

Heap Garbage Collection in XSB: Practice and Experience

Bart Demoen
Kostantinos Sagonas

Report CW 272, September 1998



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

Heap Garbage Collection in XSB: Practice and Experience

Bart Demoen
Kostantinos Sagonas

Report CW 272, September 1998

Department of Computer Science, K.U.Leuven

Abstract

Starting from a theoretical understanding of the issues involved in the implementation of a heap garbage collector in a logic programming system with built-in tabling, and from an actual collector that did not take tabling (i.e. suspended computations) into account we have build two working heap garbage collectors (one mark&slide, one mark©) for XSB on top of a CHAT implementation model for the suspension/resumption of consumers. We discuss implementation issues and decisions that are general to heap garbage collections for the WAM and issues that are specific to an implementation with tabling: as such, this paper documents our 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 which were made possible through having implemented the garbage collectors.

Heap Garbage Collection in XSB: Practice and Experience ^{*}

Bart Demoen Konstantinos Sagonas

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

Abstract

Starting from a theoretical understanding of the issues involved in the implementation of a heap garbage collector in a logic programming system with built-in tabling, and from an actual collector that did not take tabling (i.e. suspended computations) into account we have build two working heap garbage collectors (one mark&slide, one mark©) for XSB on top of a CHAT implementation model for the suspension/resumption of consumers. We discuss implementation issues and decisions that are general to heap garbage collections for the WAM and issues that are specific to an implementation with tabling: as such, this paper documents our 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 which were made possible through having implemented the garbage collectors.

1 Introduction

In September 1996, we began the development of a heap garbage collector for XSB. A mark&slide collector was written, closely following e.g. [1] and previous experience: 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 and thus less accurate — i.e. leave out early reset and just consider everything pointer reachable as useful. Even though this would have been completely acceptable from an engineering perspective (see e.g. [21]), this option 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 of what is needed for accurate memory management in tabled abstract machines, whose internals we briefly review in Section 2, we resumed work on the garbage collector in August 1998 trying to finish the sliding collector, while at the same time implementing a copying collector as well:

^{*}This paper consists of \approx 17 pages. The appendix contains additional information but is not part of the submission.

the copying collector uses the same marking phase as the sliding one (see [3]) 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.

This paper is on one hand a trace of issues that came up, real problems that occurred, their solutions, decisions we took and why. These are the contents of Sections 4 and 5. Some — perhaps all — 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 this report on the practical aspects of building a garbage collector is of interest 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 6. Finally, Section 7 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.

2 Memory Organization in WAM-based Tabled Abstract Machines

Preliminaries: The implementation of tabling on top of the WAM [20] 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 possibilities 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 of tabling, and refer the interested reader to the above references for differences between these abstract machines¹. Note that recently a tabling mechanism named *linear tabling* emerged which does not use suspension/resumption [17]. Heap management in a linear tabling system is exactly like in a Prolog system without tabling.

Independently of the implementation model that is chosen for the suspension/resumption mechanism, tabling calls for sophisticated memory management. Indeed, tabling systems have inherently more complex memory models and in general their space requirements are 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 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 the purpose of this paper is to report our implementation exper-

¹All relevant papers are accessible at <http://www.csd.uu.se/~kostis/Papers/>.

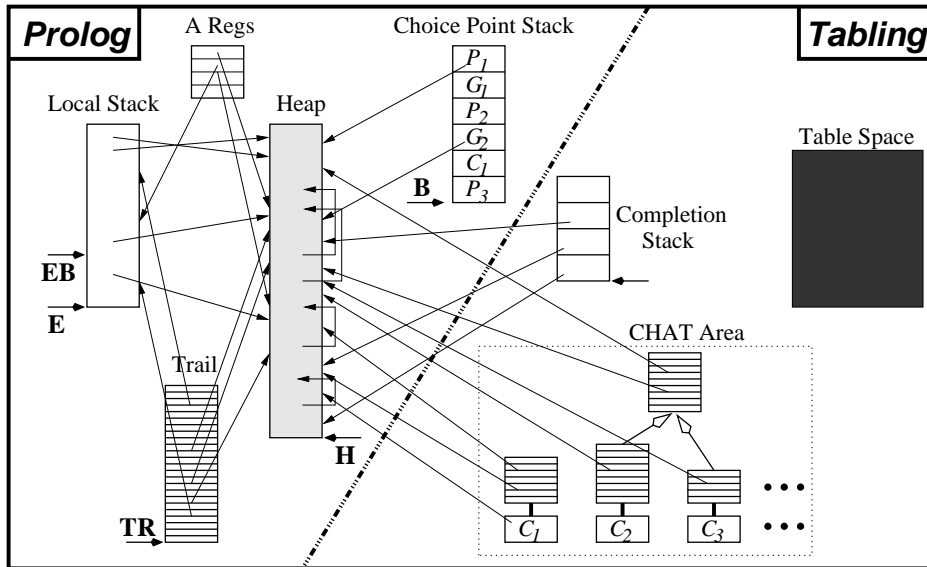


Figure 1: Memory snapshot of a CHAT-based abstract machine when garbage collection is triggered.

