Prolog and Abduction 4 writing Garbage Collectors*

Bart Demoen

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

Abstract. It seems silly and impractical to write a garbage collector (for a Prolog engine) in Prolog itself. But doing so has least three advantages: (1) one gets a runnable specification of the garbage collector, (2) it can enhance understanding of the algorithms involved without having to worry about pointer details, (3) the garbage collector written in Prolog can be used for debugging its implementation in a lower level language. We show how a sliding collector can be specified in Prolog in a reasonably declarative way, how it can be executed best within a tabling environment, and how it was of use during the development of the heap garbage collectors of BinProlog, XSB and more recently ilProlog. We indicate how the Prolog implementation can reconstruct the classical Morris algorithm. We also specify the garbage collection process in an abductive formalism and speculate on how constraints on the abductive solver can retrieve well-known algorithms.

1 Introduction

This work originates from our involvement in writing garbage collectors for different systems: in the mid-eighties, it was for BIM_Prolog [1], in 1994 for BinProlog [4] ([6]) and from 1996 on for XSB ([8],[9]). More recently, we wrote a heap garbage collector for ilProlog (see [7]). Because of the roots of our work, the setting of the current paper is garbage collection of a WAM heap - or global stack as it is also named. Therefor, we will assume some knowledge of WAM: see for instance [1]. Also some knowledge of the algorithms described in [12], [5] and of heap garbage collection for Prolog in general, e.g. from [2], [3], will be helpful.

One recurring problem in the implementation of such garbage collectors is of course debugging them: one is bound to make errors in the pointer fiddling involved and also, the adaptations to one's own need of the algorithms in literature can be erroneous in a way that is only discovered after the C code has been written. This is often caused by the fact that memory management is added as an after-thought to the system and partly because usually one does not understand at all the usefulness logic of a system until one gets down to implementing a (precise) garbage collector for it. One popular way to debug is to insert print statements all over in the source code. Also debugging tools like gdb or dbxtool can be handy, but they are notoriously unhelpful in debugging problems that show up in another part of the code than where the bug really is. Also, hexadecimal memory dumps (which such tools give you easily) are not the most pleasant objects to deal with and act like the proverbial haystack. Still, bugs must be found. And when a bug - or set of bugs - cannot be eliminated, the ultimate resort might be to rewrite the code from scratch: this actually happened in the BIM_Prolog project. In 1994, about 10 years after the BIM_Prolog experience, we went into implementing the garbage collectors for BinProlog [6]: the first version by Geert Engels in the context of his engineering thesis [2] was quite good, but we knew that debugging would come in sooner or later, especially since BinProlog has features like a blackboard, findall bubbles in the heap, backtrackable destructive assignment and term

* Extended Abstracts of LOPSTR 2000, Tenth International Workshop on Logic-based Program Synthesis and Transformation, 24-28 July 2000, London, UK. Technical Report Report Series, Department of Computer Science, University of Manchester, ISSN 1361-6161. Report number UMCS-00-6-1. http://www.cs.man.ac.uk/cstechrep/titles00.html

[1] now named MasterProLog

[2] masters level elsewhere

compression. We anticipated that a decent stack dump would be crucial and we then already chose a Prolog format: heap, choice point stack and trail (BinProlog has no environment stack) could at any moment be dumped in the form of Prolog facts. We benefitted greatly from the stack dumps in Prolog format, but we didn't really use them much for debugging the garbage collectors, because Geert Engels did a very good job. When we started implementing the XSB [13] collector in 1996, we actually first (re)implemented the stack expansion routines as a warm up - to get used to the XSB macros, tagging schema etc. Unfortunately (but not unexpectedly) we even needed to debug our stack expansion routines - not just because they were buggy, but mainly because the invariants we were lead to believe were not respected by the rest of the implementation or plainly undocumented and forgotten by the rest of the team. XSB did have functions for dumping the stacks in a symbolic way and they were of some help. But we were missing something: the dumps could be looked at by humans, but could not easily be treated with programmable tools like Prolog. Basically the format of the dump was wrong: it should have been in Prolog syntax because that allows easy general processing, in particular, dumps in Prolog syntax can be consulted. So we changed the stack dump routines to produce output in the form of Prolog facts. We'll give some more details on the Prolog format later and how we used it. The point about syntax seems very trivial and to some extent it is of course. But without this last step in the series of trivial steps from a hexadecimal core dump, over some human readable form to Prolog facts, we would have made no progress in this area, as Prolog is for us the main universal programming tool. And more bugs in our garbage collectors would have survived for a longer time.

