# Memory Management for Prolog with Tabling

Bart Demoen Konstantinos Sagonas

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

**Abstract**

Tabling can be implemented in a (WAM-based) Prolog system by means of SLG-WAM: consumers suspend and their state is preserved by freezing the execution stacks. XSB is a system that currently implements tabling based on the SLG-WAM. The memory model is quite complex and attempts to understand the notion of usefulness of data in XSB well enough to build a precise garbage collector have failed in the past. CAT is a recent alternative to SLG-WAM: it suspends consumers by copying parts of the execution stacks. The memory model is simpler and the design of a more precise garbage collector became feasible. CAT also provided the necessary insights in the usefulness of data in the context of the SLG-WAM. This paper describes the memory management of tabled logic programming systems, whether based on the SLG-WAM or on CAT. Since CAT can perform arbitrarily worse than SLG-WAM space-wise, also a minor garbage collection on creation of the CAT areas is described and its effectiveness is discussed.

## 1 Introduction

Incorporation of tabling (also known as memoization) in the execution model of 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 repeated subcomputations are avoided and 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 [16], have been proven useful to a wide range of application areas such as parsing, databases [16], program analysis [7, 8], and recently verification through model checking [14]. At the same time, tabling introduces some extra complications in the standard implementation platform for Prolog, the WAM [17]. Most of the complications are 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. The SLG-WAM [15], the abstract machine of XSB, preserves consumer states 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.

Freezing is not the only possible way to implement the control of tabled execution. In [10], we introduced the 'Copying Approach to Tabling' abbreviated CAT. In short, CAT preserves and reinstalls the execution environments of suspended computations through *copying*. CAT resembles other copy-based techniques, notably MUSE [2] and the implementation of backtracking in C described in [11]. Introducing backtracking into a language (like Prolog) requires an extra complication in the stacks: indeed, the WAM local 'stack' which keeps information about the environments of clauses is really a cactus stack (i.e. some environments which are not in the current branch of the computation are needed on backtracking) and [11] maps this to a regular (C) stack by copying. Similarly, OR-parallelism on top of Prolog requires an even more complex stack structure than Prolog, and MUSE reduces this back to the Prolog stacks by using copying to separate the areas of different workers. Also the implementation of tabling in SLG-WAM often turns the stacks of the underlying Prolog engine into *spaghetti* stacks since branches of the search tree advance their computation in asynchronous, interleaved ways. CAT is an attempt to reduce the complexity of the SLG-WAM stacks to the relatively well understood stacks of the WAM: indeed, the stack structure of SLG-WAM is difficult to understand and in particular the implications on memory management are very hard to grasp. The simplification that CAT offers also pays off here: reasoning about reachability and usefulness is much easier than in the freezing implementation of SLG-WAM.

Independently of the implementation model that is chosen for the suspension/resumption mechanism, tabling systems have inherently more complex memory models and in general their space requirements are bigger than those of plain Prolog systems. In short, tabling calls for more sophisticated memory management. Despite this, issues involved in the memory management of tabled logic programming systems are currently unexplored in the literature, and this

Appears in *Proceedings of the 1998 ACM SIGPLAN International Symposium on Memory Management*, Vancouver, B.C., Canada, Oct. 1998. ACM Press.

paper fills this gap by providing a relatively complete account. We discuss the *usefulness logic* of Prolog with tabling and based on this we describe garbage collection schemes for both SLG-WAM- and CAT-based implementations. We strongly believe that the issues discussed in this paper more or less carry over to most other possible WAM-based implementations of tabling. On the other hand, this paper does not address issues related to memory management of the table space; this is an orthogonal issue.

In Section 2 we state the concepts and notations we use in the sequel of the paper and review the two approaches to implementing tabling: the SLG-WAM and the CAT. The description of these two approaches is relatively self-contained, however, its focus is on issues related to memory management and as such the description is not exhaustive; the reader is therefore referred to [15, 10] for full details of the two implementation schemes. Section 3 explains cooperation between CAT and heap garbage collection, with a detailed discussion on early reset. We pay special attention to how the understanding of garbage collection for CAT, can be used for designing a garbage collector for the SLG-WAM (or indeed for any WAM-based) implementation model. Section 4 on the other hand is specific to the CAT model and shows further space improvements during copying for the heap and the local stack. We end with a performance evaluation of this selective copying and some concluding remarks.

We assume the reader to be familiar with Prolog and the WAM (see [1] for a general introduction) and to some extent with tabled execution of logic programs. Also knowledge of garbage collection techniques for Prolog will help; see [5, 9] for specific instances of heap garbage collection for Prolog and [4, 3] for a more general introduction.

## 2 Concepts and Terminology of Tabling and its Implementation

Tabled resolution records calls to designated *tabled* predicates and their answers in a persistent global data structure called a *table*. All other predicates are executed as in Prolog. When a call, $s$, to a tabled predicate is encountered, a check is made to determine whether $s$ already exists in the table or is new to the evaluation. If $s$ is new, $s$ is called a *generator*, it will be entered in the table and execution will continue as in Prolog by resolving $s$ against the program clauses of the predicate. Let $\theta$ denote the accumulated bindings for variables in $s$ upon successful return from a program clause of $s$. Then $s\theta$ constitutes an *answer* for $s$. If this answer is new to the evaluation then it is also inserted in the table and is consumed by $s$; otherwise the evaluation fails (a vital step to ensure termination). If, on the other hand, $s$ already appears in the table, it will not use the program clauses as done in usual Prolog execution, but will resolve against the answers that currently exist or will be derived for the corresponding generator. In this case, $s$ is called a *consumer* subgoal.