rience by discussing garbage collection issues that are nowhere presented in the literature and to experimentally evaluate alternative garbage collection schemes in the context of tabled execution of logic programs, this section only contains information from [8, 9] that is necessary to make the current document reasonably self-contained. It should be relatively straightforward to deduce how the discussed issues translate in a CAT or in an SLG-WAM-based implementation; whenever it is not the case, their translation is explicitly mentioned. **are we doing that anywhere?** We concentrate on the following typical scenario: BMD

The memory management policy has decided that the heap better be garbage collected. The issues to consider are: 1) find the set of reachable 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.

Figure 1 shows a rough picture² 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 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

²In figures, the relative size of memory areas is not realistic. By convention, all stacks grow downwards.

collection techniques are standard and well-described in the literature; see e.g. [1, 3, 6]. We will concentrate on the memory areas of the ‘Tabling’ part and the choice point stack because it differs from that of the WAM.

As the memory snapshot shows, the evaluation involved consumers, some of which, e.g. C_2, C_3, \dots are currently suspended (appear only in the CHAT area which is explained 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 (i.e. there is a one-to-one correspondence between generators and tables) 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 will be 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) reinstallation of the consumer choice point C immediately below G and 2) setting $C[H]$ to $G[H]$ so that C does not reclaim any heap that G protects (e.g. in Figure 2, G_2 and C_1 protect the same heap: from the top till the dotted line). Finally³, 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 Figure 2).

As mentioned in the beginning of this section, the implementation of tabling becomes more complicated when there exist mutual dependencies between subgoals. The execution environments of the associated generators, G_1, \dots, 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 area known as *Completion Stack*. In Figure 2, the situation depicted is as follows:

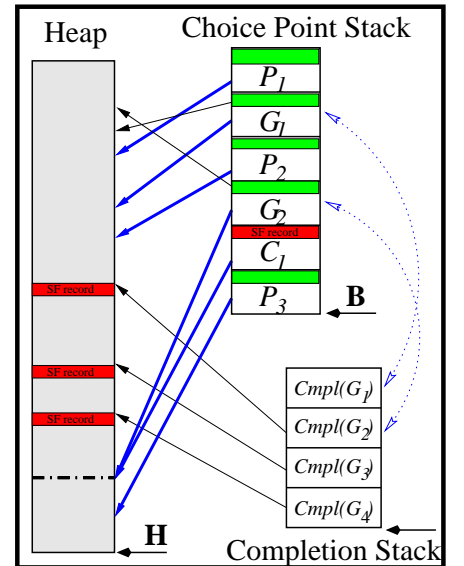


Figure 2: Detail of choice point & completion stack w.r.t. the heap.

³In programs containing tabled negation or aggregation, a fourth type of choice point called a *completion suspension frame* gets into the picture; its treatment by the garbage collector is similar to consumer choice points and so it is not described specially. Henceforth, we will use the term *suspended computation* to refer to both types of suspension.

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 the dotted lines in Figure 2.

Substitution factors As noted in [13], 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 persists backtracking: in CHAT a cell of each completion stack frame $Cmpl(G_i)$ points to SF ; for consumers the substitution factor can be part of the consumer choice point as described in [13]; see Figure 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 Figure 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 possible 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]). Note that the $C[H]$ field of suspended consumer choice points that reside in CHAT areas can be safely ignored by garbage collection: as mentioned this field gets a new value upon resumption of C and reinstallation of its choice point. 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 5.3, 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.

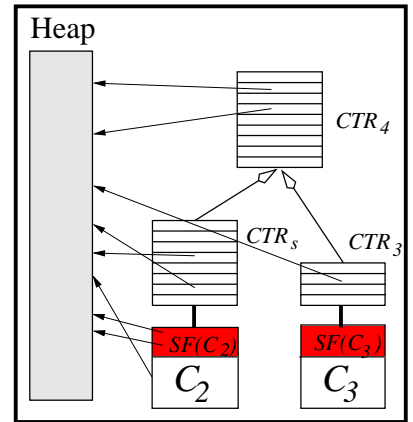


Figure 3: Detail of CHAT area.

