Heap Memory Management in Prolog with Tabling: Principles and Practice

Bart Demoen
bmd@cs.kuleuven.ac.be

Konstantinos Sagonas
kostis@csd.uu.se

Abstract

We address memory management aspects of WAM-based logic programming systems that support tabled evaluation through the use of a suspension/resumption mechanism. We describe the memory organization and usefulness logic of such systems, and issues that have to be resolved for effective and efficient garbage collection. Special attention is given to early reset in the context of suspended computations and to what is involved in the implementation of a segment order preserving copying collector for a tabled Prolog system. We also report our experience from building two working heap garbage collectors (one mark&slide and one mark&copy) for the XSB system on top of a CHAT-based tabled abstract machine: we discuss general implementation choices for heap garbage collection in 'plain' WAM and issues that are specific to WAM with tabling. Finally, we compare the performance of our collectors with those of other Prolog systems and extensively analyze their characteristics. Thus, this article documents our own implementation and serves as guidance to anyone interested in proper memory management of tabled systems or systems that are similar in certain aspects.

1 Introduction

The incorporation of tabling (also known as memoization or tabulation in the context of functional languages [Bir80]) in non-deterministic programming languages such as Prolog leads to a more powerful and flexible paradigm: tabled logic programming. More specifically, through the use of tabling not only are repeated subcomputations avoided but also more programs terminate; thus the resulting execution model allows more specifications to be executable. As a result, practical systems incorporating tabling, such as the XSB system [SSW94], have been proven useful in a wide range of application areas such as parsing, logic-based databases [SSW94, YK00], program analysis [CDS98], and verification using techniques from model checking [CDD$^+$98]. At the same time, tabling introduces some extra complications in the standard implementation platform for Prolog, the WAM [War83]. Most of the complications can be attributed to the inherent asynchrony of answer generation and consumption, or in other words to the more flexible control that tabling requires: Control, i.e., the need to *suspend* and *resume* computations, is a main issue in a tabling implementation because some subgoals, called *generators*, generate answers that go into the tables, while other subgoals, called *consumers*, consume answers from the tables; as soon as a generator depends on a consumer, the generator and the consumer must be able to work in a coroutining fashion, something that is not readily possible in the WAM which reclaims space on backtracking. The need to suspend computations means that execution environments of these computations must be preserved. CHAT [DS00], the default abstract machine of XSB, preserves consumer states partly by *freezing* them, i.e., by not reclaiming space on backtracking as is done in WAM and by allocating new space outside the frozen areas. Execution environments of consumers are shared and end up interspersed on the stacks as in the SLG-WAM [SS98]. Copying-based implementations of tabling, such as CAT [DS99], are also possible but their space requirements are usually higher than sharing-based ones.

All the implementation models for the suspension/resumption mechanism result in tabling systems that have inherently more complex memory models than plain Prolog and in general their space requirements are bigger. In short, tabling requires more sophisticated memory management than Prolog. Despite this, issues involved in the memory management of tabled logic programming systems are currently unexplored in the literature, and this article fills this gap by providing a relatively complete account.

More specifically, we discuss the *usefulness logic* [BRU92] of Prolog with tabling (Section 5) and the issues that have to be addressed in order to add proper memory management in a tabling system (Sections 6–8), we document our garbage collection schemes, and extensively analyze their performance characteristics (Section 10). We describe memory management issues mainly for CHAT-like tabled abstract machines, but we believe that the issues carry over to other WAM-based implementations of tabling. On the other hand, this article does not address the memory management of the table space as this is an orthogonal issue.

This article was partly motivated by a desire to report our practice and experience in developing garbage collectors for logic programming systems in general, and for XSB in particular. In September 1996, we begun the development of a mark&slide collector for XSB, closely following previous experience. Pretty soon the collector worked as long as tabling was not used. When trying to extend it for tabled programs, we could not get a grasp on *early reset* (see e.g., [PBW85, BRU92]) 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; see e.g., [Zor93]. However it 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 CAT implementation schema [DS99] which lead indirectly to the understanding of the usefulness logic of logic programming systems with tabling described in Section 6 of this article. 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. We finally integrated our garbage collectors in the XSB system in February 1999 and since June 1999 they are part of the XSB release.

During the implementation we struggled a lot with technical details and misunderstandings of the invariants of the tabling run-time system. So, a large part of this article is a trace of issues that came up, real implementation problems that occurred, their solutions, decisions we took and why. These are the contents of Sections 5 and 8. 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 reporting 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 developing a system without thinking about its proper memory management. Sections 6 and 7 discuss in detail early reset and *variable shunting* in the context of tabled logic programming, respectively. Section 9 discusses issues that are involved in the implementation of a segment order preserving copying collector for a tabled system. Section 10 presents some performance measurements for our collectors on plain Prolog programs as well as on programs with tabling. Finally, Section 11 presents empirical data on the extent of 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 in Section 2, garbage collection in the WAM in Section 3, and present a short discussion on the need for garbage collection of the heap in a Prolog system with tabling in Section 4.

2 Memory organization in WAM-based tabled abstract machines

We assume familiarity with Prolog and the WAM [War83] (see also [AK91] for a general introduction) and to some extent with tabled execution of logic programs. Also knowledge of garbage collection techniques for Prolog will help but nevertheless we review them in Section 3.1; see also [BL94, DET96] for specific instances of heap garbage collection for Prolog and [BRU92, ACHS88] for a more general introduction. For a good overview of garbage collection techniques, see [JL96]. A good memory management glossary can be found at http://www.xanalys.com/software_tools/mm/glossary/.

Preliminaries As mentioned, the implementation of tabling on top of the WAM 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 implementing suspension/resumption in the WAM exist: either by totally sharing the execution environments by interspersing them in the WAM stacks (as in the SLG-WAM [SS98]), or by a total copying approach (as in CAT [DS99]), or by a hybrid approach (as in CHAT [DS00]). In this article, we mainly concentrate on a CHAT implementation of tabling, and refer the interested reader to the above references for differences between these abstract machines. Note that recently a different form of a tabling mechanism, named *linear tabling*, has emerged which does not require suspension/resumption [ZSYY00]. As a result, heap management in a linear tabling system is like in a plain Prolog system. Linear tabling is currently implemented in B-Prolog.

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 their space requirements are different from (usually bigger than) those of plain Prolog systems; see Section 4. As advocated in e.g., [BRU92], 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. This article presents the usefulness logic of Prolog systems with tabling, and describes how operations such as early reset can in principle be performed with equal accuracy in a CHAT or in an SLG-WAM-based abstract machine and implementation issues that this involves. Thus, this section only contains information from [DS00] that is necessary to make the current document reasonably self-contained.

The anatomy of CHAT Figure 1 shows a rough picture[1] 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. Because of this, a new register is introduced, named **EB**, which keeps the value of the top of the local stack upon choice point creation (this register behaves similarly to WAM's **HB** register). Besides this, the only

[1] In figures, the relative size of memory areas is not significant. By convention, all stacks grow downwards.

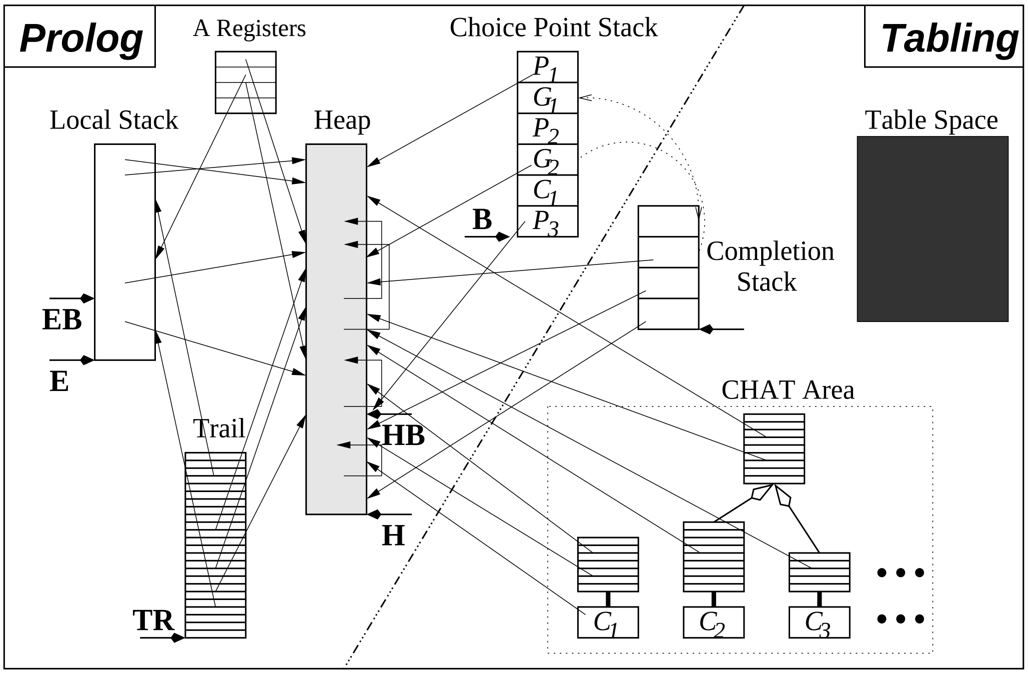


Figure 1: Memory snapshot of a CHAT-based abstract machine when garbage collection is triggered.

other 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. The architecture of the 'Prolog' part and the corresponding garbage collection techniques are standard; the latter are reviewed in Section 3.1. We will concentrate on the memory areas of the 'Tabling' part of the figure and the choice point stack because they are new to and slightly differ from that of the WAM, respectively.

As the memory snapshot of Figure 1 shows, the evaluation involved consumers, some of which, e.g., $C_2, C_3, \ldots$ 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 the CHAT architecture is not required; however, it is important to see how CHAT has arrived in this state and, more importantly, how execution might continue after a garbage collection. We thus describe the actions and memory organization of a CHAT-based abstract machine viewed from a heap garbage collection perspective.

Choice points and 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 subgoal 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 the H field of C, denoted by $C \rightarrow H$, to $G \rightarrow H$ 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 beginning of the heap till the dotted line). 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. Finally, in programs containing tabled negation or aggregation, a fourth type of choice point called a *completion suspension frame* might also exist in the stack. Since its treatment by the garbage collector is similar to that of consumer choice points we will not describe it in detail and will use the term *suspended computation* to refer to both types of suspension.