The first use of these Prolog dumps of stacks was just for consistency checking in the context of the expansion routines: does the expanded stack have as many live cells as the original; are they in the same order; is every pointer relocated correctly ... Some of them are simple checks that could have been performed in a different way, but the Prolog format made life already easier. Later we used the Prolog dumps for checking the garbage collection process: are there as many surviving cells as marked ones; are all root pointers redirected appropriately [3] etc.. Such simple properties can be checked with simple - non-recursive - Prolog queries. The more we used the stack dumps, the more we appreciated their usefulness and we started asking queries that involve recursion: e.g. is every reachable cell also marked ? Such recursive queries can only naturally be asked in a tabling environment. So it was very fortunate that we were working in and with XSB at that time.

After some time, we decided not just to *check* dumps but also to *collect* dumps, i.e. given a dump of the stacks **before** the garbage collection, produce independently of the actual collection a dump of the collected heap and then compare with what the real garbage collector produced. So the idea occurred to us that the collector itself can be specified and executed in Prolog, as a shadow and specification of the real one written in C and as a guarding angel, or watch dog, for the latter. In later sections we describe a heap garbage collector for Prolog and how it was (or could have been) used for debugging. Indeed, our description will be a mixture of what we actually did and how we now think it should/could have been done. Also, we'll confuse what happened during the development of BinProlog, XSB and ilProlog.

Having a garbage collector specified in Prolog is nice, but not completely satisfactory: when several algorithms are used in the implementation [4], one needs to write more than one Prolog program to mimic them. The other disadvantage of the Prolog program is that its correctness - however much more obvious than the correctness of the corresponding C code - must be established. So we wanted a more generic specification of the garbage collection process in logic and one that allows a higher confidence in its correctness. Because of active research in abduction around us, we started getting interested in specifying a garbage collector in an abductive formalism. Within this formalism, one has the possibility to express the logical relationship between objects - in this case e.g. the heap before and after collection - without having to implement a particular algorithm. Indeed, there is only a little point in implementing the same algorithm in Prolog as in C, as one is bound to make the same conceptual errors. And while a garbage collector written in Prolog describes one particular instance of an algorithm, in practice, one is also interested in the whole class that a generic algorithm represents. So we made an attempt at specifying a generic and abductive garbage collector: see section 5. Two instances of it will specify the sliding collector (completely) and the class of copying collectors. We will also speculate that by giving constraints on the order in which facts can be abduced, one actually can reconstruct the classical algorithms of Morris and Cheney, and given strong enough compile time garbage collection they will have the same space performance as an imperative specification of the same algorithms.

[3] redirection of pointers is more involved in the garbage collection context than in the expansion case

[4] BinProlog and XSB had/has both a sliding and a copying collector

2 Dumping the stacks as Prolog facts

We implemented a (builtin) predicate print_all_stacks/0 and also made it possible that the same functionality is called from within e.g. gdb. It produces files LS_i, $HEAP_i$, CP_i, TR_i, $REGS_i$ when print_all_stacks/0 is called for the i-th time during a particular run of the system. [5] These refer to the local stack, heap, choice point stack, trail stack and WAM registers. There was also functionality to print these selectively (i.e. some portions of some of the stacks). Some small samples from these files should be enough to get the gist; from HEAP:

```
heap( 27, not_marked, funct, '_$abort_cutpoint'/1).
heap( 28, not_marked, int  , 72).
heap( 29, not_marked, funct, ':-'/2).
heap( 30, not_marked, struct-ref_heap, 27).
heap( 31, not_marked, atom , 'true').
```

From LS:

```
ls( 57, not_marked, ref_ls, 50).
ls( 58, not_marked, code, 136162920-putpbreg).
ls( 59, not_marked, int  , 224).
ls( 60, not_marked, atom , '| ?- ').
ls( 61, not_marked, int  , 1).
ls( 62, not_marked, int  , 260).
ls( 63, not_marked, ref_ls, 63).
```

The first argument is the index in the array used for implementing the stack; the second argument indicates whether the entry is marked or not [6]; the third argument describes what sort of item is there and the last one its value. Most symbols are self-explanatory. *putpbreg* is an XSB specific abstract instruction.