3 The need for Garbage Collection in a Prolog system with Tabling

It is generally accepted that high-level languages must rely on automatic memory management, which includes garbage collection. Prolog is no exception. The impact of tabling on memory consumption is not a priori clear and one might thus wonder to which extent a Prolog system with tabling will need garbage collection or really benefit from it. Since the scope of this paper is the heap, we will here discuss shortly the effect of tabling on heap consumption: other memory areas (the tables, the CHAT areas, the completion stack ...) will not be discussed.

There are essentially three ways in which tabling affects heap consumption:

1. because of suspension of consumers, parts of the heap are frozen and cannot be reclaimed on backtracking; this increases heap consumption until completion has taken place
2. tabling can remove computations that require considerable heap; this can lower the heap consumption arbitrarily as we will show in an example later
3. tabling can diminish sharing and in this way also increase the heap consumption arbitrarily; also this will be exemplified

The following definition of a predicate $p/3$ will serve for both examples:

```
p(N,Term,Out) :-  
  ( N = 0 ->  
    Out = Term  
  ; M is N - 1,  
    trans(Term,NewTerm),  
    p(M,NewTerm,Out)  
  ).
```

The example that shows point 2 above, is gotten by defining

```
trans(X,X) :- do_some_memoryintensive_computation.
```

The query $?-p(N, 1, Out)$. has $O(N)$ memory consumption if `trans/2` is not tabled and constant memory consumption if `trans/2` is tabled.

For showing point 3 above, we define:

```
trans(X,X).
```

If `trans/2` is tabled, the space consumption of the query $?-p(N, [a], Out)$. is $O(N)$ while if `trans/2` is not tabled, space consumption is independent of N .

The examples show that tabling can both increase and decrease heap consumption. It follows that also Prolog with tabling needs garbage collection.

4 General issues in implementing a Heap Garbage Collector

This section discusses issues that are relevant to the implementation of a heap garbage collector in any WAM-based implementation: some or all of these are folklore, but have not been published before as far as we know ⁴.

⁴We would be glad to give credit to people who have originally noted similar things.

4.1 Dealing with uninitialized 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 Prolog; 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, most Prolog systems (including XSB) have no explicit ⁵ information about which permanent variables are alive at a particular moment in the computation ⁶. In such a system one faces a choice between the following options, given in increasing order of difficulty of implementation:

1. initialize environment variables to some atomic value (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 quickest to implement by extending the `allocate` whose extra cost is usually quite low. Note that also SICStus ensures that all permanent variables are initialized before the first call instruction.

4.2 The test for heap overflow

XSB, like many other logic programming systems, relies on software tests for overflow of all its stacks. In XSB, overflow was checked every time the H register was increased. 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 Prolog procedure. 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.

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. The compiler should be further adapted to count more precisely how much the margin should be (as in MasterProlog). Another change to be made to the implementation of built-in predicates is that they should not commit any changes until they are sure to have enough heap space and if not, call the collector. This is reasonably straightforward to implement but tedious.

If built-in predicates that consume heap are inlined, a variant of the `test_heap` instruction might also be needed in the body of clauses: it needs to have access to the length of the current environment. Also inlined built-in predicates that create a choice point need this information for garbage collection reasons. Such a built-in occurred in the treatment of tabled negation, as the `negation_suspend` built-in predicate lays down such a choice point (called a completion suspension frame in [14]) and subsequent garbage collection would not do appropriate marking without modification. We have simply disabled the inlining of this particular built-in, because it was not time critical.

A final word on the generation of `test_heap` that is specific to tabled execution. In the WAM, and also in a CAT implementation of tabling, it is enough to execute the `test_heap` instruction at

⁵The information is implicit in the WAM code.

⁶MasterProlog is a notable exception.

the entry point of each predicate. However, in an SLG-WAM or in a CHAT implementation, a test for heap overflow should be done on every fail operation as well: the reason is that failure in the SLG-WAM and CHAT may not reclaim heap space. Thus, the `test_heap` instruction should be executed at the beginning of each clause.