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

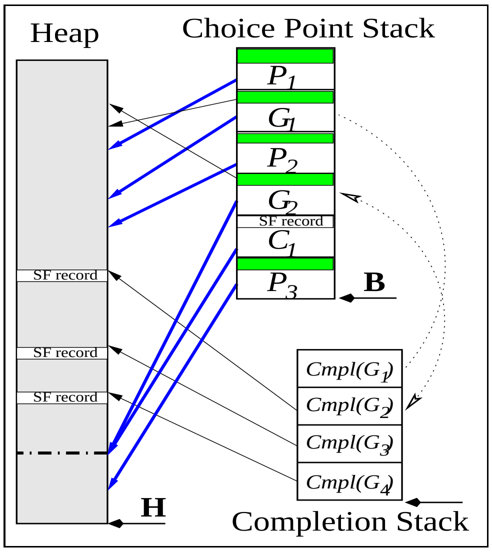
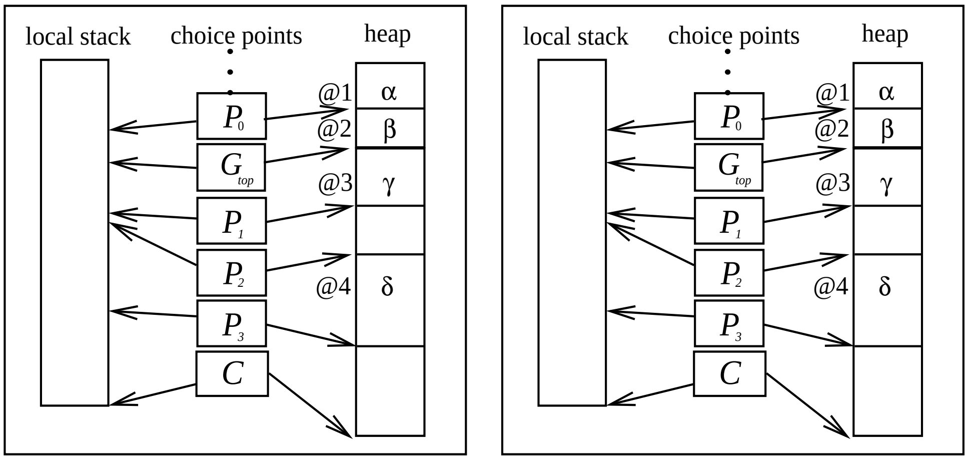


Figure 2: Detail of choice point and completion stack w.r.t. the heap.

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; in CHAT, the trail and choice point stacks can be reclaimed but e.g., the heap cannot: $G_i{\rightarrow}H$ 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: 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. The completion stack frame of a generator G_i is denoted by $Cmpl(G_i)$.

Substitution factors As noted in [RRS+99], 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 $Cmpl(G_i)$ points to SF; for consumers the substitution factor can be part of the consumer choice point as described in [RRS+99]; see Figure 2.

CHAT area The first time that a consumer is suspended, CHAT preserves its execution state through a *CHAT-protect* mechanism comprising of a freezing and a copying action. The *CHAT freeze* mechanism modifies H and EB fields in choice points between the consumer (which is at the top of the stack) and the topmost generator choice point to protect the heap and local stack that existed before encountering the consumer. In this way, it is ensured that parts of the heap and local stack that the consumer might need for its proper resumption are not reclaimed on backtracking. The idea of the CHAT freeze mechanism is illustrated in Figure 3; G_{top} denotes the topmost generator choice point. 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 a way that corresponds to the segmentation of the trail according to generator choice points; see [DS00] for a more detailed explanation. 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 4 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 upon garbage collection: they are the trail values



(a) Before CHAT freeze.

(b) After CHAT freeze.

Figure 3: Protecting heap and local stack information of a consumer through CHAT-freeze.

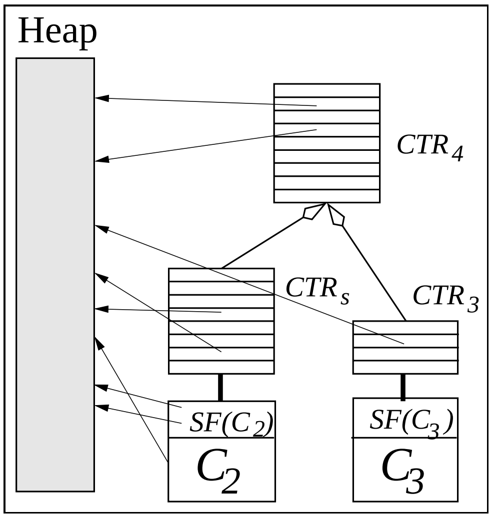


Figure 4: Detail of CHAT area.

of the CTR sub-areas, substitution factors of suspended consumers and the D field (the value of delay register as saved in each consumer choice point; see [SSW96]). Note that the H field of suspended consumer choice points that reside in CHAT areas can be safely ignored by garbage collection: this field gets a new value upon resumption of C and reinstallation of its choice point on the choice point stack.

It is important to note the following:

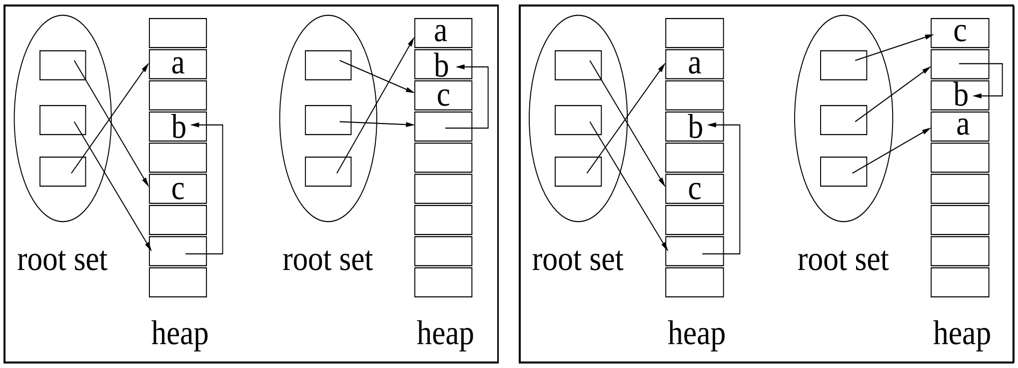
- The CHAT sub-areas are allocated dynamically and in non-contiguous space; how this influences garbage collection is described in Section 8.3.
- In CHAT, suspended computations have parts of their execution state (e.g., information about trailed binding) saved in a private area while other parts of the execution state (e.g., environments) are either shared or interspersed in the WAM stacks together with the state of the active computation.

3 Implementation of heap garbage collection in the WAM

We discuss issues that are relevant to heap garbage collection in any WAM-based implementation of Prolog: some of them are folklore but have not been published before as far as we know. We start by presenting a short overview of heap garbage collection in the context of the WAM.

3.1 Heap garbage collection in the WAM

Two garbage collection techniques are commonly used in WAM-based Prolog systems: mark&slide and mark&copy. At the higher level, marking makes explicit which cells are reachable so that this information can later be used in constant time and also counts the number of reachable cells: this number is needed in the sliding phase and is useful for copying. At the lower level, marking follows a set of root pointers and the Prolog terms they point to while setting a mark bit for each reachable cell. Marking can be performed by a recursive function, or by a pointer reversal algorithm [ACHS88]. The mark bit can be in the cells themselves, or in a separate data structure which is usually implemented as a bit array. Mark bits in the cells themselves



(a) Before and after a sliding collection. (b) Before and after a copying collection.

Figure 5: Heaps before and after sliding and copying garbage collections.

usually restrict the range of valid (heap) pointers in a system. Mark bits in a separate data structure enhance locality of reference because they can be packed together and thus their reading results in fewer cache misses than when they are in the header of the allocated objects: see [Boe00] for more details.

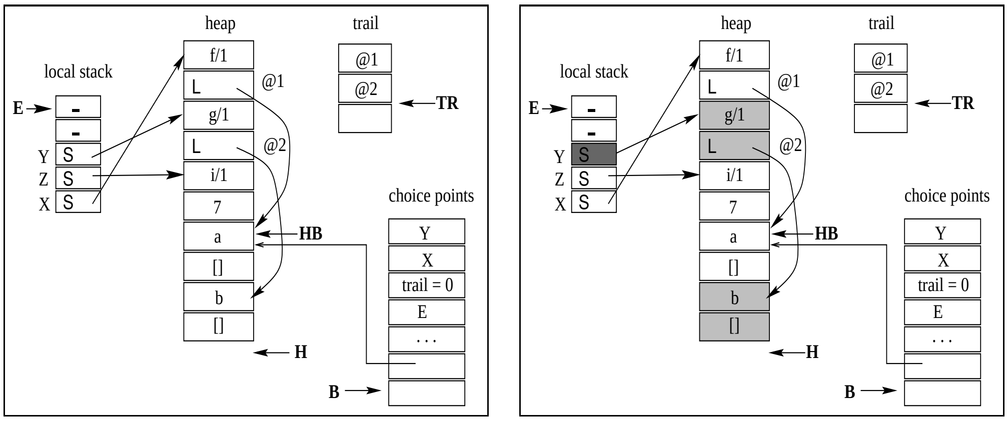
The sliding phase compacts the heap by shoving the useful cells in the direction of the heap bottom, thereby making a contiguous area free for cheap allocation by pointer bumping. At the same time the root pointers are relocated: Morris' algorithm [Mor78] is frequently used; see also Figure 5(a). Notable characteristics of sliding are that the order of cells on the heap is retained and that its complexity is linear in the size of the heap. When the percentage of useful data is low compared to the size of the heap, the sliding time can be reduced using the technique of [CMES00]. Apart from the mark bit, sliding also requires a chain bit.

A copying algorithm such as Cheney's [Che70] compacts the heap (named a *semi-space*) by copying the terms reachable from root pointers to a second semi-space. After the collection is finished, the computation happens in the second semi-space. Naturally, the allocation of new terms also happens at this second semi-space. Later collections will switch the roles of the two semi-spaces. The main characteristics of copying are: the order of cells on the heap is not necessarily retained; its complexity is linear in the size of the reachable (i.e., useful) data; the computation can use only half of the available heap space (because the semi-spaces exist at the same time). Assuming that the root set is examined in the order from top to bottom, the heap before and after a copying garbage collection is shown in Figure 5(b).

Sliding seems the natural choice in the WAM because, by keeping the order of cells on the heap, garbage collection does not interfere with the usual cheap reclamation of heap on backtracking. An additional advantage is that the (`compare/3`) order between free variables is not affected by such a garbage collection. So sliding has been the traditional choice for WAM-based systems. However, the ISO Prolog standard [ISO95] does not define the order between free variables, [BL94] has presented some evidence that the loss of cheap reclamation of heap on backtracking caused by a copying collector is not too bad, and finally [DET96] presents a practical segment order preserving copying algorithm that retains fully the backtracking reclamation of the WAM; see also Section 9. Together with the different complexity of a copying collector, the fact that no chain bit is needed, and that overall a copying collector is easier to implement, copying seems to be an attractive alternative to sliding for Prolog systems.