Tabling cannot be implemented using the purely depth-first search control of the WAM as the generation and consumption of answers are asynchronous and quite often mutually dependent events. This calls for a mechanism to *suspend* and *resume* computations of consumers: for the purposes of this paper it is sufficient to assume that a consumer is suspended when it has consumed all answers from the table, and is possibly resumed upon determining that it has unresolved answers to consume. In implementation terms, the need for a suspension/resumption mechanism means that an abstract machine for tabled execution has to alter the control — and thus the memory management — of the WAM to either preserve or be able to reconstruct execution environments of consumers until these do not need to be resumed anymore. In definite programs, this condition happens when generators have exhausted all their program clauses and all consumers have consumed all answers that generators have produced and inserted in the tables: we then say that these subgoals are *complete*[1].

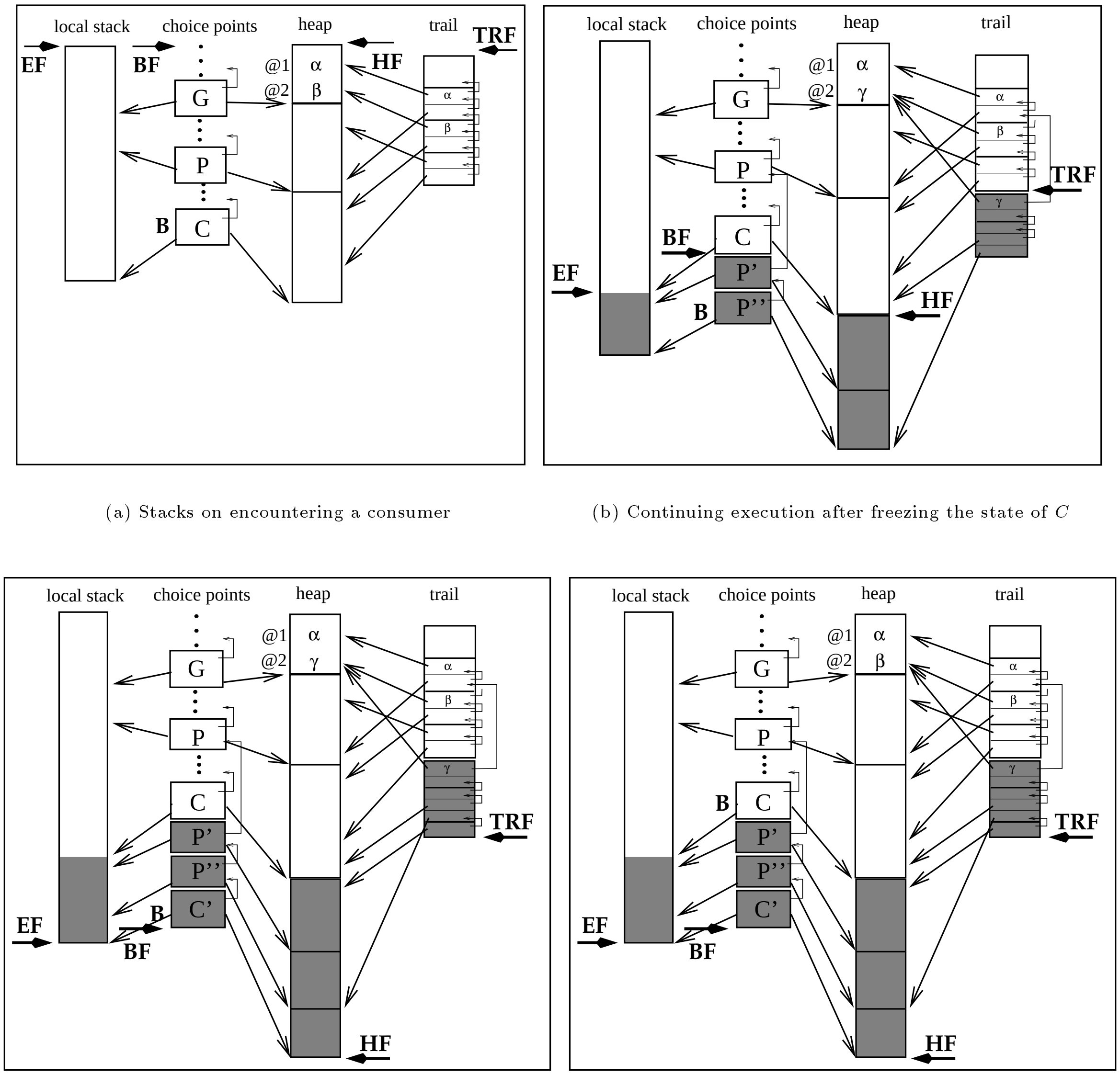
As Prolog execution is supported as well, the starting point for an implementation of tabling is the WAM, but we assume a WAM with four stacks: trail, choice points, local stack (the environments) and the heap. Their top of stacks are pointed by WAM registers denoted respectively as **TR**, **B**, **A** and **H** when we need them. **E** points to the most recent environment (the fact that we need both **E** and **A** in WAM is a bit unfortunate for the description). We denote by **B**[E] the environment field in the choice point pointed by **B** and likewise for the other fields of choice points or other WAM data structures. In our description as well as in the figures, the stacks will grow downwards, i.e. higher in the stack means older, lower means younger.

### 2.1 SLG-WAM: The environment-sharing implementation of tabling

Tabling can be implemented by modifying the WAM as is done by the SLG-WAM [15]. In this approach, execution environments of suspended consumers are preserved by *freezing* the WAM stacks, i.e. by not allowing backtracking to reclaim space in the stacks as is done in the WAM. In implementation terms, this means that the SLG-WAM adds an extra set of *freeze registers* to the WAM, one for each stack and allocation of new information occurs below the frozen part of the stack. Suspension of a consumer is performed in the SLG-WAM by creating a consumer choice point to backtrack through the answers in the table, setting the freeze registers to point to the current top of the stacks, and upon exhausting all answers fall back to the previous choice point by failing as in the WAM (i.e. undoing the variable bindings and restoring the WAM registers) but without reclaiming any space. Frozen space is reclaimed *only* upon determining completion. Note that a side-effect of having frozen segments in the stacks is that the stacks actually represent trees: for example, contrary to the WAM, choice points on the same branch of the computation may not be contiguous and the previous choice point may be arbitrarily higher in the stack.