4.3 Bubbles in the stacks

A bubble is a region which contains something in a different format than the usual WAM representation of Prolog terms. Such a bubble must be clearly identifiable, otherwise garbage collection cannot deal with it. XSB version 1.8 and earlier used bubbles on the heap for various tasks: one of them being the implementation of a copy-once `findall/3`, another one in `assert/1`. Previous experience in the implementation of the BinProlog garbage collector [6] showed that dealing with such bubbles is a pain one wishes to avoid. Fortunately XSB has become bubble-less since version 1.9.

Bubbles existed elsewhere in XSB as well: notably some choice points contained non-tagged data, so that it was impossible to scan a choice point without taking into account its particular lay-out. If one wishes a more uniform treatment of choice points, such untagged data must be avoided: the small overhead in execution is not worth the increase in code complexity in the garbage collector. See also section 5.4.

4.4 H pointers in choice points

First, note that the H pointers 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 heap pointer from the scheduling generator. A similar reasoning holds for SLG-WAM. So only the H pointers of the choice points in the active computation need to be considered for marking and possibly chaining. 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 pointer of the choice point can be treated. At that point, the **H** register can point to a cell which was marked already and no further action is required. Otherwise, **H** points to a cell that was not marked and then two ways of dealing with this situation are common practice:

1. mark the cell and fill it with an atomic object;
2. scan the heap in the direction of the top for the first marked cell and make the **H** register point to it

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 following observation:

```
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 become marked for some other reason.
```

Since according to [8] the suspended computations are marked after the active computation, and since the substitution variables are *never* reachable from the active computation (only from the frames in the completion stack), in a CHAT garbage collector this action needs to be postponed until after the substitution variables are marked. In our implementation, 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.

4.5 Trail compaction during garbage collection

We have chosen not to perform trail compaction during garbage collection. Trail compaction is possible whenever early reset is performed (cf. Section 5.1). When trail compaction is omitted, one must make sure that trail cells that could have been removed point to something that is “reasonable”. One can reserve a particular cell on the heap (in the beginning) for this purpose, i.e. all trail entries that are subject to early reset can be made to point to this one heap cell⁷. Since reserving such a cell would involve changes elsewhere in XSB, we have chosen to mark the cell that was reset and to 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 7 as in our measurements we counted these cells as garbage.

4.6 Chain bits for dynamically allocated root pointers

When implementing a mark&slide collector in a system whose cell representation does not allow for the mark and chain bits to be stored in the same machine word as the cell (this holds for XSB) one is faced with the problem where to store the chain bit for root pointers. As long as these root pointers are to be found in easily recognizable and contiguous areas — like choice point stack, trail stack, local stack — one can allocate a parallel chain/mark bit array and the translation from a root pointer to its corresponding bit is straightforward. For small sets of root pointers, it is also a good solution to just copy them to the top of the heap (such is the treatment of e.g. the argument registers), 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 in the CHAT area, as the trail chunks are allocated dynamically using `malloc()`. In Section 5.3 we discuss in more detail how we dealt with this issue.

4.7 The copying collector

The copying collector was implemented following the ideas of [3]. We deviate slightly from the usual two-space schema of Cheney (see [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. There are two reasons for this:

1. we believe this scheme uses memory resources more economically: usually, the *to*-space can be quite a bit smaller than the *from*-space;
2. 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].

⁷This trick seems folklore and was once described in `comp.lang.prolog` by R.A. O’Keefe.

5 Tabling-specific issues of Heap Garbage Collection

As mentioned, in CHAT and in the SLG-WAM the active computation and the suspended ones have their data intertwined on the shared heap. In such an implementation scheme for the heap, 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 case of 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, starting from a consumer in the CHAT area, the marking does not need to consider the part of the heap that is older than the generator G up to which the consumer has his execution environment CHAT-protected, or even the choice points between \mathbf{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. In short, garbage collection can be implemented efficiently in a CHAT-based tabled abstract machine.

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.

5.1 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 for its alternative branches, i.e. on backtracking) can be set to unbound during garbage collection and the trail entry itself can be discarded as well. 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 — as the suspended computations have to be taken into account as well — but still possible to do for both the active and the suspended computations. [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 Figure 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 Figure 3) and some trailed variable being unreachable in the forward computation of one but not of the other. Appendix B contains an example program which exhibits this situation for the a trailed variable in the local stack; similar examples can be constructed for trailed heap variables.

5.2 Marking of substitution factors & marking 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 support⁸, 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:

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. At this point, 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

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

accessed at the return point of the corresponding tabled clause. Otherwise, a wrong answer is inserted in the table and in fact the problem will not be visible until a subsequent occurrence of `t(X)` tries to resolve against this answer in the table. Dealing with this issue by looking at the code in the forward continuation gets unnecessarily 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]).