Marking starts in general from a *root set*. For a basic WAM implementation the root set consists of the environments, the choice points, the trail, and the argument registers. If only pointer reachability is used for marking, the order of traversing the root set is immaterial. Even in the WAM where every cell has its type (by the tag) such an approach is more conservative than necessary: e.g., the usefulness logic of Prolog execution teaches that marking becomes potentially more precise when marking is performed from the most recent part of the computation to the older part, and from the nearest (future) alternative computation to the later alternatives. Such order allows one to exploit early reset [BRU92]. Note that this order is also natural in the context of the WAM as the stacks are already organised in such a way to make this order of traversal of the root set possible. Finally, note also that *environment trimming* [War83] which is implemented in many Prolog systems (but not in XSB) is a way to make marking more accurate.

Figure 6 and subsequent ones show the situation of the stacks at the start of garbage collection (Figure 6(a)) and the different phases of the marking. The Prolog code corresponding to it is:



(a) Before marking. (b) After marking.

Figure 6: WAM stacks before and after marking the current computation.

```
main :- b(X,Y,Z), create_cp(X,Y), use(Z), gc_heap, use(Y).
b(f(_),g(_),i(7)).
create_cp(f([a]),g([b])).
create_cp(_,_).
use(_).
```

and the query is `?- main.` Here and in the sequel, we use the predicate `gc_heap/0` to indicate the point in the computation where garbage collection is triggered. In figures, we use `L` and `S` to denote list and structure tags of cells, respectively.

Marking the current computation consists of marking the argument registers first (there are none in this example) and then starting from the current environment (indicated by **E**) and the current continuation pointer, **CP**, which points into the code of the `main/0` clause just after the goal `gc_heap`. In this example, the **CP** indicates that only `Y` is active in the forward computation. In the WAM, the call instruction has as argument the length to which the current environment can be trimmed: this can be used as an approximation of the set of live environment variables. A more precise method uses live variable maps (see Section 3.3) as in MasterProlog or Yap. Less precise methods might have to resort to initializing environment variables. More alternatives and their pros and cons are discussed in Section 3.3. Figure 6(b) shows the stacks after marking; the marked heap cells are shaded.

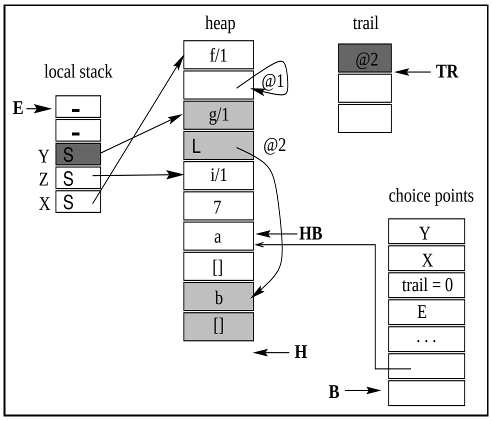


Figure 7: Stacks after early reset.

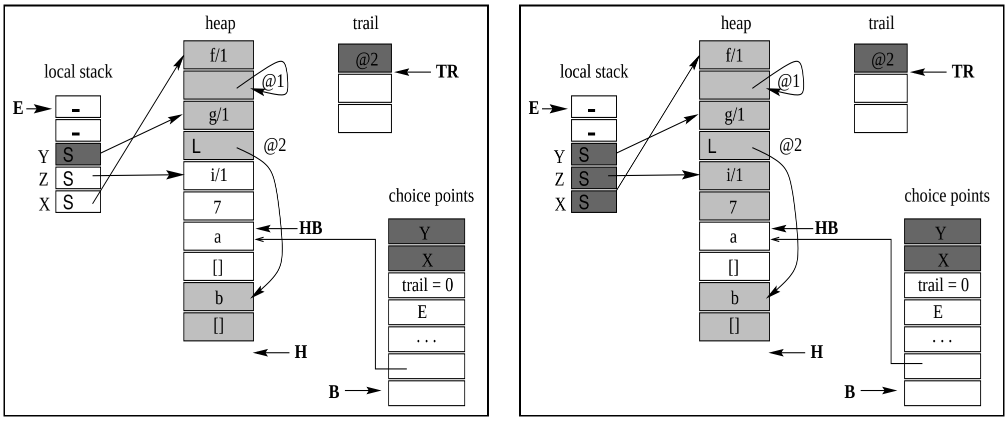
Root cells in other stacks that have been treated are shown shaded with a darker shade.

Now comes early reset: the part of the trail between the current trail top **TR** and the trail top saved in the current choice point indicated by **B**, is used not for marking, but for resetting the cells that are not marked yet, but pointed to from the trail. The corresponding trail entry can be removed. Figure 7 shows that the cell at address `@1` is pointer reachable, but unreachable in forward computation because it is not (yet) marked. Since the trail points to it, it can be reset to a self-reference.

The next step consists in marking the arguments saved in the choice point: these arguments are needed on backtracking. Since `Y` has been marked before, there are no changes there. `X` is marked. Figure 8(a) shows the resulting state of the stacks.

Then follows the marking of the failure continuation, or put in a different way: the forward computation that will result from backtracking. The **E** and **CP** from the choice point are used as a starting point. This **CP** points into the `main/0` clause just after the call to `b/2`: it indicates that there are 3 live variables in the environment. Consequentely, `X`, `Y`, and `Z` are marked; see Figure 8(b).

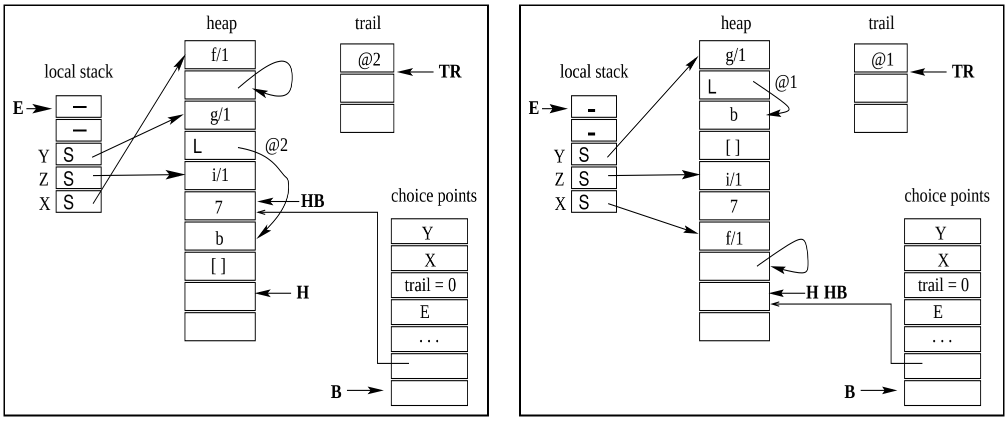
Finally, Figure 9 shows the heap and local stack after a sliding (Figure 9(a)) and after a copying (Figure 9(b)) garbage collection. The copying figure results from treating the root pointers in the environment in the order `Y, Z, X`. Other orders of treating the root pointers will result in heaps with different orders of cells that survive garbage collection. Note that in



(a) After marking the choice point arguments.

(b) After marking of the failure continuation.

Figure 8: Stacks after marking the choice point arguments and the alternative computation.



(a) After a sliding garbage collection.

(b) After a copying garbage collection.

Figure 9: Heap and local stacks after sliding and copying garbage collections.

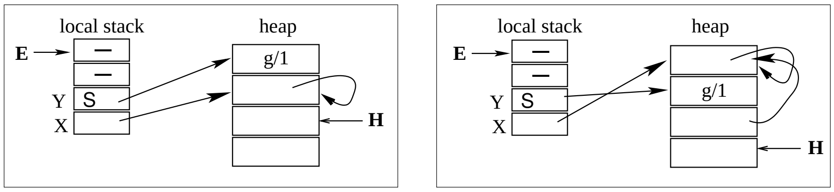


Figure 10: Double copying of an internal variable.

the copying picture, the **HB** pointer now equals **H** and that the H field of the choice point has been adapted to point to the top of the heap. If there would have been more choice points, their H fields would all be equal to **H** at this point, which is exactly why reclamation of heap on backtracking is prevented after a copying garbage collection.

Both the sliding and the copying collector can be enhanced with generations: a minor collection will collect only the new generation and ignore the older generation that has survived collection. The WAM caters in a natural way for the *write barrier* (in the WAM, **HB**) and the *exception list* (in the WAM, the trail) known in other implementations, but both deterministic computations and backtracking complicate the issues.

A final point about marking: usually copying schemas do not mark before copying. Bevemyr and Lindgren in [BL94] point out that it is necessary to do so in the WAM, because otherwise, one runs the risk that free variables are copied twice. Figure 10 shows this phenomenon. When first `X` and later `Y` is copied, we end up with a heap whose size is *bigger* than before! This situation seems rare in practice however: [Zho00] presents the implementation of a copying collector that does not mark first, but postpones the treatment of such variables. Empirical data indicates that the number of postponed variables is small in the context of the incremental collector of [Zho00]. Whether this is true in general remains to be seen. The remaining part of this section discusses implementational aspects of heap garbage collection in the WAM.

3.2 The test for heap overflow

Like many other (logic) programming systems, XSB relies on software tests for stack overflows. In fact, an older XSB version was checking for heap overflow 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 WAM 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.

To support and trigger the garbage collector, the XSB compiler was adapted so as to generate the instruction `test_heap` with a fixed margin. A related change was 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. This change is reasonably straightforward 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. Such a built-in occurred in the treatment of tabled negation, as the `negation_suspend` built-in predicate lays down a completion suspension frame in the choice point stack (cf. Section 2) and subsequent garbage collection would not do appropriate marking without modification. In XSB, we have simply disabled the inlining of this particular built-in, because it was not time critical.

Instead of checking for heap overflow, some systems rely on hardware assistance to detect overflow by allocating a (read- and write-) protected page just after the heap. The advantage is that software overflow tests are no longer needed. However, in the context of precise garbage collection, it becomes more difficult to deal with a heap overflow in this schema, as the interrupt can occur at any moment during head unification or during the construction of the body arguments. Software tests, on the other hand, make the implementation easier. Moreover, the overhead introduced by the software check in the schema described above, is usually neglegible in non-toy benchmarks.