Memory areas of the SLG-WAM and their relationships are depicted in Figure 1. Initially all SLG-WAM freeze registers point to the beginning of the stacks; they are shown by arrows next to each stack. After executing some Prolog code the execution encounters a generator $G$ and creates a generator choice point for it. The execution continues, some more choice points are created and eventually a consumer $C$ is encountered. The SLG-WAM stacks at this point are shown in Figure 1(a). The heap and the trail are shown segmented by choice points; the same segmentation is not shown for the local stack as it is a cactus stack. From the

[1] Note that, contrary to what happens in the memoization of functional languages, non-determinism introduces an extra complication in the picture as it is usually not known in advance how many answers a given subgoal will produce and until when its consumers might need to be resumed and their execution environments be preserved.




(a) Stacks on encountering a consumer

(b) Continuing execution after freezing the state of $C$

(c) Stacks upon freezing another consumer, $C'$

(d) Stacks upon resuming a frozen consumer $C$

Figure 1: Memory areas while executing under an SLG-WAM-based implementation.

trail, some pointers point to cells older than the generator $G$: these cells have addresses @1 and @2 in the picture, and the values of the cells are $\alpha$ and $\beta$. One can see that a trail entry in this picture consists of two pointers and a value, while in WAM, a trail entry is just one pointer: this new trail structure will be explained later on. On encountering $C$ the stacks are frozen by setting the freeze registers to point to the current top of the stack (cf. Figure 1(b)). After possibly returning some answers to $C$, the execution fails out of $\mathbf{B}_C$, and suppose that the youngest choice point with unexplored alternatives is $\mathbf{B}_P$. As shown in Figure 1(b), allocation of new information (shown in a darker shade) takes place below the freeze registers and no memory above the freeze registers is reclaimed. Notice the conceptual tree form of e.g. the choice point stack as shown by previous pointers from choice points. Furthermore, note that by continuing with some other part of the computation, some cells may change value: e.g. cell @2 from $\beta$ to $\gamma$. Finally, by continuing execution, more consumers can be encountered and in this case further freezing of the stacks will take place causing the freeze resisters to move further down in the stacks; see Figure 1(c).

As expected, to resume a suspended computation of a consumer, the SLG-WAM needs to have a mechanism to reconstitute its execution environment. Besides resetting the WAM registers (e.g. setting $\mathbf{B}$ to point to the consumer choice point $C$ as in Figure 1(d)), the variable bindings at the time of suspension have to be restored. This can be done using what is known as a *forward trail* [18]. As fully described in [15], each entry in the forward trail consists of three cells: a reference cell, a value cell, and a pointer to the previous trail entry, as opposed to the WAM trail which consists of only a reference cell. These entries are shown in e.g. Figures 1(a) and 1(b): entries for @1 and @2 record the values $\alpha, \beta$, and $\gamma$ (the value cell for other trail entries is not explicitly indicated) and because of the previous pointers the trail is also tree-structured. Given this trail, restoring the execution environment $EE$ from a current execution environment $EE_c$, is a matter of untrailing from $EE_c$ to a common ancestor of $EE_c$ and $EE$, and then using values in the forward trail to reconstitute the environment of $EE$. Note that after restoring the execution environment of $C$ the heap value at address @2 switches back to its value, $\beta$, that it contained upon creation of the consumer $C$. Finally, note that the freeze register of a stack can be above (Figure 1(b)), equal to (Figure 1(c)), or below (Figure 1(d)) the corresponding WAM register of the stack.

### 2.2 CAT: The environment-copying implementation of tabling

Instead of maintaining execution environments of suspended consumers through freezing the stacks and using an extended trail to reconstruct them, one could also preserve environments of consumers by copying all the relevant information about them in a separate memory area, let execution proceed as in the WAM, and reinstall these copies whenever the corresponding consumers need to be resumed. This is the main idea behind CAT: the 'Copying Approach to Tabling'. An advantage of this approach is that, contrary to the SLG-WAM, Prolog execution happens as in the WAM: there is no need for a forward trail nor freeze registers and the stacks do not have a tree form. CAT selectively copies information needed to reinstall suspended environments in *CAT areas* as briefly explained below[2].

The CAT area has four memory areas (containing information from each of the four WAM stacks). Figure 2 shows these memory areas in a CAT-based implementation. When execution encounters a consumer, a choice point $C$ is created for it. Let the youngest generator choice point in the stack be $G$ (the dots show possible Prolog choice points that appear in between). A CAT copy is about to be made; the situation is depicted in Figure 2(a). The shaded parts in Figure 2(a) show exactly what CAT copies. From the heap. CAT copies the part between the current top $\mathbf{H}$ and $\mathbf{B}_G$[H]. We refer to the region of the CAT area that contains the copy of the heap as the *CAT heap* and likewise for the other areas. The part of the local stack that needs copying is between $\mathbf{A}$ and $\mathbf{B}_G$[E]. One could think that from the choice point stack, CAT needs to copy from $\mathbf{B}$ till $\mathbf{B}_G$, but [10] argues this is wrong: instead, it is correct to copy only the consumer choice point (cf. also Section 3.2 for an explanation). Copying the trail is more complicated: as we do not save the part of the heap that is older than $\mathbf{B}_G$ and since this part can contain values that were put there during execution more recent than $\mathbf{B}_G$, we need to save together with the trailed addresses also the values these trailed addresses now contain; we do not need a similar double trail for the part of the heap that is more recent than $\mathbf{B}_G$, because we copy that part completely. The copied information is saved in a CAT area which is separate from the stacks (cf. Figure 2(b)) and execution continues as in the WAM by failing out of the consumer choice point. Contrary to what happens in the SLG-WAM, backtracking in CAT now reclaims space. Figure 2(b) shows a possible situation where backtracking has taken place up to a Prolog choice point $P$ which lay between $G$ and $C$ in Figure 2(a), and then an alternative path of the computation was tried (shown in a darker shade). Note that this new computation has resulted in the stacks having different contents than what is saved in the CAT area (although as indicated by the lighter shading some parts are still intact).