Actually, to overcome this particular problem compiler support is desirable but not strictly required: an alternative is to force a marking of 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 Figure 2). In this way, problems caused by this kind of premature early reset are bypassed. We also note in passing that similar problems exist concerning the treatment of the delay list which is conceptually also a part of the substitution factor record but as an optimization is not implemented as such (cf. [15]).

5.3 A chain bit in cells of the CHAT area

As Figure 3 shows, objects in the CHAT area — in particular the trail increments and the substitution factor variables in the consumer choice point — can contain references to the heap. The CHAT sub-areas however, are dynamically and separately allocated and so one needs to cater for the chain bits in the CHAT sub-areas themselves. Our implementation is as follows: a sequence of N words that all need a chain bit, is implemented 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 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. This means that we reserve a byte

⁸As a general comment, it is not unusual that garbage collection requires compiler support: cf. the missing support in the WAM for initialization of local variables and the story in this subsection.

for each chain bit. A pointer p to a CHAT object, can now be translated to a pointer to its chain byte as follows: let $i = (((\text{int})p)/S)\%(S + 1)$, then $\text{pointer_chain_byte} = (\text{char } *) (p + S - i) + i$.

5.4 Diversity of choice points

One major implementation complication we had to deal with was related to the fact that in XSB not all choice points have the same layout. This is partly due to the different functions that choice points can have in a tabled implementation (cf. Section 2 and [13, 14]), but also due to the fact that different XSB developers made non-uniform decisions. The problem is really that some of these choice points contain bubbles.

The first solution that comes to mind, i.e. to have the garbage collector check for the type of choice point (it can be deduced from its alternative field) is both error-prone for maintenance and non-practical as there are too many opcodes possible. As a permanent solution, a uniform representation for choice points was introduced in XSB which also avoids bubbles.

6 Performance Evaluation

Putting our garbage collectors in perspective Apart from XSB, few Prolog systems had 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 the copying one. Touati and Hama report on a partially copying collector (for the most recent segment only) that is combined with a sliding collector: see [19] for more details. 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 for example 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.

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 the tests that include backtracking not always clear cut.

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 heap⁹. 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

⁹The particular form of `makelists/3` was chosen so as not to disadvantage BinProlog because of its binarization.

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 ¹⁰ 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.

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

```

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

```

Table 1: Performance of sliding and copying garbage collectors on an artificial Prolog program.

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 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
<code>-m</code>	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 garbage collection 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

¹⁰now named MasterProlog

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 how much garbage is collected) the copying collector is disadvantaged in this set of benchmarks. A generational schema can in all cases improve the figures.

7 Measuring Fragmentation and 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 never been reported in the literature. 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 (without tabling) 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 (i.e. useful) cells and the size of the heap (difference between current heap top and heap bottom) are 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	144	11	1	20	5	0.2

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

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

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 figures would show an even higher fragmentation, if XSB would trim its environments and have a more sophisticated determinism detection.

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. 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.

The fragmentation for **tboyer** is extremely small when compared to the higher fragmentation for **boyer**: inspection of **tboyer** teaches that it benefits a lot from the effect described in the second point of 3.

8 Concluding Remarks

There is usually a gap between the theoretical understanding of a memory management principle and its actual implementation. This paper bridges such a gap, because it makes implementation choices concrete and as such it can be of value to implementors that consider garbage collection in a system with tabling or in a system that is similar in certain respects. Indeed, we strongly believe that the underlying concepts are also applicable to systems that support other forms of execution based on a suspension/resumption mechanism: not only tabled ones.