The building of such printing routines is an art in itself on which we'll not dwell too much here: depending on the desired level of accuracy of the printing routines and the actual optimization of the representation, it might take a lot of effort to get at facts as above. But one also wants to cater for **impossible** data, i.e. stack contents that occur in abnormal (read: buggy) situations, e.g. when garbage collection has incorrectly relocated a pointer to the heap, to a pointer in between heap and local stack (which is a relevant thing to consider in the context of the stack allocation of XSB). So, the dump could contain things like:

```
ls( 666, marked, between_h_ls, 17/23).
```

where the 17 means the index from the start of the heap and 23 is rom the start of the local stack. This is useful, because the origin of such a between_h_ls pointer could be either the local stack or the heap.

We also employed a category of unexpected pointers; so the output

```
heap( 137, not_marked, strange, 123).
heap( 138, not_marked, ref_nowhere, 4).
```

would indicate a strange tag and a reference into a non-identifiable region. Such output usually indicates a bug, an object the printing routines are not yet aware off or a badly understood invariant. It is important that the printing routines don't stop printing at the first occurrence of something unusual: later strange things might make it easier to find the origin of the problem.

Both in XSB and BinProlog, every entry in the stacks is tagged, so that an entry can be printed in isolation. When there are entries on the stack that are not tagged and have no wrapping entries on the same stack (like in ilProlog) it becomes much more difficult to make sense of them.

Another issue is that we want such a stack dump to be possible at (almost) any moment during the collection process and even general execution. E.g. after (or during) marking, in between the up- and downward phase of a sliding collector or in the middle of a copying process.

There is one problem with the idea of the Prolog syntax: once it is stated explicitly that stack dumps must be in Prolog, it seems so obviously right that the statement itself becomes pedantic.

[5] for XSB there is also a $CHAT_i$, while BinProlog lacks LS_i

[6] since the dump was done before a marking took place, everything is unmarked

3 Simple queries on the dumps

Checking elementary invariants is quite easy by means of a Prolog query now, e.g. it is a WAM invariant that all structures are on the heap, which means that whenever a struct-X (a pointer with a structure tag) is found, the X should be ref_heap and it should point to a heap cell which contains a funct. The corresponding query to check this is at the left:

```
?- struct_pointer(I,WhereTo,Index),
   (WhereTo = ref_heap -> true ; error(I,Index)),
   heap(Index,_,What,_),
   (What = funct -> true ; error(I,Index)).

struct_pointer(I,WhereTo,Index) :-
        ( ls(I,_,struct-WhereTo,Index)
        ; heap(I,_,struct-WhereTo,Index)
        ; aregs(I,struct-WhereTo,Index)
        ; cp(I,_,struct-WhereTo,Index)
        ).
```

```
?- heap(N,_,list-ref_heap,I),
   \+((heap(I,_,_,_),heap(I+1,_,_,_))),
   error(list_points_not_in_heap(N)).
```

At the right is a query that detects list tagged pointers (on the heap) that do not point to two consecutive cells that are not both on the heap: that situation could result from an erroneously implemented builtin.

More complicated invariants can be easily stated and checked.

In the context of garbage collection, the most crucial property of a cell is whether it is reachable from the root set. So, we implement a predicate reachable/1 which given a dump can generate all reachable cells on the heap, or test the reachability of a particular cell. The following program specifies this property:

```
:- table is_reachable_from/2, reachable/1.

reachable(Cell) :-
        root(Root),
        is_reachable_from(Cell,Root).

is_reachable_from(A,B) :-
        A = B.
is_reachable_from(A,B) :-
        heap(B,_,list-ref_heap,N),
        ( is_reachable_from(A,N)
        ; is_reachable_from(A,N+1)
        ).

is_reachable_from(A,B) :-
        heap(B,_,struct-ref_heap,N),
        is_reachable_from(A,N).
is_reachable_from(A,B) :-
        heap(B,_,funct,Name/Arity),
        between(X,B+Arity+1,B+Arity+N),
        is_reachable_from(A,X).
```

Note that the second argument of *heap*/4 was ignored in the above code: *reachable*/1 is indeed meant to check reachability independent of the actual marking phase.

It is clear that *reachable*/1 could have been written without resorting to tabling: one just has to program the loop check [7]. But it is also clear that with tabling, *reachable*/1 is easier to write, easier to understand as correct and even more efficient, since we use it often, as in the following program which checks whether the marking phase has indeed marked all and only the reachable cells.

```
check_marking :-
    heap(I,marked,_,_), \+(reachable(I)),
    error(marked_but_not_reachable(I)).
check_marking :-
    heap(I,not_marked,_,_), reachable(I),
    error(reachable_but_not_marked(I)).
```

Reachable/1 is also tabled, so that when it is called with ground argument, XSB succeeds it at most once.