Let us assume that, as in the case of SLG-WAM, another consumer $C'$ is encountered and its choice point is immediately below $P''$. Another CAT area will be created at this point and the resulting state of the CAT memory areas is shown in Figure 3(a). Eventually, through backtracking execution will fall back to $G$ and after $G$ exhausts all resolution with program clauses, the evaluation reinstalls consumers with unresolved answers that have copied up to the generator $G$[3]. The resulting stacks (but not the CAT areas as they are the same as those of Figure 3(a)) are shown in Figure 3(b): through copying, the state of one consumer, $C$ in this case, has just been reinstalled below the stacks of $G$ and now $C$ can start consuming its answers from the table. Once $C$ consumes all available answers, a similar reinstallation will take place for $C'$ and all other consumers that the evaluation has possibly encountered. Note that after reinstalling the consumer, the choice point and trail stack are in general smaller than at the time of saving the CAT area. The CAT areas themselves remain in existence until it can be determined that the associated generators are complete. Since each CAT area can be separately allocated and freed, memory management of CAT areas is relatively straightforward and is described in [10].

At any moment, the computation that is going on and its stacks, are referred to as *current*. Space for computations that have been *frozen* (SLG-WAM terminology) or *copied* (CAT terminology) are *inactive*. Note that there is conceptually a one-to-one mapping between frozen consumers and CAT areas: this can also be seen by comparing Figures 1(c) and 3(a).

## 3 Heap Garbage Collection

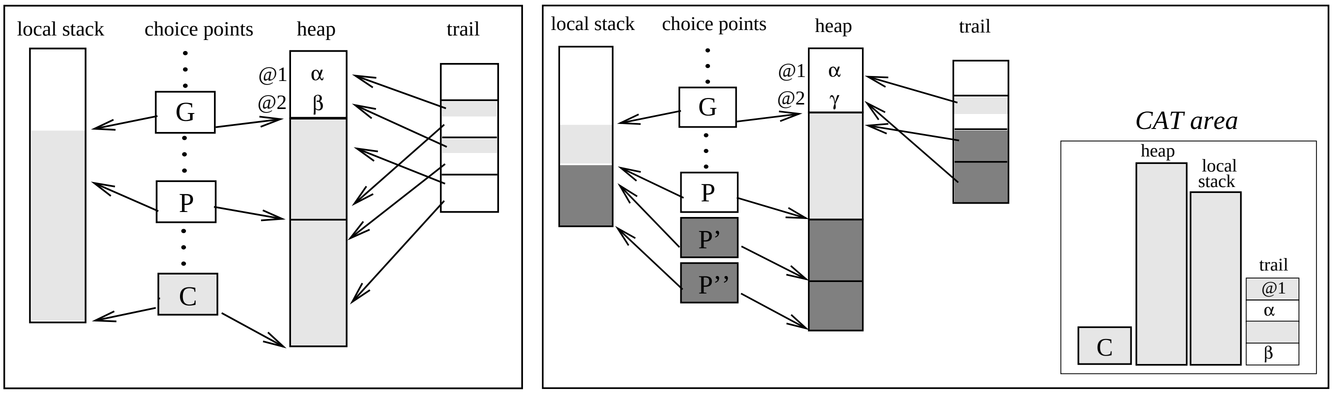
In this section we will concentrate on aspects of heap garbage collection for both CAT and SLG-WAM. The common situation is as follows: the current computation needs to be collected and there are one or more consumers suspended. We will describe the issues for a sliding collector (see [12]) but most of the issues translate easily to a compacting collector (as in [5, 9]) as they are independent of the garbage collection techniques that are being employed (see also [4]). The issues are:

- to find the reachable data
- to move it appropriately while adapting all pointers to it

In both sliding or compacting, a marking phase is performed (see [5] for why this is necessary for compacting). The root set in WAM consists of the argument registers, the saved argument registers in the choice points, the local variables in the environments reachable from the current top

[2] Actually, it does so in a more incremental way, but as this is not relevant for this paper we refer those interested to [10] for more details.

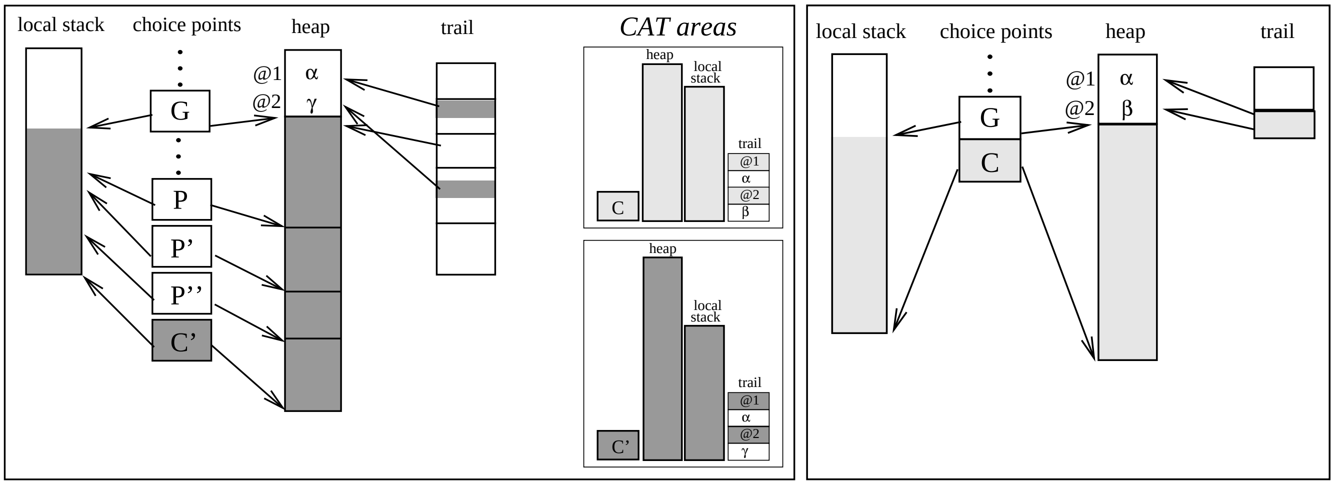
[3] This may depend on the scheduling strategy that is used; the strategy which governs when answers are returned to consumers — see [10].