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 in Section 7 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 a better memory management and some Prolog implementations 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 indicate 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 several people from the XSB implementation team for helping us understand code they designed or wrote ages ago: in particular, we want to thank David S. Warren, Terrance Swift and Prasad Rao. The second author was supported by a K.U. Leuven junior scientist 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 Y. Bekkers and J. Cohen, editors, *Proceedings of IWMM'92: International Workshop on Memory Management*, number 637 in LNCS, pages 82–102, St. Malo, France, Sept. 1992. Springer-Verlag.
- [3] J. Beveymyr and T. Lindgren. A Simple and Efficient Copying Garbage Collector for Prolog. In M. Hermenegildo and J. Penjam, editors, *Proceedings of the Sixth International Symposium on Programming Language Implementation and Logic Programming*, number 844 in LNCS, pages 88–101, Madrid, Spain, Sept. 1994. Springer-Verlag.
- [4] M. Carlsson. *Design and Implementation of an Or-Parallel Prolog Engine*. PhD thesis, Department of Telecommunication and Computer Systems, The Royal Institute of Technology (KTH), Stockholm, 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, 10th International Symposium, PLILP'98, Held Jointly with the 6th International Conference, ALP'98*, 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: First International Workshop*, 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 the 1985 Symposium on Logic Programming*, pages 185–198, Boston, Massachusetts, July 1985. IEEE Computer Society Press.
- [13] I. V. Ramakrishnan, P. Rao, K. Sagonas, T. Swift, and D. S. Warren. Efficient Access Mechanisms for Tabled Logic Programs. *Journal of Logic Programming*, 38(1):31–54, Jan. 1999.
- [14] K. Sagonas and T. Swift. An Abstract Machine for Tabled Execution of Fixed-Order Stratified Logic Programs. *ACM Transactions on Programming Languages and Systems*, 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 M. Maher, editor, *Proceedings of the Joint International Conference and Symposium on Logic Programming*, pages 274–288, Bonn, Germany, Sept. 1996. The MIT Press.
- [16] P. M. Sansom. Combining copying and compacting garbage collection or Dual-mode garbage collection. In R. Heldal, C. K. Holst, and P. Wadler, editors, *Functional Programming, Workshops in Computing*, Glasgow, Aug. 1991. Springer-Verlag.
- [17] Shen, Y.D., Yuan, L., You, J.H. and Zhou, N.F. Linear Tabulated Resolution Based on Prolog Control strategy. Submitted for publication
- [18] P. Tarau and U. Neumerkel. A Novel Term Compression Scheme and Data Representation in the BinWAM. In M. Hermenegildo and J. Penjam, editors, *Proceedings of the Sixth International Symposium on Programming Language Implementation and Logic Programming*, number 844 in LNCS, pages 73–87, Madrid, Spain, Sept. 1994. Springer-Verlag.
- [19] H. Touati and T. Hama. A light-weight Prolog garbage collector. In ICOT, editor, *Proceedings of the International Conference on Fifth Generation Computer Systems (FGCS'88)*, pages 922–930, Tokyo, Japan, Nov./Dec. 1988. OHMSHA Ltd. Tokyo and Springer-Verlag.
- [20] D. H. D. Warren. An Abstract Prolog Instruction Set. Technical Report 309, SRI International, Menlo Park, U.S.A., Oct. 1983.
- [21] B. Zorn. The measured cost of Conservative Garbage Collection. *Software Practice and Experience*, 23(7):733–756, July 1993.

A Debugging and Testing a Garbage Collector

A.1 Debugging

A garbage collector does wonders in debugging a system ! Indeed, a side-effect of writing a garbage collector for a system that never had any is that one is about to discover most of the features of the system that a garbage collector does not like (e.g. places in the code where invariants are not respected), as well as many of its bugs. The garbage collector rarely misses them or leaves them unpunished !

On the other hand, even without having to deal with this problem, debugging a garbage collector is quite hard. A proof of correctness of the algorithms is of course handy, but the theory might not quite fit the system the garbage collector is built into, and the actual implementation might have overlooked some issues. Debugging becomes considerably easier if lots of testing code is added to the garbage collector: since garbage collection relies on some invariants — e.g. that all pointers in the trail point to heap or local stack cells; all structure pointers point into the heap — it is worth adding code that explicitly tests these invariants. Apart from testing such invariants (that actually should be guaranteed by the run time system as well), we also added tests that the collector satisfies:

- after the garbage collection, all the mark bits are zero
- after a copying garbage collection, the amount of cells copied in the *to*-space equals the amount of cells marked as reachable
- during chaining, the substitution factor was marked already

To ease debugging, we also wrote a series of *dump* functions, which dump the different memory areas in the form of a set of Prolog facts. For instance, the heap dump at the top level of XSB looks like:

```
heap('100d98', 0, not_m, funct, '$load_undef'/1).
heap('100d9c', 1, not_m, undef, _).
heap('100da0', 2, not_m, funct, 'conset'/2).
heap('100da4', 3, not_m, atom, '$abort_cutpoint').
heap('100da8', 4, not_m, int, 72).
.....
heap('100e04', 27, not_m, funct, 'atom'/1).
heap('100e08', 28, not_m, atom, 'print_all_stacks').
heap('100e0c', 29, not_m, cs-ref_heap, 27).
heap('100e10', 30, not_m, atom, []) .
heap('100e14', 31, not_m, atom, 'print_all_stacks').
```