3.3 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 Prolog versions 3 and earlier; see also [Car90]) 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, as noted in Section 3.1, most Prolog systems have no explicit information about which permanent variables are alive at a particular moment in the computation. This information is implicit in the WAM code. In such a system one faces a choice between the following four 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 precise information about the liveness of permanent variables. Live variable maps, introduced in [BL71], are commonly used. Basically, the compiler emits for every continuation point at which garbage collection is possible, information about the set of live (i.e., initialized and not yet dead) stack slots (and possibly also registers). In the setting of an abstract machine like the WAM, generating this information is straightforward. This method is currently used in MasterProlog and in YAP. Techniques for compacting live variable maps also exist; for a recent one see [Tar00].
4. Reconstruct the liveness information through scanning or de-compilation of the WAM code. The advantage of this alternative is that there is no time or space cost when garbage collection is not triggered. It is currently planned that SICStus version 4 will be using this method[2]. An early version of the BIM-Prolog garbage collector was using this method but it was later abandoned in favour of live variable maps.

In XSB, we have opted for the first solution because it was the quickest to implement by extending the `allocate` instruction.

3.4 H fields in the choice point stack

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

[2]Mats Carlsson, personal communication.

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 cell that was not marked; such is e.g., the case in Figure 8(b). 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, which is the one used in Figure 9(a) (compare it with Figure 8(b)) 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. This is because the sliding phase has to test continuously whether an old segment boundary is crossed. We prefer the first method and in our implementation used the tagged integer `666` as the atomic object. We illustrate it in Figure 11: Figures 11(a) and 11(b) show the WAM stacks after applying the method to the stacks of Figure 8(b) and after the sliding collection is finished, respectively.

The correctness of this operation in plain WAM is based on the observation that the action

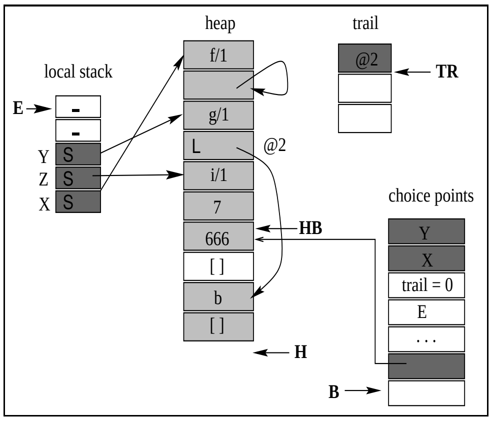
$$\texttt{if}\ (!\ \texttt{marked}(B{\rightarrow}H))\ \{\ *(B{\rightarrow}H) = \texttt{tagged\_int(666)};\ \texttt{mark}(B{\rightarrow}H);\ \}$$

can be performed as soon as it is sure that the cell at address $B{\rightarrow}H$ will not be marked later (i.e., will not be found useful) for some other reason.

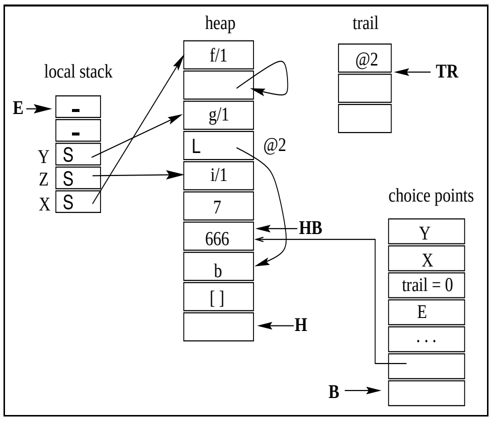
3.5 Trail compaction during garbage collection

Trail compaction is possible whenever early reset is performed (cf. Section 3.1). In fact, Figures 7–9 and 11 all show the trail after it has been compacted. 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 cell.[3] Since reserving such a cell would have involved changes elsewhere

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



(a) After marking the H field of choice points.



(b) After a sliding garbage collection.

Figure 11: Heap and local stacks during and after a sliding garbage collection in our implementation.

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 performance results of Section 11 as in our measurements we counted these cells as garbage.

3.6 Chain bits for dynamically allocated root pointers

The XSB cell representation in one machine word cannot cater for the extra bits which the garbage collector needs. In particular, when implementing the mark&slide collector we were faced with the problem of 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, 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 in our implementation), 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. In Section 8.3 we discuss in more detail how we dealt with this issue.

3.7 The copying collector

The copying collector was implemented following the ideas of [BL94]. Since we perform a marking phase, we know exactly how large the *to*-space needs to be to hold the copy of the useful data. We thus deviate slightly from the usual two-space schema of Cheney [Che70], by allocating at the beginning of the copying phase a *to*-space of exact size. This *to*-space thus lies in an area independent of the WAM stacks. After having copied the non-garbage to the *to*-space, we copy it back as a block to the original *from*-space and release the allocated *to*-space. This has a principled reason: this implementation scheme uses memory resources more economically because usually, the *to*-space can be significantly 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 as one area 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 [DET96]. Also, in a generational schema, such copying back is almost a necessity.

4 The impact of tabling on memory consumption

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. We discuss here shortly how tabling affects heap consumption in essentially three ways:

1. Suspension of consumer subgoals freezes parts of the heap which cannot be reclaimed until completion of (i.e., generation of all answers for) these subgoals has taken place.

2. Tabling can avoid repeated sub-computations that require considerable heap. This can lower the heap consumption arbitrarily.

3. Tabling can diminish sharing and in this way increase the heap consumption arbitrarily.

The following definition of a predicate `p/3` will help in illustrating the last two points:

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

The example that shows point 2 above is obtained by defining:

```
trans(X,X) :- do_some_memory_intensive_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 $O(N)$, because the term that `NewTerm` is bound to is repeatedly copied out of the tables, 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.

5 Heap garbage collection in the WAM with tabling

In this section we will concentrate on high-level aspects of heap garbage collection for tabled systems. The common situation is as follows: the current computation needs to be collected and there are one or more consumers suspended. As mentioned, the issues to consider are: 1) how to find the reachable data, and 2) how to move it appropriately while adapting all pointers to it. The root set in WAM consists of the argument registers, the saved argument registers in the choice points, and the local variables in the environments reachable from the current top environment or from the choice points. Marking is done in a WAM heap garbage collector by considering this root set from newer to older (mainly because that is the way frames are linked to each other) and in the set of *backtracking states*[4] from current to more in the future: this latter order corresponds to treating younger choice points (and their continuation) before older choice points.

5.1 Heap garbage collection for copying implementations of tabling

A possible WAM-based implementation of tabling is one that preserves execution states of consumers by copying them from the WAM stacks to a separate memory area, and reinstalls them by copying them back from this area to the WAM stacks. Consider first an implementation that, for both suspension and resumption, totally copies the execution state of each consumer; this implementation is not practical, but it helps to illustrate the issues. In such an implementation, it is clear that it is possible to collect just the current computation; i.e., perform garbage collection only in the WAM stacks and leave the memory areas of suspended consumers unchanged. Since it is the current computation that overflows, this seems the natural thing to do, especially since the current and the saved (suspended) computations might be unrelated. Furthermore, the consumer might never need to be resumed and thus avoiding the collection of its state might pay off. It is also clear that in this implementation scheme heap garbage collection happens exactly as in the WAM.

Now consider an implementation that incrementally copies the relevant part of the heap corresponding to each suspended consumer, such as CAT [DS99]. In such a scheme, consumers have parts of their heap (i.e., the heap which is needed for the proper re-installation of their execution environment) privately saved, but also share a part of that heap with the current computation: the part that is older than the heap point up to which their state has been copied. Several pointers to this shared part of the current computation might exist from the partially copied state of the consumers. These pointers need to be followed for marking and take part in the usual relocation during the chaining or forwarding phase of garbage collection. Note, however, that in such a scheme marking can be performed *without* reinstalling the state of

[4] A *backtracking state* is the continuation of the computation which results from backtracking, possibly more than once. We have so far referred to the first occurrence of a backtracking state as a *failure continuation*. A backtracking state is represented implicitly in the WAM stacks by a choicepoint, the chain of environments reachable from that choicepoint and the reset information of the trail younger than that choicepoint.

each consumer back on the stacks and that root cells in the shared part of the current computation need not be followed for marking more than once. Finally, note that there is a choice on whether to collect the parts of the consumers' heap that have been saved through copying: depending on the policy used and the expected number of consumer re-installations, garbage collection might also involve collecting the privately saved heap of none, some, or all suspended consumers.

5.2 Heap garbage collection for heap-sharing implementations of tabling

In tabling implementations that freeze the heap, such as CHAT or SLG-WAM, it seems difficult to collect the current computation without collecting the suspended consumers at the same time: their data are intertwined on the heap. So, the right approach here is to use as root set the current computation (and its backtracking states) as well as the suspended consumers. Each consumer is characterized by its choice point together with a part of the local stack and trail that is newer than the generator of its answers. As mentioned in the introduction, before CAT was conceived (i.e., when SLG-WAM was the only known way of efficiently implementing tabling in the WAM) our understanding of the usefulness logic of tabled logic programming system was quite poor: the only way we could envision marking the state of a frozen consumer was by melting it first (i.e., reinstall its bindings by using the SLG-WAM forward trail), and then mark it as if it were a current computation. However, a previous marking (during the same garbage collection, either a marking of another consumer or the current computation), would have set some mark bits, and it was not clear how they had to be taken into account.

Such a situation happens when there are two views on the same cell, depending on two separate threads of execution. In these views, the cell has different values and in one view, usefulness logic dictates that marking is not recursive if the envelope of the cell was not marked before. Figure 12 shows this situation in more detail. Consider that the box around the smaller box (representing e.g., a structure) is reachable from both views 1 and 2. In view 1, the cell in the smaller box (e.g., an argument of this structure) has contents `A`, in view 2, `B`. Suppose that view 2 is followed for marking first; then the enveloping box has not been marked before and the object `B` is alive

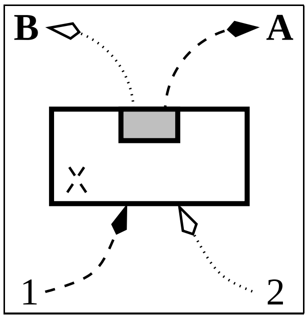


Figure 12: A cell reachable through two different paths.

according to the usefulness logic. However, marking from view 1 first sets the mark bit of the enveloping box on; when later marking from view 2 happens, `B` will be considered dead.

This is exactly the situation when the suspension/resumption mechanism that tabling needs enters the picture. Then, different consumers and the current computation correspond to different threads of execution, each having their own view on the contents and liveness of reachable cells.

In the setting of a copying implementation of tabling, such as CAT, it was easier to arrive to the conclusion that a consumer can be marked without being reinstalled, that marking a consumer does not need to consider the part of the stack that is older than the generator up to which the copy was made, and that even the choice points between the consumer and the generator need not be considered. Indeed, in CAT, at the time of reinstalling the consumer, these intermediate choice points have already been removed by backtracking.

In the setting of CHAT and SLG-WAM, two consumers can also share a part of the local stack, so we expect that it pays off to avoid marking from the same roots (in this shared part) more than once. Similarly for the shared part of trail in SLG-WAM. It is also common in a plain Prolog garbage collector to avoid marking from the same environment cell, which can be reachable from different choice points.

6 Early reset and the order of marking

As mentioned in the introduction, *early reset* in the context of tabled logic programming remained long a mystery to us. For a good account on this issue in Prolog, see [BRU92], or [ACHS88] and [PBW85]. The series of figures in

Section 3.1 also shows early reset.

In a CAT-based implementation, it is clear that during garbage collection, early reset can be performed for the current computation. The reason is that the suspended computations, i.e., the consumers, have their computation state privately saved and the stacks collapse to exactly those of the WAM for which early reset is possible. If this applies to CAT, it has to apply to CHAT and SLG-WAM as well, as the matter of usefulness of data is related to the abstract suspension mechanism rather than to its actual realization. However, there remains the question whether during marking of the consumers early reset is allowed and/or possible.

6.1 Order of early reset between current and suspended computations

A CHAT trail entry of a consumer, say C, can contain a reference to a heap or local stack entry that is not reachable from the continuation of C: there is no harm in removing this CHAT trail entry. However, setting the corresponding heap entry to undefined in a non-discriminating way might be wrong, since the current computation might need the cell with its current value. In that case, and if marking of the current computation were performed first, the mark bit of this trailed heap entry would be set and the marking phase of the consumer could decide not to make the cell undefined.

On the other hand, the mark bit does not reflect *who* (the consumer or the current computation) references the cell, so that it is possible that by marking from the current computation first, the consumer loses an opportunity for early reset—or at least trail compaction of the CHAT trail areas. This means that the order in which consumers and the current computation are marked can be crucial. For the sake of focusing on the ideas, assume that, at the moment of the garbage collection, there is the current computation and that the set of suspended computations consist of just one consumer. Obviously a generator choice point, which can schedule the consumer (some time in the future), is in the current computation at that moment. Two reasonable orders of marking are:

1. First mark the current computation completely and then the consumer.

2. Interleave the marking of the current computation and the consumer in the following way: first mark the current computation up to the

generator, then mark the consumer up to the generator; finally, mark the rest of the current computation. Since the consumer shares this latter part with the current computation, there is no need to go back to marking the rest of the consumer.

Both methods can be easily generalized to the situation in which there is more than one consumer.

The second order of marking is more precise as it is possible that, by first marking the current computation completely, some opportunity for early reset is lost for one or more consumers. For the first order, the only way to remedy this, would be to have a mark bit for each consumer, which is of course impractical since the number of consumers can be large. Since the additional benefit of performing early reset in the WAM is usually small (cf. also Section 10), we prefer the first order of marking because of the simplicity of its implementation.

Because of mark bits set earlier during the marking of the current computation, early reset is different for the current computation and for suspended consumers. This is described in more detail:

Figure 13 shows a pointer tr which points to a CHAT trail entry: the value cell of the trail entry contains a value v_1; the reference cell of the trail entry is a and points to a heap (or local stack) location which contains a value v_2: in general $v_1 \neq v_2$.

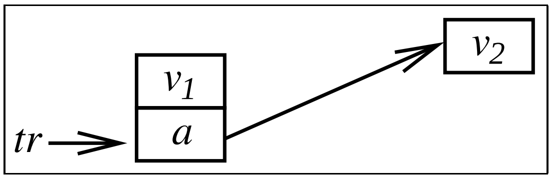


Figure 13: A CHAT trail entry.

Figure 14 contains some pseudocode for treating one trail entry during marking: the code is different for the current computation and for the consumer and assumes that the current computation is marked first. Function `mark_object()` marks a value representing a Prolog object. From this piece of code, one can deduce two ways in which early reset for a consumer is suboptimal. If a heap (or local stack) cell reachable from the trail is marked already, this can be due to two reasons:

1. the cell was reachable from the consumer itself

code for current computation	*code for each suspended consumer*

`if (! marked(`a`))`
`{ *`a` = UNDEF;`
`remove_trail_entry(`tr`);`
`}`
	`if (! marked(`a`))`
	`{ *`a` = UNDEF;`
	`remove_trail_entry(`tr`);`
	`}`
	`else /* marking is necessary`
	`* in general */`
	`mark_object(`v_1`);`

Figure 14: Pseudocode for performing early reset during marking of the current computation and of consumers.

2. the cell was reachable from another consumer or the current computation.

These two cases cannot be distinguished if there is only one mark bit. As a result, in either case, not only will early reset be prohibited, but also the data structure v_1 needs to be marked. Note that this would be unnecessary if the cell had not been reachable from the consumer one is currently marking. It follows that only the current computation can do optimal early reset; the suspended consumers approximate it. Another conclusion is that early reset during marking of the suspended consumers can reset the heap entries and remove CHAT trail entries.

The above analysis was made for CHAT. It can be applied in an almost straightforward way to SLG-WAM: after marking the current computation—performing early reset in the course of doing so—one can mark (and early reset) the state of the consumers, which is captured by the cells reachable from the consumer choice point and the part of the trail starting at the consumer up to where it meets the trail of the generator that can schedule the consumer. The part of the trail older than this generator will have been marked already, since the consumers that are scheduled by this generator share the trail with the generator. Figure 13 can also serve as a picture of an SLG-WAM forward trail entry (not including the back pointer): for the current computation, it is always true that $v_1 == v_2$. Accordingly, the code above applies to SLG-WAM as well.

6.2 Performing early reset when trail chunks are shared

The previous section explained how to mark the suspended consumers (in the CHAT area) *after* the marking of the current computation. But 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 4 shows, in CHAT trail entry chunks are possibly shared between several suspended consumers. 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 such consumer C mark the reachable environments and heap; only *after* this operation is finished for each C mark and early reset the trail of C. It is indeed possible to have two suspended consumers which share a part of the trail while a trailed variable is reachable in the forward computation of the first consumer but not of the second. This is illustrated by the following example which exhibits this situation for a trailed variable in the local stack; similar examples can be constructed for trailed heap variables.

Example 6.1 In the program of Figure 15, 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 the query `?- main.` 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 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, $\mathtt{c}_{G_2}$`(Z)`, and, upon backtracking out of the choice point of this generator, they need to have their state CHAT-protected till their own generator $\mathtt{b}_{G_2}$`(_)`. In particular, the part of the trail that lies between $G_1 \rightarrow TR$ and $G_2 \rightarrow TR$ is saved (as a value trail) *once* and is shared between the two consumers. The situation is pretty much that depicted in Figure 4 but in a more simplified form; it is shown in detail as part of Figure 15. 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 failure continuation) and its binding can be early reset as far as C_1 is concerned. However, this action is wrong as this

main :- b_{G_1}(_).	create_cp.
use(_).	create_cp :- fail.
:- table b/1. b(A) :- a(_), A = a. b(b).	:- table c/1. c(1). c(2).
a(_) :- create_cp, X = xxx, c_{G_2}(Z), (Z = 1 -> use(X), b_{C_1}(_) ; Z = 2 -> b_{C_2}(_), use(X)). a(_) :- gc_heap.	

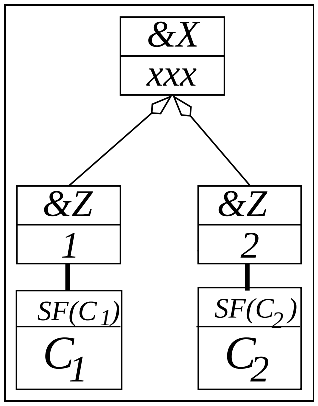


Figure 15: Program exhibiting trail sharing between suspended computations in CHAT.

variable (and its value) is also used in the forward continuation of C_2 whose suspended state shares this part of the trail. The safe thing is to postpone early reset from suspended consumers until after marking of the heap and local stack has taken place for all such consumers.

One way to understand this is as follows: two consumers are always each others' failure continuations, as one can be scheduled after the other. If two consumers share a part of the trail, early reset in one of them can influence the other. This means that the early reset of the part of the computation which corresponds to this shared trail must be postponed.

7 Variable shunting

7.1 Variable shunting in the WAM

Variable shunting in the WAM consists in finding variables which are only visible as bound, and replacing references to such variables with their binding values. This way, chains of variables, which are unavoidable in 'plain' WAM, are short-circuited and subsequent dereference operations on these variables become more direct and thus more efficient. In addition, if the intermediate cells are not referenced from anywhere else, they can be reclaimed. Usually, shunted variables are those for which no choice point exists between variable creation and binding time; see [BRU92]. Variables that are trailed require special treatment.

We illustrate a simple form of variable shunting with an example. Let us use the predicate `var_shunt/0` to indicate the point in the computation where variable shunting is performed. Consider the execution of `?- main.` against the program:

```
main :- h(T,A,B,C), C = B, B = A, A = x, var_shunt, use(B,C).
h(f(X,Y,Z),X,Y,Z).
use(X,X).
```

Before performing variable shunting, the heap contains two chains of variables which are bound to the constant `x`. Variable shunting eliminates these chains and allows a direct access to the value `x`; the call to `use/2` will be more efficient. The situation is illustrated further in Figure 16. In this example, no reclamation of space becomes possible due to variable shunting.

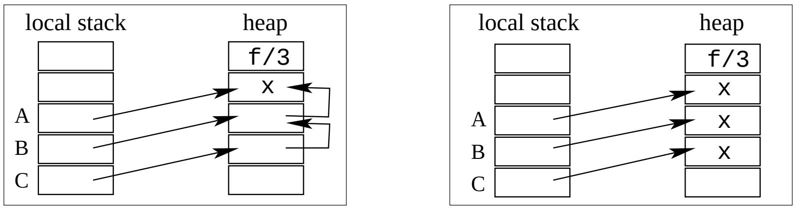


Figure 16: Local stack and heap before and after variable shunting.

As noted in [SC91], variable shunting is an operation which in principle is independent of garbage collection and, in fact, there are trade-offs involved in performing it before or after garbage collection. However, in most systems, variable shunting is—when implemented—implemented as part of the garbage collector (e.g., in SICStus, MALI [BCRU86]). Although in contrived examples an unbounded amount of memory is needed if no variable shunting is performed (cf. [SC91]), most implementors believe that in most 'plain' Prolog programs variable shunting is not really worth its while: the overhead of performing variable shunting usually outweights its benefits and the extra execution speed due to faster dereferencing is negligible in most programs because most variable chains are short. On the other hand, variable shunting is considered as potentially very important in implementations that support *attributed variables* or *mutable terms* [LH90, Neu90]. For example, variable shunting can straightforwardly reclaim the space occupied by the 'attribute' part of an attributed variable when that variable is shunted; see also [BRU92, Sections 4–5]. However, to the best of our knowledge, this conventional wisdom has so far not been experimentally backed up in a systematic way.

7.2 Variable shunting in the WAM with tabling

As the discussion with early reset has shown, the current computation, when treated first, can make changes to the state of the WAM stacks as long as these changes are consistent with its forward and failure continuations (suspended consumers can also be considered as failure continuations of the current computation). The same treatment of the current computation holds for performing variable shunting as well. As for the possibility of doing variable shunting for the consumers, similarly to the case of early reset, one must respect the following obvious rule: a consumer cannot change anything that another consumer might need. In the case of early reset, this rule resulted in the safe implementation described earlier: a consumer can remove an entry from its trail, but it should not reset a cell in a shared part of the computation. Early reset in the shared part of the computation can occur only when all consumers agree with it—a condition which is usually too costly to test. Similar observations hold for performing variable shunting.

However, variable shunting differs from early reset in that it needs to consider some additional issues: The algorithm for performing variable shunting in the WAM, described in [SC91], proceeds by considering the choice points from older to newest (i.e., following the directionality of variable bindings—notice that this order is the opposite of that of the garbage collector marking phase) and traverses at some point the corresponding heap segments to determine which variables are considered as having received their value at the segment under consideration. This information is needed to shunt variables at this time. This traversal, especially determining that values are considered to be created at a given heap segment, is difficult to do for a segment which is shared. This is because each suspended consumer can have a view of the heap through its private trail which is totally different from the view that the current computation or other consumers have. This means that in order to know the contents of a cell on the heap, the consumer's saved trail needs to be inspected, or that, alternatively (and more efficiently), the consumer should be reinstalled.

When performing variable shunting, the changes that can be made in the consumer's private view of the shared heap, are twofold:

1. A consumer can change its view of the shared heap through modifying its value trail; in principle this action does not affect other consumers.

2. A consumer can change cells on the shared heap; as argued above, such

changes must be agreed upon by all consumers, or they might cause problems.

Since it is difficult to ensure or validate the latter agreement (as mentioned, it requires a mark bit for each suspended consumer), it seems best to perform shunting only for trailed cells; i.e., those for which a consumer has a private value. We illustrate the issues with an example.

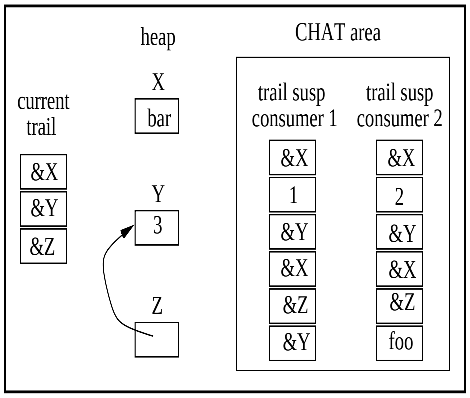
Example 7.1 Figure 17 shows a relevant part of the state of the computation just before (Figure 17(a)) and after (Figure 17(b)) performing variable shunting which takes place at the moment of the goal `var_shunt` which is part of the current computation in the program below:

```
main :-
   L = [X,Y,Z],
   (  Z = Y, Y = X, X = 1, susp_consumer_1, use(X,Y,Z)
   ;  X = Y, X = 2, Z = foo, susp_consumer_2, use(X,Y,Z)
   ;  X = bar, Y = Z, Z = 3, var_shunt, use(X,Y,Z)
   ).
```

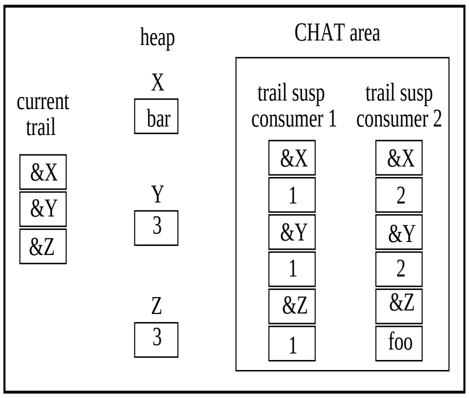
The goals `susp_consumer_1` and `susp_consumer_2` denote suspended consumers. As part of the list construct, the variables `X`, `Y`, and `Z` reside in the shared heap. Each consumer has its own private view on these variables through its saved trail (in the CHAT area) and this view differs also from that of the current computation. Variable shunting in the current computation alters the heap, while variable shunting in the suspended consumers results in the adaptation of the consumers' private trails. The example is quite simple because all the bindings have the same time stamp, i.e., they occurred in the same segment.

8 Implementation of heap garbage collection in the WAM with tabling

As described in Section 5, the usefulness logic of tabled evaluation 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 Section 6, one can perform the marking phase of garbage collection *without* reinstalling the execution states of the suspended computations on the stacks. Moreover, the marking phase of a suspended



(a) Before variable shunting.



(b) After variable shunting.

Figure 17: Shunting in the current computation and in suspended consumers.

consumer C needs to consider neither the part of the heap that is older than the generator G up to which C has its execution environment CHAT-protected, nor any choice point between **B** (WAM top of choice point stack) register and G. 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 issue is again related to the order of marking the state of 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.

8.1 Marking of substitution factors and 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 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 [SS98, SSW96] 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 `?- main.` w.r.t. the tabled program given below. 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; see [SS98]. In such a situation, a Prolog garbage collector would invoke early reset of `X`. This is correct as, according to 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 on the right) was not designed to reflect the usefulness logic of tabling. As explained in [SS98], 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.

XSB program	XSB abstract code
`main :- 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 extra compiler support is desirable but not strictly required: the 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

[5] As a general comment, it is not unusual that garbage collection requires compiler support; see e.g., the missing support in the WAM for initialization of local variables described in Section 3.3.

completion stack—this is where CHAT keeps a pointer to the substitution factor record of generators (cf. [DS00] and Figure 2). In this way, problems caused by this kind of premature early reset are avoided. 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; see [SSW96].

8.2 H fields in suspended consumer choice points

As mentioned in Section 3.4, H fields of choice points on the choice point stack need special treatment. On the other hand, 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. Since according to Section 6 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 the above action (on the choice points in the active computation) 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.

8.3 A chain bit in cells of the CHAT area

As Figure 4 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 implemented 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 make sure that each group of $(S+1)$ words is aligned on a $(S+1)$-boundary. S was chosen as `sizeof(Cell *)` that is the size of the pointer type used for cells of the WAM stacks and actually, for convenience, the garbage collectors in XSB version 2.2 use a byte for each chain bit in CHAT areas. A pointer p to a CHAT object, can then be translated to a pointer to its chain byte as follows. Let

$$i = ((((\texttt{int})p)/S)\%(S+1)),$$

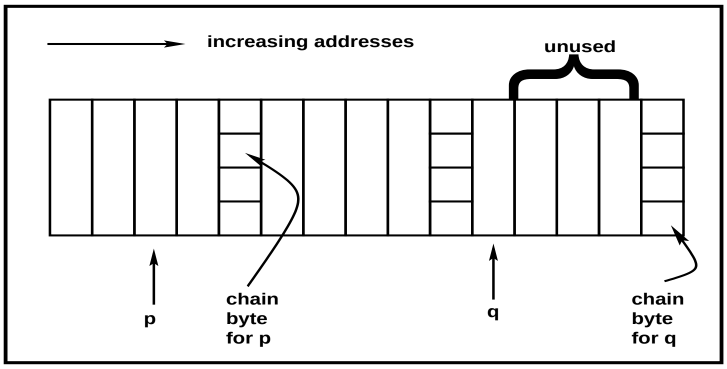


Figure 18: Chain bytes.

then

$$pointer\_chain\_byte = (\texttt{char} *)(p + S - i) + i.$$

Figure 18 illustrates this for $S = 4$ and $N = 9$.

8.4 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 [RRS$^+$99, SS98]), 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, i.e., untagged data.

The first solution that comes to mind is to let the collector check for the type of choice point: it can be deduced from its alternative field. This makes maintenance error-prone and is 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 in the choice points.

9 A segment order preserving copying garbage collector for Prolog with tabling

[DET96] describes a segment order preserving copying garbage collector for BinProlog. Although it is not as simple to implement as a 'plain' copying collector *àla* [BL94], such a collector is quite appealing as it combines at a very small cost the advantages of garbage collection based on copying with the full ability of the WAM to perform cheap reclamation of the heap upon backtracking. So it is interesting to consider adapting it to a system that supports tabling such as XSB. Indeed, in this section we will show that it is perfectly possible to do this type of garbage collection in the context of a WAM-based tabled abstract machine without losing any of its advantages.

There are two differences between XSB and BinProlog that need to be addressed: one concerning the 'Prolog' part of XSB's abstract machine and one to the 'Tabling' part (cf. Figure 1). First, contrary to XSB, BinProlog has no local stack and consequently [DET96] does not describe how to deal with this stack. Secondly, we must explain what preservation of segments means in the context of suspended computations and how garbage collection should be performed in order to ensure this property. Note that the first issue is not XSB-specific; indeed the following solution is applicable to every WAM implementation with a local stack.

9.1 Dealing with the local stack

BinProlog is based on BinWAM [TN94] and does not have a local stack: because of binarization, the equivalent of environments resides on the heap instead. This simplifies the implementation of the garbage collector as it avoids the (engineering only) issue of traversing the local stack in the order from older to newer segments. Recall that the marking phase must be carried out from newer to older, because otherwise early reset becomes ineffective; see also Section 3. The reverse traversal is necessary during the *copying* phase and in [DET96] is only performed for the choice points. The same reverse traversal over the choice points can be adapted slightly to do the reverse traversal of the environments as shown in the code shown in Figure 19.

Maintaining `prev_e` avoids traversing the same environment repeatedly. Note that it is not necessary to traverse environments in the order from older to newer: indeed, in the set of environments belonging to a particular segment (i.e., the part of the computation between two successive choice points) the order of treating the environments is immaterial and thus it appears most natural to treat environments in the order that the WAM has already linked them. On the other hand, choice points have to be traversed in an order from older to younger: the `next()` function achieves this. Its functionality can be implemented using a reversal of the links of choice points as described in e.g., [DET96]. Note that this is the same kind of traversal

```
prev_e = 0;
b = oldest_choicepoint;
while (b)
  {
    e = b->ereg;
    cp = b->cpreg;
    while (e > prev_e)
      {
        treat(e,cp);
        cp = e->cpreg;
        e = e->ereg;
      }
    prev_e = e->ereg;
    b = next(b);
  }
```

Figure 19: Handling of the local stack.

which is needed to perform variable shunting in the WAM (with or without tabling); cf. Section 7. Finally, function `treat(e,cp)` is meant to copy the permanent variables from `e` that point into the heap segment that is currently considered: to determine whether a heap pointer points in this segment, a simple range test is enough; in fact, this is the small cost mentioned earlier.

The example program below shows that one environment can contain pointers to different heap segments. Figure 20 illustrates it further. Consider the execution of `?- main.` against the program:

```
main :- eq1(X), create_cp, eq2(Y), gc_heap, use(X,Y).
create_cp.
create_cp.
eq1([a]).
eq2([b]).
```

Variables `X` and `Y` belong to the same environment (of `main/0`) but `X` points to a segment S_1 which is older than the choice point of `create_cp/0`, i.e., the part of the heap that is above **HB**; `Y` belongs to a segment S_2 which is younger: in the figure S_2 is the part between **HB** and **H**. When segment S_1 is treated during the copying, `Y` should not be copied, as otherwise `Y` would move to an older segment. The treatment of `Y` is postponed until it is found

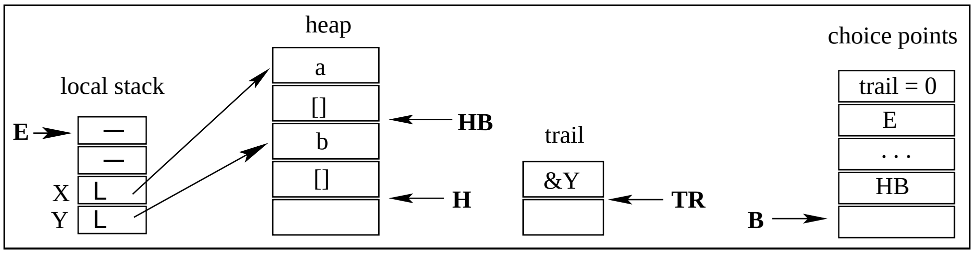


Figure 20: Two local variables in the same environment pointing to different segments.

on the trail: it necessarily occurs there, because `Y` is part of a WAM area (environment in this case) which has the same age as S_1, but contains a value that has the age of the newer S_2. The generalization of this situation can be found in [DET96]. The mechanism of postponing the treatment of pointers from the local stack to a segment newer than the one which is currently treated is actually completely similar to what is described in [DET96] for the heap only.

9.2 Dealing with suspended computations

The following reasoning is particular to CHAT: let G be a generator and $C_1, C_2, \ldots, C_n$ the set of consumers for which their state is currently saved (by CHAT protect: CHAT freeze & CHAT copy) up to G. When a consumer C_i is reinstalled (by G), it will get the H pointer for the corresponding field of its choice point from G, so one can consider all of the heap reachable from C_i as being older than G. Also, in CHAT the heap space belonging to each C_i is not released earlier than the heap space allocated between G and the choice point immediately preceding G. So, it is correctly segment preserving, to deal during copying with all consumers C_i at the same time as dealing with G. The order in which this happens is not important.

Since the generator up to which a consumer has its state CHAT-protected changes in time (e.g., because of CHAT's laziness and incrementality in performing this operation; see [DS00]), a consumer's state can belong at different moments in time to different generator segments. This is no problem, as the generator segments always become older ones. This means that if for some reason it is determined that a consumer C will eventually have its state saved up to a particular generator G, it would be correct to deal with the state of

C during garbage collection at the moment one deals with G, even if C does not yet have its state saved up to G at the moment of garbage collection.

It is now clear how to transpose this to SLG-WAM: when during the bottom-up traversal of choice points a generator G is encountered, the set of consumers it can schedule should be treated at the same time as G. This means: for each relevant consumer choice point, treat the entries pointing to the heap and follow the chain of environments, but do not follow the chain of choice points between the consumer and G. The CHAT reasoning shows that the latter is correct.

10 Performance evaluation

After we implemented the sliding collector for XSB, we also implemented a copying collector reusing the marking phase that is common for both collectors and following [BL94]. As explained in Section 3.1, Prolog systems have usually opted for a sliding collector. Besides XSB, only a few other Prolog systems have more than one garbage collector: [BL94] reports that Reform Prolog also had a copying and a sliding collector. BinProlog gave up its sliding collector in favor of (segment order preserving) copying. Touati and Hama report on a partially copying collector (for the most recent segment only) that is combined with a sliding collector: see [TH88] for more details. In addition, XSB is currently the only system that has tabling implemented at the engine level. So, for 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 [San91] and the results carry over to Prolog. From an implementation point of view, 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.

10.1 Performance in programs without tabling

Since the relative merits of copying and sliding have been discussed in the literature at length we felt that there was no good reason to test extensively programs without tabling. We just present one set of benchmarks mainly to show that our garbage collectors perform decently. The test programs are shown below; they each build two data structures of the same type which are interleaved on the heap.

```
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 four different sorts of data structures are described in Table 1 by the facts for `longds/2` and `shortds/2`. Note that length of the second data structure is 1/10 the length of the first one. The two queries, `q1` and `q2`, represent the following two situations respectively: either most of the data in the heap survives garbage collection, or only a small fraction (here about 10%) does.

left list	`longds([[[[[[[[[[R|10]|9]|8]|7]|6]|5]|4]|3]|2]|1],R)` `shortds([R|1],R)`
right list	`longds([1,2,3,4,5,6,7,8,9,10|R],R)` `shortds([1|R],R)`
left f/2	`longds(f(f(f(f(f(f(f(f(f(f(R,1),2),3),4),5),6),7),8),9),10),R)` `shortds(f(R,1),R)`
right f/2	`longds(f(1,f(2,f(3,f(4,f(5,f(6,f(7,f(8,f(9,f(10,R)))))))))),R)` `shortds(f(1,R),R)`

Table 1: Construction of the datastructures.

The Prolog systems used in this performance comparison were started with enough initial heap so that no garbage 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 described in this article; in ProLog_by_BIM 4.1[6] and SICStus 3.7 using sliding collectors; in BinProlog 6.84 with its segment order preserving copying collector. The number `15000` in the program 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 [ACHS88]; in XSB and BIM by a self-managed stack.

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

Table 2: Performance comparison of Prolog garbage collectors using different heap data sets.

The figures in Table 2 represent the timings for two queries with different garbage collection rates for four different data structures: two of them are essentially lists while two others are structures constructed with the functor `f/2`; two are recursive to the left, and two 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* [TN94], a technique which reduces every right recursive binary constructor to the size of the optimized list representation;[7] 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.

[6] Now named MasterProlog.

[7] Term compression has a similar effect on constructors with higher arity as well

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

- The performance of the XSB garbage collectors is competitive with that of garbage collectors in commercially available systems.
- All collectors perform much better on high percentage of garbage than on low; this is as it should be.
- The XSB and SICStus Prolog garbage collectors are rather insensitive to the left/right issue, while the BIM and BinProlog ones are more sensitive. For both systems, the marking schema is the reason, while for BinProlog, the term compression asymmetry is a reason as well.
- On the used benchmark programs, 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. The columns under `left f/2` are most informative here, because with this datastructure, the heap layout in all systems is most similar. Of course, the issue of copying versus sliding garbage collection for Prolog (with tabling) can not be decided only on the basis of the above deterministic programs.

10.2 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 [DS00] 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. Besides using tabling, other characteristics of these programs are that they contain non-deterministic predicates and that the test finds all solutions through backtracking. In Table 3, we indicate for each benchmark program 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 execution time without garbage collection is given in the last row of the table. All times are again in milliseconds but now 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
time (no GC)	219	400	130	151	121	400	100	560

Table 3: Performance of sliding and copying garbage collection on a set of benchmarks with tabling.

The fact that our copying collector is not segment order preserving, makes the interpretation of the results of this set of tests not always clear cut. This is because, with the sliding collector, more memory is cheaply reclaimed by backtracking. However, the following observations and conclusions can be made:

- In all cases the sliding collector gets invoked less frequently than the (non segment order preserving) copying collector and collects less garbage. This is caused by the loss of heap reclamation on backtracking for the copying collector.

- In some cases the sliding collector can spend less time than the copying collector. The reason for this last behavior seems to 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 all tested programs.

- The copying collector is handicapped in two ways by the bechmark set: First, most of the data remains useful; this can be deduced from the low figures of garbage that is collected. Secondly, in programs where a considerable amount of backtracking is involved, a copying collector can be called arbitrarily more often than a sliding one, when given the same space for its two semi-spaces together, as the sliding collector uses for the heap. A generational garbage collection schema [LH83] can in all cases improve the performance figures.

11 Measuring fragmentation and effectiveness of early reset

Prolog implementors have paid little attention to the concept of external fragmentation: as far as we know, measurements of external 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., [JW98]. 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 [Sch90] being a notable exception. We will combine both measurements. It is important to realise that the results about fragmentation tell something about the memory allocator, not about the garbage collector!

According to [JW98], external fragmentation can be expressed in several ways. To us, the most informative seems the one that expresses how much memory is wasted: a piece of memory is wasted by an allocator, if a better allocator can avoid allocating it at all. A perfect (with respect to the implemented usefulness logic) allocator is one that wastes no memory. It can be approximated by performing a compacting garbage collection at every attempt to allocate a new piece of memory. The relative difference between the actual amount allocated by an imperfect allocator and the amount allocated by the perfect allocator is the external fragmentation we will report later on.

In order to measure the amount of memory allocated by a particular allocator, we disable garbage collection and keep track of the high water mark for the heap during the execution of a program. This quantity is denoted *heapsize_without_collection*.

During the same run, the marking phase of the collector is run regularly, so that at the end of the run, the maximum number of marked cells gives a decent approximation of the amount of memory a perfect allocator would have needed. This quantity is reported as *minimal_heap*. With less heap space, the program cannot run. With exactly this heap space, the program might trigger garbage collection at every predicate call.

The fragmentation then becomes:

$$\frac{heapsize\_without\_collection - minimal\_heap}{heapsize\_without\_collection}$$

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 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 4 and 5 with and without performing early reset. Additionally, the tables contain the average 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	reduce	smplanal	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 4: 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 figures would probably show an even higher fragmentation if XSB trimmed its environments or had more sophisticated determinism detection. This is because both environment trimming (i.e., disregarding from the environment variables which are not live) and fewer choice points make the marking phase more precise, which in turn results in collecting more garbage.

fragmentation	cs_o	cs_r	disj_o	gabriel	kalah_o	peep	pg	read_o	tboyer
w early reset	52.3	51.7	63.4	62.9	83.6	67.3	73.8	53.9	0.44
w/o 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
pred calls (K)	72	138	40	47	45	134	34	169	6.6
max heap (K)	1.1	1.2	0.8	2.1	4.8	28	12	43	67

Table 5: Fragmentation in a set of programs with tabling with and without early reset.

Similarly, 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 depend very much 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 [Sch90] 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.

The fragmentation for tboyer is extremely small when compared to the higher fragmentation for boyer. Closer inspection of tboyer reveals that it benefits a lot from the effect mentioned as point 2 in Section 4. What happens is that in this benchmark are repeated computations which require a lot of heap space and generate garbage, but when tabling is used many of them are avoided.