[7] loops can occur because the implementation supports rational trees, or simply doesn't do occurs check, or because the heap is corrupted by bugs

4 A sliding collector in Prolog

We will specify a sliding collector for a Prolog heap (see [12], [2]). Assume the (old - before collection) heap was dumped in Prolog list format, then the predicate sliding_gc/4 will produce the new (after collection) heap as a list as well.

```
% sliding_gc/4: argument 1 and 2 are the old (input) and new (output) heap
%               argument 3 is the next index in NewHeap (input)
%               argument 4 is the translation table for the indices (input)

sliding_gc([],[],_,_).

sliding_gc([old_heap(N,ref_heap,M)|RestOldHeap],NewHeap,NextI,TransTable) :-
    reachable(N),
    NewHeap = [new_heap(NextI,ref_heap,NewM)|RestNewHeap],
    putin(TransTable,N->NextI,T1),
    putin(T1,M->NewM,NewTransTable),
    sliding_gc(RestOldHeap,RestNewHeap,NextI+1,NewTransTable).

sliding_gc([old_heap(N,atom,Name)|RestOldHeap],NewHeap,NextI,TransTable) :-
    reachable(N),
    NewHeap = [new_heap(NextI,atom,Name)|RestNewHeap],
    putin(TransTable,N->NextI,NewTransTable),
    sliding_gc(RestOldHeap,RestNewHeap,NextI+1,NewTransTable).

sliding_gc([old_heap(N,WhatTo-ref_heap,M)|RestOldHeap],NewHeap,NextI,TransTable) :-
    reachable(N),
    (WhatTo = list ; WhatTo = struct),
    NewHeap = [new_heap(NextI,WhatTo-ref_heap,NewM)|RestNewHeap],
    putin(TransTable,N->NextI,T1),
    putin(T1,M->NewM,NewTransTable),
    sliding_gc(RestOldHeap,RestNewHeap,NextI+1,NewTransTable).

sliding_gc([old_heap(N,funct,Name/Arity)|RestOldHeap],NewHeap,NextI,TransTable) :-
    reachable(N),
    NewHeap = [new_heap(NextI,funct,Name/Arity)|RestNewHeap],
    putin(TransTable,N->NextI,NewTransTable),
    sliding_gc(RestOldHeap,RestNewHeap,NextI+1,NewTransTable).

sliding_gc([old_heap(N,_,_)|RestHeap],NewHeap,Place,TransTable) :-
    \+(reachable(N)),
    sliding_gc(RestHeap,NewHeap,Place,TransTable).
```

The above code implements a sliding collector (although by Prolog's nature not an in-place one) and since also Morris' algorithm does so, one can wonder to what extent the above Prolog code hides and/or contains Morris' algorithm.

First of all, the above code shows no separate marking phase: the imperative description of Morris' algorithm executes a marking algorithm first, because it is too difficult to describe it otherwise. The code above executes marking as a result of the execution of the reachable/1 goal and the role of tabling is very prominent: the mark bits - so essential in the efficient imperative description - are allocated and implemented by the tabling mechanism; the tables indeed remember whether a heap entry is marked or not. Should we hope for an automatic transformation from the encoded information in the table to the more efficient imperative representation ? Or should we actually wonder whether the potentially more dense tabled information is more efficient than allocating one bit per heap entry ? For us, the issue is not clear cut.

The relocation of root pointers can be implemented in more than one way: a separate piece of code could use the translation table that is build up. But in the spirit of Morris, one could add the root set (virtually) at one end of the heap.

We have kept deliberately the third argument of sliding_gc/3 as an abstract data type and not given code for putin/3. This argument is meant to hold the translation of old heap addresses to new heap addresses.

One cannot see the Morris upward and downward pass in the Prolog code above; on the contrary: the above code does only one pass and it is a downward pass. However, imagine that the unifications NewHeap = ... are delayed, as well as the calls to putin/1, until after the recursive call to sliding_gc/4, then it becomes more plausible that the two-pass algorithm is indeed close. Together with the right implementation of the abstract data type of the TransTable, one might retrieve a Morris' algorithm, but still not in-place.

However, the old heap is - in the terminology of the Mercury [11] - destructive input while the new heap is unique output. This means that the list representing the old heap can be reused by the newly constructed new heap and finally we do have an in-place sliding collector using LP technology !