The meaning of the arguments of the `heap/5` predicate in this case is:

1. the virtual address of the cell
2. index in heap
3. whether the cell is marked/chained
4. type of the cell
5. value of the cell

Given such a set of Prolog facts, it is not difficult — in particular for a sliding collector — to write a Prolog program that computes the heap dump after garbage collection, given the dump at the beginning of the garbage collection process ¹¹. By comparing the computed dump with the actual dump, some bugs are much easier to find. Also, it is quite easy to query a heap/local stack dump, for instance by:

```
?- heap(_,Index,_,cs-Ref,_), Ref \== ref_heap.
```

to check whether any structure pointer points outside the heap, and

```
?- heap(_,Index,_,cs-ref_heap,Struct), not(heap(_,Struct,_,funct,_)).
```

to check whether every structure pointer points to a functor.

The availability of tabling in XSB made it quite easy to formulate certain garbage collection related queries — *reachability is a transitive relation!* — without having to worry about termination. For example, reachability queries were guaranteed to terminate even if the heap was corrupted or contained cyclic terms. Naturally, during the run of such a query, garbage collection had to be deactivated.

A.2 Testing

Apart from debugging, the garbage collector was also tested in three ways:

1. running artificial test programs which were meant to test the garbage collector under specific conditions: e.g. when there were 2 consumers suspended, both with non-empty trail of the same cell and while there was garbage in the heap specific to the consumers as well as in the heap of the active computation.
2. running the XSB test suite (enhanced with benchmark programs from [9]) while performing a garbage collection every 100 clause calls; this was achieved by putting a call counter in the `test_heap` instruction (cf. Section 4.2). This leads to numerous garbage collections during most of the test programs and indeed, some problems showed up only after more than 50 garbage collections had taken place.
3. running the same suite while starting XSB with a small amount of heap only.

The third method tests at the same time the interaction of garbage collection with the heap/local stack expansion routines and indeed, at some point a bug was detected which showed only if the non-most recent choice point had a value in its H field that equals the value of the **H** register at the moment of expansion.

B Early reset and the order of marking suspended consumers

The program in Figure 4 shows why performing early reset for suspended consumers without first finishing the marking of environments and heap reachable from consumers is not safe. Predicate `create_cp/0` is a dummy predicate which creates a choice point: this makes sure that the binding of **X** to `xxx` is trailed and also that **X** stays in the environment. The execution of a query such as `?- test.` encounters two generators (denoted by the subscripts g_i in the program) and two consumers (denoted by the subscripts c_i). The subscripts reflect the order in which generators and

¹¹Note that such a Prolog program is in fact a declarative implementation of the garbage collector !

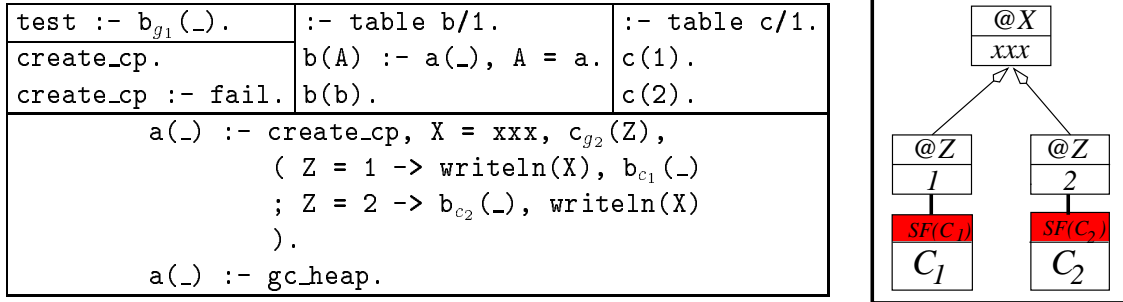


Figure 4: Program exhibiting trail sharing between suspended computations in CHAT.

consumers are encountered. Before garbage collection occurs, the state of a CHAT-based tabled abstract machine is as follows: Each of the two consumers has had a part of its state separately CHAT-protected till the younger of the two generators, $c_{g_2}(Z)$, and upon backtracking out of the choice point of this generator, they need to have their state CHAT-protected till their own generator $b_{g_2}(-)$. In particular, the part of the trail that lies between $g_1[\text{TR}]$ and $g_2[\text{TR}]$ is saved (as a value trail) *once* and is shared between the two consumers. The situation is pretty much that depicted in Figure 3 but in a more simplified form; it is shown in detail as part of Figure 4.