(a) Stacks just before making a CAT copy; shaded parts show what is to be copied

(b) Stacks and one CAT area after making a CAT copy (for C) and continuing computation

Figure 2: Memory areas while executing under a CAT-based implementation.




(a) CAT memory areas immediately after making another CAT copy now for consumer C’; shaded parts show what was copied

(b) Stacks upon reinstalling the CAT area for consumer C; CAT areas are not shown

Figure 3: Memory areas while executing under a CAT-based implementation (Figure 2 continued).

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. It ensures the optimal possibility for doing virtual backtracking or early reset (see [4]).

[4] A backtracking state is the continuation of a computation which results from backtracking, possibly more than once.

### 3.1 Heap garbage collection for CAT

In CAT, it is possible to collect just the current computation; i.e. perform garbage collection only in the stacks and leave the CAT areas of consumers unchanged (up to relocation of pointers as explained below). Since it is the current computation that overflows, this looks the natural thing to do, especially since most often the current computation and the saved consumer are unrelated; this means that the collection of the consumer state might never be needed. A drawback is that potentially, the consumer could after its reinstallation cause very quickly a new garbage collection that might have been avoided by collecting also the CAT areas. We deal with CAT areas — and thus with this issue — in Section 4 where we describe a scheme for saving in CAT areas only data that is reachable by the consumer.

Assuming the existence of this scheme, the need for garbage collection in CAT areas disappears.

Consumers share a part of the current computation: the part that is older than the generator up to which they have copied information. The CAT area of a consumer can have the following pointers to the current computation:

1. from the CAT heap to the shared heap part
2. from the CAT trail to the shared heap part
3. from the CAT local stack to the shared local stack and shared heap part

These pointers are to be followed for marking and take part in the usual relocation during the second phase of the sliding algorithm.

Moreover, the CAT area contains in general also pointers to the heap and local stack as they will be after reinstalling the consumer: conceptually, such a pointer references the CAT heap (or CAT local stack) but in reality it points to the active heap or local stack. Such a pointer we name a *CAT internal* pointer. During the marking phase, it is followed as if pointing to the CAT area itself. Since the exact place where the consumer will be reinstalled will change because of the garbage collection, the CAT internal heap pointers need to be relocated by the same amount by which the heap pointer from its generator has changed. Figure 4 illustrates the situation.

$G$ denotes the generator choice point up to which a consumer has been saved: from this consumer, only the CAT heap is shown. The top of heap at the moment of creation of $G$ is $h$. The CAT heap contains references $a$ and $b$ to the old part of the current heap, while the references $p$ and $q$ are CAT internal heap pointers: they do point to the heap, but the data they point is currently in the CAT heap and will be in the heap only after the consumer will been reinstalled. Consequently, the references $a$ and $b$ are relocated according to the sliding process taking place in the old part of the heap, while the pointers $p$ and $q$ are shifted by the same amount as the $h$ pointer.

Since, as shown in Figure 3(a), there may exist several CAT areas at the same time (for different consumers), it is worthwhile noting that no references from one consumer to another consumer can exist: references are always to the heap and local stack in which the current computation evolves.

The above can be adapted to a copying collector: in a schema that requires marking before copying (see [5, 9]), the marking phase remains the same. Then copying [6] takes place of the reachable current heap, after which the forwarding pointers contain the information to relocate the pointers from the CAT area to the current heap. The relocation of the CAT internal heap pointers remains as explained above.

In this schema, the CAT areas are not collected themselves; one can argue that a consumer could provoke very quickly after its reinstallation a new collection. This is obviously true. So the schema has to be made incremental, in the sense that if the part of the computation older than the generator has been collected already, it should not be collected after its consumer is reinstalled and causes overflow; only the part that corresponds to the installed consumer should be collected. For more details on incremental garbage collection for Prolog, see for instance [5]. In this way, the collection of the consumer is postponed until it is really needed and it might never be !

### 3.2 Heap garbage collection for SLG-WAM

In the SLG-WAM, it seems difficult to collect the current computation without collecting the frozen consumers at the same time: their data are intertwined on the stacks. So, the proposal here is to use as root set the current computation (and its backtracking states) as well as the frozen 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. In the past, the only way we could imagine marking a consumer, was by melting it first (i.e. reinstall its bindings by using the 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), will 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 threads of execution. In these views, the cell has different values and in one view, *usefulness logic* [4] dictates that marking is not recursive if the envelope of the cell was not marked before. Figure 5 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 bigger box has not been marked before and the object `B` is dead according to the usefulness logic. However, marking from view 1 first, sets the mark bit on, so that when later marking from view 2 happens, `B` will be considered alive.

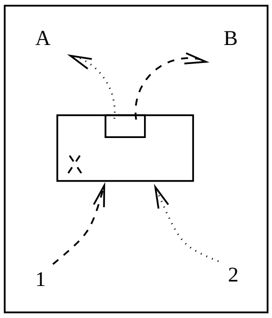



Figure 5: A cell reachable through two different paths.

This is exactly the situation in tabling, where different consumers and the current computation correspond to different threads of execution, each having their view on the contents and liveness of reachable cells.