12 Concluding remarks

In this article, we presented both the theoretical understanding of memory management principles and their actual implementation. On the one hand, we discussed the memory organization and usefulness logic of logic programming systems with tabling and issues that have to be resolved for their effective and efficient garbage collection. On the other, we addressed the practical aspects of heap garbage collection in the WAM with or without tabling, and reported our experience and the main lessons learned from implementing two garbage collectors in XSB. We hold that by making implementation choices concrete and documented—even those that are considered folklore—this article 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 uses a fixed allocation schema for data structures (allocation on the top of the heap), about which there is relatively little empirical knowledge in the context of logic programming: the figures we presented in Section 11 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 trade some efficiency for better memory management and some Prolog implementations have even lived for quite a while without garbage collection at all. In all, it is fair to say that 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 topic of fragmentation about which there seem no figures available in literature, there is no published 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.

Acknowledgements

We are grateful to several people from the XSB implementation team for helping us understand their code; in particular, we want to thank David S. Warren, Terrance Swift, and Prasad Rao. Some good suggestions from an

anonymous reviewer helped in improving the presentation.

References

[ACHS88] Karen Appleby, Mats Carlsson, Seif Haridi, and Dan Sahlin. Garbage collection for Prolog based on WAM. *Communications of the ACM*, 31(6):719–741, June 1988.

[AK91] Hassan Aït-Kaci. *Warren's Abstract Machine: A Tutorial Reconstruction.* The MIT Press, Cambridge, Massachusetts, 1991. See also: http://www.isg.sfu.ca/~hak/documents/wam.html.

[BCRU86] Yves Bekkers, Bernard Canet, Olivier Ridoux, and Lucien Ungaro. MALI: A memory with a real-time garbage collector for implementing logic programming languages. In *Proceedings of the 1986 Symposium on Logic Programming*, pages 258–264. IEEE Computer Society Press, September 1986.

[Bir80] Richard S. Bird. Tabulation techniques for recursive programs. *ACM Computing Surveys*, 12(4):403–417, December 1980.

[BL71] P. Branquart and J. Lewi. A scheme of storage allocation and garbage collection for Algol-68. In J. E. L. Peck, editor, *Algol-68 Implementation*, pages 198–238. North-Holland, Amsterdam, 1971.

[BL94] Johan Bevemyr and Thomas Lindgren. A simple and efficient copying garbage collector for Prolog. In Manuel Hermenegildo and Jaan Penjam, editors, *Proceedings of the Sixth International Symposium on Programming Language Implementation and Logic Programming*, number 844 in LNCS, pages 88–101. Springer-Verlag, September 1994.

[Boe00] Hans-Juergen Boehm. Reducing garbage collector cache misses. In *Proceedings of ISMM'2000: ACM SIGPLAN International Symposium on Memory Management*, pages 59–64. ACM Press, October 2000.

[BRU92] Yves Bekkers, Olivier Ridoux, and Lucien Ungaro. Dynamic memory management for sequential logic programming languages.

In Yves Bekkers and Jaques Cohen, editors, *Proceedings of IWMM'92: International Workshop on Memory Management*, number 637 in LNCS, pages 82–102. Springer-Verlag, September 1992.

[Car90] Mats Carlsson. *Design and Implementation of an Or-Parallel Prolog Engine*. PhD thesis, Department of Telecommunication and Computer Systems, The Royal Institute of Technology (KTH), Stokholm, Sweden, March 1990.

[CDD+98] Baoqiu Cui, Yifei Dong, Xiaoqun Du, K. Narayan Kumar, C. R. Ramakrishnan, I. V. Ramakrishnan, Scott A. Smolka, and David S. Warren. Logic programming and model checking. In Catuscia Palamidessi, Hugh Glaser, and Karl Meinke, editors, *Principles of Declarative Programming, 10th International Symposium, PLILP'98*, number 1490 in LNCS, pages 1–20. Springer, September 1998.

[CDS98] Michael Codish, Bart Demoen, and Konstantinos Sagonas. Semantics-based program analysis for logic-based languages using XSB. *Springer International Journal of Software Tools for Technology Transfer*, 2(1):29–45, November 1998.

[Che70] C. J. Cheney. A nonrecursive list compacting algorithm. *Communications of the ACM*, 13(11):677–678, November 1970.

[CMES00] Yoo C. Chung, Soo-Mook Moon, Kemal Ebcioglu, and Dan Sahlin. Reducing sweep time for a nearly empty heap. In *Conference Record of the* 27^{th} *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 378–389. ACM Press, January 2000.

[DET96] Bart Demoen, Geert Engels, and Paul Tarau. Segment order preserving copying garbage collection for WAM based Prolog. In *Proceedings of the 1996 ACM Symposium on Applied Computing*, pages 380–386. ACM Press, February 1996.

[DS99] Bart Demoen and Konstantinos Sagonas. CAT: the Copying Approach to Tabling. *Journal of Functional and Logic Programming*, November 1999. Special issue on selected papers from PLILP/ALP'98.

[DS00] Bart Demoen and Konstantinos Sagonas. CHAT: the Copy-Hybrid Approach to Tabling. *Future Generation Computer Systems*, 16(7):809–830, May 2000.

[ISO95] Information technology - Programming languages - Prolog - Part 1: General Core. ISO/IEC 13211-1, 1995. See also http://www.logic-programming.org/prolog_std.html.

[JL96] Richard Jones and Rafael Lins. *Garbage Collection: Algorithms for automatic memory management*. John Wiley, 1996. See also http://www.cs.ukc.ac.uk/people/staff/rej/gcbook/gcbook.html.

[JW98] Mark S. Johnstone and Paul R. Wilson. The memory fragmentation problem: Solved? In *Proceedings of ISMM'98: ACM SIGPLAN International Symposium on Memory Management*, pages 26–36. ACM Press, October 1998.

[LH83] Henry Lieberman and Carl Hewitt. A real-time garbage collector based on the lifetimes of objects. *Communications of the ACM*, 26(8):419–429, June 1983.

[LH90] Serge Le Huitouze. A new data structure for implementing extensions to Prolog. In Pierre Deransart and Jan Maluszynski, editors, *Programming Language Implementation and Logic Programming, 2nd International Workshop PLILP'90*, number 456 in LNCS, pages 136–150. Springer-Verlag, August 1990.

[Mor78] F. Lockwood Morris. A time- and space-efficient garbage compaction algorithm. *Communications of the ACM*, 21(8):662–665, August 1978.

[Neu90] Ulrich Neumerkel. Extensible unification by metastructures. In Maurice Bruynooghe, editor, *Proceedings of the Second Workshop on Meta-programming in Logic*, pages 352–363, April 1990.

[PBW85] Edwin Pittomvils, Maurice Bruynooghe, and Yves D. Willems. Towards a real time garbage collector for Prolog. In *Proceedings of the 1985 Symposium on Logic Programming*, pages 185–198. IEEE Computer Society Press, July 1985.

[RRS+99] I. V. Ramakrishnan, Prasad Rao, Konstantinos Sagonas, Terrance Swift, and David S. Warren. Efficient access mechanisms for tabled logic programs. *Journal of Logic Programming*, 38(1):31–54, January 1999.

[San91] Patrick M. Sansom. Combining copying and compacting garbage collection or Dual-mode garbage collection. In Rogardt Heldal, Carsten Kehler Holst, and Philip Wadler, editors, *Functional Programming*, Workshops in Computing. Springer-Verlag, August 1991.

[SC91] Dan Sahlin and Mats Carlsson. Variable shunting for the WAM. Technical Report SICS/R-91/9107, SICS, 1991.

[Sch90] Joachim Schimpf. Garbage collection for Prolog based on twin cells. In *Proceedings of the 1990 Implementation Workshop (held in conjunction with NACLP)*, pages 16–25, Austin, Texas, November 1990.

[SS98] Konstantinos Sagonas and Terrance 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.

[SSW94] Konstantinos Sagonas, Terrance Swift, and David S. Warren. XSB as an efficient deductive database engine. In *Proceedings of the ACM SIGMOD International Conference on the Management of Data*, pages 442–453. ACM Press, May 1994.

[SSW96] Konstantinos Sagonas, Terrance Swift, and David S. Warren. An abstract machine for computing the well-founded semantics. In Michael Maher, editor, *Proceedings of the Joint International Conference and Symposium on Logic Programming*, pages 274–288. The MIT Press, September 1996.

[Tar00] David Tarditi. Compact garbage collection tables. In *Proceedings of ISMM'2000: ACM SIGPLAN International Symposium on Memory Management*, pages 50–58. ACM Press, October 2000.

[TH88] Hervé Touati and Toshiyuki Hama. A light-weight Prolog garbage collector. In *Proceedings of the International Conference on*

Fifth Generation Computer Systems (FGCS'88), pages 922–930. OHMSHA Ltd. Tokyo and Springer-Verlag, November/December 1988.

[TN94] Paul Tarau and Ulrich Neumerkel. A novel term compression scheme and data representation in the BinWAM. In Manuel Hermenegildo and Jaan Penjam, editors, *Proceedings of the Sixth International Symposium on Programming Language Implementation and Logic Programming*, number 844 in LNCS, pages 73–87. Springer-Verlag, September 1994.

[War83] David H. D. Warren. An abstract Prolog instruction set. Technical Report 309, SRI International, Menlo Park, U.S.A., October 1983.

[YK00] Guizhen Yang and Michael Kifer. FLORA: Implementing an efficient DOOD system using a tabling logic engine. In John Lloyd, Veronica Dahl, Ulrich Furbach, Manfred Kerber, Kung-Kiu Lau, Catuscia Palamidessi, Luís Moniz Pereira, Yehoshua Sagiv, and Peter J. Stuckey, editors, *Proceedings of Computational Logic — CL-2000*, number 1861 in LNAI, pages 1078–1093. Springer, July 2000.

[Zho00] Neng-Fa Zhou. Garbage collection in B-Prolog. In *Proceedings of the First Workshop on Memory Management in Logic Programming Implementations*, July 2000. Co-located with CL'2000. See http://www.cs.kuleuven.ac.be/~bmd/mmws.html.

[Zor93] Benjamin Zorn. The measured cost of conservative garbage collection. *Software Practice and Experience*, 23(7):733–756, July 1993.

[ZSYY00] Neng-Fa Zhou, Yi-Dong Shen, Li-Yan Yuan, and Jia-Huai You. Implementation of a linear tabling mechanism. In Enrico Pontelli and Vítor Santos Costa, editors, *Practical Aspects of Declarative Languages: Second International Workshop*, number 1753 in LNCS, pages 109–123. Springer, January 2000.