Now consider the garbage collection in the second clause of `a/1` and suppose that C_1 is followed for marking first. Variable `X` is unreachable for C_1 (it is not used in either its forward or backwards continuation) and its binding can be early reset as far as C_1 is concerned. However, this action is wrong as this variable (and its value) is also used in the forward continuation of C_2 whose suspended state shares this part of the trail. As explained in Section 5.1, the safe thing to do is to postpone early reset from suspended consumers until after marking of heap and local stack has taken place for all such consumers.

C Some more Performance figures on Prolog Garbage Collectors

This section represents the timings for a set of queries for programs which can be summarized as follows:

```

makeds(N,DS1,DS2) :-
    N = 0, !,
    DS1 = [],
    DS2 = [].
makeds(N,DS1,DS2) :-
    M is N - 1,
    longds(DS1,Rest1),
    shortds(DS2,Rest2),
    makeds(M,Rest1,Rest2).

q1 :- makeds(15000,DS1,DS2), gc_heap, use(DS1,DS2).
q2 :- makeds(15000,_,DS2), gc_heap, use(_,DS2).

use(_,_).

```

The different sorts of data structures that the benchmark builds on the heap, are described in

Table 5 by the facts for `longds/2` and `shortds/2`.

left list	<code>longds([[[[[[[[[[[R 0 9 8 7 6 5 4 3 2 1],R)</code>	<code>shortds(,R)</code>
right list	<code>longds([1,2,3,4,5,6,7,8,9,0 R],R)</code>	<code>shortds([1 R],R)</code>
left <code>f/2</code>	<code>longds(f(f(f(f(f(f(f(f(R,1),2),3),4),5),6),7),8),9),0),R)</code>	<code>shortds(f(R,1),R)</code>
right <code>f/2</code>	<code>longds(f(1,f(2,f(3,f(4,f(5,f(6,f(7,f(8,f(9,f(0,R))))))))),R)</code>	<code>shortds(f(1,R),R)</code>

Table 5: Construction of the datastructures

The number 15000 was chosen because higher numbers overflow the C-stack that BinProlog uses for its recursive marking phase. On the other hand, this is not a limitation for the other systems: marking is implemented in SICStus by in-place pointer reversal; in XSB and BIM by a self-managed stack.

	left list		right list		left <code>f/2</code>		right <code>f/2</code>	
	q1	q2	q1	q2	q1	q2	q1	q2
XSB copying	143	19	139	20	199	25	186	25
BinProlog	277	36	144	24	276	34	145	24
XSB sliding	219	48	229	51	322	72	267	61
BIM	420	106	222	99	282	115	482	146
SICStus	201	62	209	36	306	94	307	93

Table 6: Performance comparison of Prolog garbage collectors using different heap data sets.

The figures in Table 6 represent the timings for two queries with different garbage collection rates (high and low percentage of useful data) for 4 different data structures: 2 are lists and 2 are structures constructed with the functor `f/2`; 2 are recursive to the left, and 2 are recursive to the right. The difference in constructor is relevant, as the WAM represents lists differently from other binary constructors, while the BinWAM treats them exactly the same. On the other hand, the BinWAM implements term compression [18], a technique which reduces every right recursive binary constructor to the optimized list representation (term compression has a similar effect on constructors with higher arity as well); but has no effect on left recursive data structures. This means that a left list in the BinWAM takes one third more space than in the WAM, a right list the same, a left `f/2` structure takes one fourth more space and a right `f/2` structure takes one third less. Also, in the BinWAM left list and left `f/2` occupy exactly the same amount of space, as do right list and right `f/2`: this is clearly visible from the figures in the table.

Another reason for measuring both left and right recursive data structures, is that depending on choices in the marking algorithm, one can perform drastically worse than the other.

Some conclusions can be made from the performance figures in the table:

- all collectors perform much better on high percentage of garbage than on low; this is as it should be
- XSB and SICStus Prolog are rather insensitive to the left/right issue, while BIM and BinProlog are more sensitive: for both the marking schema is the reason, while for BinProlog, the term compression asymmetry is as well
- copying beats sliding; this is not immediately clear from the figures: one must normalize (w.r.t. the effect of term compression and its asymmetry) the figures of BinProlog to see this.

It would have been interesting to have the timings separated out for the marking phase (which is the same in both copying or sliding), so that a clearer insight could be gained on what marking schema is optimal.