In the setting of CAT, it was easier to come 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 SLG-WAM, two consumers can also share a part of the trail and local stack, so we expect that it pays off to avoid marking from the same roots (in this shared part) more than once. 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.

### 3.3 Virtual backtracking and the order of marking

Virtual backtracking, or early reset, in the context of tabled logic programming remained long a mystery to us. For a

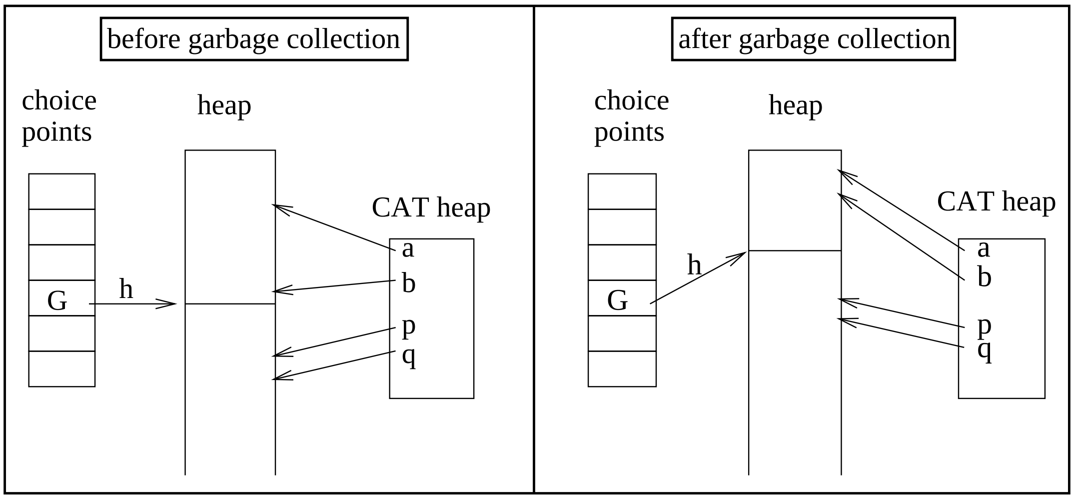



Figure 4: Some references from the CAT heap before and after garbage collection.

good account on this issue in Prolog, see [4], or [3] and [13]. The idea of virtual backtracking is that a trailed heap entry which is not reachable for the forward continuation of the current computation (but might be for alternative branches of the current computation, i.e. on backtracking) can be set to unbound during garbage collection and the trail entry itself can be discarded as well. The situation is recognized during marking, and it is essential that the continuation is marked before the future alteratives.

The execution of `?- main.` against the following artificial piece of Prolog code shows a (well-known in the literature) potential for early reset:

```
main :- h(f(X)).
h(Z) :- Z = f([1,2,3]),
        last_use_of(Z),
        gc, % do garbage collection
        more_goals.
h(Z) :- ...
```

The binding of the variable `X` (in `main`) to the list `[1,2,3]` is on the trail, but since `Z` is not useful in forward execution after garbage collection, it can be discarded during garbage collection, even though the data structure in `Z` is reachable from the choice point.

For CAT, it is clear that during garbage collection, early reset can be performed for the current computation: the suspended computations, i.e. the consumers, have their own computation state saved. If this applies to CAT, it has to apply to SLG-WAM as well, as the matter of usefulness of data is related to the abstract suspension mechanism rather than to the actual realization. However, we also want to answer the question whether during marking of the consumers early reset is allowed and/or possible.

A CAT trail entry of a consumer, say $C$, can contain a reference to a heap entry that is not reachable from the continuation of $C$: there is no harm in removing this CAT 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 was done first, the mark bit of this trailed heap entry was set and the marking phase of the consumer can 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 the generator first, the consumer looses an opportunity for early reset — or at least CAT trail compaction. This means that the order in which consumers and the current computation are marked, can be crucial. For the sake of focusing the ideas, assume that at the moment of the garbage collection, there is just the current computation and one consumer. Obviously the 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 marking the current computation completely, some opportunity for early reset is lost for some 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 (see Table 1 which shows the number of consumers in some benchmarks). However, since it has never been proven in the context of Prolog that early reset is really worth its while (cf. also Section 5), we think that the first order of marking is probably to be preferred in practive due to 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 a consumer. We describe more in detail. Figure 6 shows a pointer *tr* which points to a CAT trail

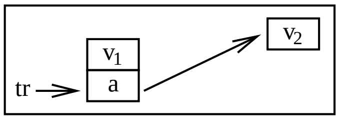



Figure 6: A CAT trail entry.

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 7 contains some pseudocode for treating one trail entry during marking: the code is different for the current computation and for the consumer. 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 have two reasons:

1. the cell was reachable from the consumer itself
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, which 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 consumers approximate it.

One conclusion is that early reset during marking of the consumers can reset the heap entries and remove CAT trail entries.

The above analysis was made for CAT. It can be applied almost straightforwardly 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 the generator will have been marked already, since the consumers that are scheduled by this generator share the trail with the generator. Figure 6 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.

## 4 Selective CAT: A minor garbage collection on saving consumers

One might be struck by the fact that while complete contiguous parts of the heap and the local stack are saved in the CAT area, the trail is copied selectively and from the choice point stack, only the top choice point is saved. [10] argues about the choice point stack that this is almost a necessity. For the trail, it was a conscious choice because it is clear that the CAT trail needs double entries and so one needs to inspect the trail entries while copying them: it was then natural to be more selective. We will show how heap and local stack can also be saved more selectively in CAT.

