, 1985.
13. I. V. Ramakrishnan, P. Rao, K. Sagonas, T. Swift, and D. S. Warren. Efficient access mechanisms for tabled logic programs. *J. of Logic Progr.*, 38(1):31–54, 1999.
14. K. Sagonas and T. Swift. An abstract machine for tabled execution of fixed-order stratified logic programs. *ACM Trans. Prog. Lang. Syst.*, 20(3):586–634, May 1998.
15. K. Sagonas, T. Swift, and D. S. Warren. An abstract machine for computing the well-founded semantics. In *Proceedings of JICSLP'96*, pages 274–288, Sept. 1996.
16. P.M. Sansom. Combining copying and compacting garbage collection or Dual-mode garbage collection. In *Functional Programming*, Glasgow, Aug. 1991. Springer.
17. J. Schimpf. Garbage collection for Prolog based on twin cells. In *Proceedings of the 1990 NACLP Implementation Workshop*, pages 16–25, Austin, Texas, 1990.
18. H. Touati and T. Hama. A light-weight Prolog garbage collector. In *Proceedings of FGCS'88*, pages 922–930, Tokyo, Japan, Nov./Dec. 1988. OHMSHA and Springer.
19. D. H. D. Warren. An abstract Prolog instruction set. Technical Report 309, SRI International, Menlo Park, U.S.A., Oct. 1983.