### 4.1 Selectively saving the heap

In general, the CAT heap contains entries that will not be reachable after the particular consumer has been reinstalled: indeed, the choice points between the consumer and the generator will no longer exist and this reduces the root set. So, the idea here is to save in the CAT heap only the part of the current heap between consumer and generator that is reachable from the current root set in *forward* execution; this takes into account that the intermediate choice points between the consumer choice point and the generator should not be saved in CAT areas. The easiest way to understand this selective saving of the heap is: assume one performs an incremental garbage collection of the heap, where only the part younger than the generator is collected and at the same time not using as root set the environments that are *only* reachable from the intermediate choice points. The result gives exactly what needs to be saved in the CAT heap. To be more precise, the marking is based on the pseudocode of Figure 8 whose underlying principle can be found in [5]. In

```
/* mark current computation */
/* B points to the consumer choice point */
e = B[E];
cp = B[CP]; /* CP is the continuation pointer */
mark_arguments_from(B);
elimit = B_G[E]; /* G is the generator up to
                    which the CAT copy is made */
while (newer(e,elimit))
   { mark_environment(e,cp);
     cp = e[CP];
     e = e[E];
   }
/* note that no other choice points are*/
/* considered for finding the useful data */
/* mark all new objects*/
/* pointed to from older regions */
tr = TR;
trlimit = B_G[TR];
while (newer(tr,trlimit))
   { mark_object(**tr);
     tr = previous(tr);
   }
```

Figure 8: Selective CAT pseudocode for marking.

this code, the mark functions must (as usual in incremental marking) refrain from marking anything that is not in the part of the heap between the generator and the consumer: the trail acts like an exception list, which contains the references from the old generation to the new generation. For clarity, we have left out early reset from Figure 8.

While the above code just shows marking for the purpose of understanding which data must be saved, it can be the first phase in a sliding garbage collection as is most common in Prolog systems, because there are good reasons to preserve the order of segments (and even of individual cells in the heap). However, as after the reinstallation of the consumer, this part of the heap is exactly one segment, a copying collector could be used as well, e.g. based on [5] or [9].

| *code for current computation* | *code for consumer* |
|---|---|
| `if (! marked(a))`<br>`{ *a = UNDEF;`<br>`remove_trail_entry(tr);`<br>`}` | `if (! marked(a))`<br>`{ *a = UNDEF;`<br>`remove_trail_entry(tr);`<br>`}`<br>`else /* marking is necessary in general */`<br>`mark_object(v1);` |

Figure 7: Pseudocode for performing early reset during marking of the current computation and of consumers.

### 4.2 Selectively saving the local stack

Simillarly to the heap, the local stack can be copied more selectively in CAT. To understand this, consider which environments are reachable after the consumer is reinstalled: only the environments in the chain starting from the environment at the moment of saving the consumer: these are exactly the environments visited by the marking code above. The environments that are *only* reachable through the choice points between the consumer and generator, will not be reachable after the consumer is reinstalled (since these choice points themselves have been removed by backtracking), so we do not need to save them in the CAT area. Once this is understood, it is reasonably straightforward to see how to implement it.

### 4.3 Performance evaluation of selective copying

CAT suffers from the problem that it can perform arbitrarily worse than SLG-WAM: In CAT, an arbitrarily large portion of the stacks might have to be copied o preserve the execution state of each consumer (see [10] for an example) and thus each suspension requires an arbitrary amount of space (and thus time) to be performed. On the other hand, in the SLG-WAM suspension is implemented by just freezing the stacks via setting freeze registers which is a constant time operation. This bad behaviour of CAT actually showed up in some benchmarks where, under a particular scheduling strategy, CAT's peak memory usage was sometimes 10 times higher than for SLG-WAM (see [10]). While the more selective saving of CAT heap and CAT local stack introduced above cannot avoid the worst case performance of CAT, it was worth investigating whether it would help in practice. After all, selective saving of the CAT areas also results in smaller reinstallation, so the cost of a garbage collection at CAT creation time might be well worth it as a CAT area is saved once but might be reinstalled several times. In preparing a full implementation, we have implemented marking, and used it as a means of getting some indication on the effectiveness of the selective copying.

Table 1 shows, for the same benchmark set as that used in [10], the peak memory usage of CAT versus CAT with selective copying (both heap and local stack) in the first two columns. The figures are in bytes. The next column shows the gain in space, and the last column shows the number of consumers which create a CAT area during the running of the benchmark. It therefore also indicates the number of times an incremental garbage collection has to be performed for obtaining the gain. The figures show the potential of selective saving of the CAT areas in reducing the space requirements, but on the other hand, for this set of benchmarks, they are rather discouraging time-wise: knowing that the `read` benchmark runs in 0.86 seconds (on a SUN Ultra Sparc 2, 168 MHz), that most consumers (94%) never need reinstalling and from the remaining ones, most (93%) get reinstalled only once, it seems hardly worthwhile to do 3180 (incremental) garbage collections. The situation might of course be completely different in programs where consumers get reinstalled multiple times and if more precise completion is performed.

## 5 Concluding Remarks

The memory model of WAM-based systems that support tabling is quite complex due to the incorporation of a suspension/resumption mechanism. This, in turn makes the usefulness logic of tabling systems more complicated than that of Prolog ones. Yet, tabling systems call for even more effective and sophisticated memory management than plain Prolog systems as their space requirements are generally bigger. Despite this need, memory management issues in the context of tabled execution were unexplored. A possible reason for this may have been the fact that the SLG-WAM was till quite recently the only possible implementation scheme for the suspension/resumption mechanism that tabling needs. Indeed, our past attempts to devise an effective garbage collector for the XSB system did not enjoy much success as we found it extremely difficult to understand the reachability issues in the cactus+spaghetti stack of the SLG-WAM model; without this understanding the best we could hope for was a conservative marking scheme, about which we had absolutely no clue how imprecise it might be in practice. The recently introduced CAT model offered a new, simpler way to implement tabling and, more importantly for this paper, the means to reason about reachability and usefulness of data in tabled evaluations regardless of the underlying implementation.

The main objective of this paper was to gain a deeper understanding of the usefulness logic of tabled logic programming systems and give a reasonably complete account of issues that are involved for their effective memoty management. To do so, we first discussed design decisions and most important implementation aspects of memory management in the CAT model. With this as basis, we also fully described garbage collection for the SLG-WAM model. In addition, we showed how at the moment of CAT area creation, a minor garbage collection can be performed and the space savings that can be expected by doing so. A renewed effort for writing the garbage collector for the XSB system (for both SLG-WAM- and CAT-based incarnations of it) is under way.

Once this task is complete, it would be desirable to perform a performance study of the effectiveness/preciseness of various garbage collection methods (i.e. early reset, variable shunting, etc.) which are discussed in this paper. However, as hinted by our main objective for this paper, this is an

| benchmark | CAT | selective CAT | gain | # of consumers |
|---|---|---|---|---|
| cs_o | 23008 | 10176 | 12832 | 38 |
| cs_r | 38104 | 20956 | 17148 | 71 |
| disj | 42812 | 22812 | 20000 | 77 |
| gabriel | 54116 | 26856 | 27260 | 80 |
| kalah | 105348 | 32032 | 73316 | 89 |
| peep | 3925708 | 739632 | 3186076 | 1721 |
| pg | 1547216 | 210292 | 1336924 | 497 |
| read | 5427168 | 2203648 | 3223520 | 3180 |

Table 1: Space requirements of CAT with and without selective copying.

issue beyond its current scope. We also note that the effectiveness of these methods are to a big extent still unexplored even for plain Prolog execution.

Although our results as presented in this paper are specific to tabled logic programming implementations, we believe that the underlying concepts might prove useful, or at least give insights, to other (LP) systems that deal with asynchronous or concurrent execution or implement some variant of a suspension/resumption mechanism.

### Acknowledgements

We thank Olivier Ridoux for suggestions and comments on an earlier draft of this paper, and the anonymous reviewers for pointing out places where more clarifications were needed. The research of the second author is supported by a junior scientist fellowship from the K.U. Leuven Research Council and CCR-9702681.

### References

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

[2] K. A. M. Ali and R. Karlsson. The Muse approach to OR-parallel Prolog. *International Journal of Parallel Programming*, 19(2):129–162, Apr. 1990.

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

[4] Y. Bekkers, O. Ridoux, and L. Ungaro. Dynamic Memory Management for Sequential Logic Programming Languages. In Y. Bekkers and J. Cohen, editors, *Proceedings of IWMM'92: International Workshop on Memory Management*, number 637 in LNCS, pages 82–102, St. Malo, France, Sept. 1992. Springer-Verlag.

[5] J. Bevemyr and T. Lindgren. A Simple and Efficient Copying Garbage Collector for Prolog. In M. Hermenegildo and J. Penjam, editors, *Proceedings of the Sixth International Symposium on Programming Language Implementation and Logic Programming*, number 844 in LNCS, pages 88–101, Madrid, Spain, Sept. 1994. Springer-Verlag.

[6] C. J. Cheney. A Nonrecursive List Compacting Algorithm. *Communications of the ACM*, 13(11):677–678, Nov. 1970.

[7] M. Codish, B. Demoen, and K. Sagonas. Semantics-Based Program Analysis for Logic-Based Languages using XSB. *Springer International Journal of Software Tools for Technology Transfer*, Oct. 1998. To appear.

[8] S. Dawson, C. R. Ramakrishnan, and D. S. Warren. Practical Program Analysis Using General Purpose Logic Programming Systems — A Case Study. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 117–126, Philadelphia, Pennsylvania, May 1996. ACM Press.

[9] B. Demoen, G. Engels, and P. Tarau. Segment Preserving Copying Garbage Collection for WAM based Prolog. In *Proceedings of the 1996 ACM Symposium on Applied Computing*, pages 380–386, Philadelphia, Feb. 1996. ACM Press.

[10] B. Demoen and K. Sagonas. CAT: the Copying Approach to Tabling. In H. Glaser, K. Meinke, and C. Palamidessi, editors, *Proceedings of the Joint Conference on PLILP/ALP'98*, number 1490 in LNCS, pages 21–35, Pisa, Italy, Sept. 1998. Springer-Verlag.

[11] P.-E. Moreau. A choice-point library for backtrack programming. In K. Sagonas, editor, *Proceedings of the JICSLP-98 Workshop on Implementation Technologies for Programming Languages based on Logic*, pages 16–31, Manchester, U.K., June 1998.

[12] F. L. Morris. A Time- and Space-Efficient Garbage Compaction Algorithm. *Communications of the ACM*, 21(8):662–665, Aug. 1978.

[13] E. Pittomvils, M. Bruynooghe, and Y. D. Willems. Towards a Real Time Garbage Collector for Prolog. In *Proceedings of the 1985 Symposium on Logic Programming*, pages 185–198, Boston, Massachusetts, July 1985. IEEE Computer Society Press.

[14] Y. S. Ramakrishna, C. R. Ramakrishnan, I. V. Ramakrishnan, S. A. Smolka, T. Swift, and D. S. Warren. Efficient Model Checking Using Tabled Resolution. In O. Grumberg, editor, *Proceedings of the 9th International Conference on Computer-Aided Verification*, number 1254 in LNCS, pages 143–154, Haifa, Israel, July 1997. Springer-Verlag.

[15] K. Sagonas and T. Swift. An Abstract Machine for Tabled Execution of Fixed-Order Stratified Logic Programs. *ACM Transactions on Programming Languages and Systems*, 20, 1998. To appear.

[16] K. Sagonas, T. Swift, and D. 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, Minneapolis, Minnesota, May 1994. ACM Press.

[17] D. H. D. Warren. An Abstract Prolog Instruction Set. Technical Report 309, SRI International, Menlo Park, U.S.A., Oct. 1983.

[18] D. S. Warren. Efficient Prolog memory management for flexible control strategies. In *Proceedings of the 1984 Symposium on Logic Programming*, pages 198–202, Atlantic City, New Jersey, Feb. 1984. IEEE Computer Science Press.