The above is partially speculation, but we believe that it is worth exploring further. Note also that to achieve some of the needed transformations from our Prolog program to the imperative Morris algorithm, it might be necessary to let compile-time garbage collection reason about run time stack frames (environments that build up in the version in which the recursive call is not in tail position) as well as data structures, or alternatively that one might have to make environments - or continuations - explicit in the Prolog code perhaps by a binarization transformation like in BinProlog.

We have in a similar style a Prolog version of a copying collector based on Cheney's algorithm and similar remarks apply there.

We could have written the collector above in many different ways, e.g. by having the marking completely before the sliding, or not by building up the lists of old and new heap, but by using the heap/4 facts directly and asserting new_heap facts. The above however is one of the versions that comes to mind naively and is obviously correct: since debugging another collector was the original purpose, (executable) correctness had to be the most desirable property of the Prolog version collector. However, in section 5 we will see a different level of specification and correctness.

5 A set of abductive copying collectors

The code for the sliding collector in section 4 is reasonably declarative: it combines the advantage of being executable with a high degree of declarativity. However, it implements one particular garbage collection algorithm, which is moreover deterministic in the sense that the relationship between the old and new heap is completely determined. This is not the case for other algorithms e.g. copying compacting, which allow for different new heaps, depending on whether one uses a depth-first or a breadth-first copying algorithm, and depending on the order in which root pointers are treated. Sliding has no such non-determinism: once marking is done, there is only one possible new heap. Clearly, Prolog is not a good language to express such non-determinism.

In an abductive setting (see for instance [10]), one can specify the relation between the old and the new heap, without a specification of how this relation is to be realized. Moreover, the natural setting is by making the heap after collection an abducible (also called open) predicate, say new_heap/3. Also the correspondence between the heap before and the heap after collection needs to be made explicit, and this is achieved by the abducible predicate transtable/2 [8], which gives for any non-garbage cell the correspondence between its old and new address. The abductive specification allows for non-determinism in the sense above. We start by specifying **all** new heaps that can correctly result from a collection, given a particular old heap (old_heap/3) and a definition of reachability (reachable/1).

```
:- open_predicate(transtable/2), open_predicate(new_heap/3).

% ensure that the new heap is
% filled from the first entry
(exists X: transtable(X,1)) <-
        reachable(_)

% ensure that the new heap
% does not have holes
(exists X: transtable(X,Y)) <-
        transtable(_,A),
        transtable(_,B),
        A < Y,
        Y < B
```

[8] one can give an abductive specification without transtable/2, but it is very clumsy

```
% ensure that a reachable cell is
% "copied" at most once
     <-
        transtable(A,A1),
        transtable(A,A2),
        A1 =\= A2
```

```
% ensure that each reachable cell is
% "copied" at least once
(exists Y: transtable(X,Y)) <-
        reachable(X)
```

```
% ensure that the copy contains the correct contents
new_heap(M,SorL-ref_heap,R) <-
        old_heap(N,struct-ref_heap,S),
        (SorL = struct ; SorL = list),
        transtable(N,M),
        transtable(R,S),
        reachable(N)
```

```
new_heap(M,ref_heap,R) <-
        old_heap(N,ref_heap,S),
        transtable(N,M),
         transtable(R,S),
         reachable(N)
```

```
new_heap(M,ForA,V) <-
        old_heap(N,ForA,V),
        (ForA = funct ; ForA = atom),
        transtable(N,M),
        reachable(N)
```

```
% successive fields in a structure or list cons must remain successive
     <-
        old_heap(N,funct,Name/Arity),
        reachable(N),
        transtable(N,N1),
        0 < X, X =< Arity,
        Y is N + X,
        transtable(X,X1),
        N - N1 =\= X - X1
```

```
     <-
        old_heap(N,list-ref_heap,R),
        reachable(N),
        transtable(R,R1),
        0 < X, X =< 2,
        Y is N + X,
        transtable(X,X1),
        R - R1 =\= X - X1
```

The above formulation of a garbage collection process allows for many correct new_heaps corresponding to the same heap before collection and particular reachability. In particular, it allows for a sliding collector which can be enforced by the constraint which retains the order of heap cells:

```
 <-
    reachable(N), reachable(M), N < M,
    transtable(N,N1), transtable(M,M1),
    M1 < N1
```

Without this constraint, the order is not necessary retained and the resulting collectors correspond to so called copying collectors.

The above specification makes (must make !) explicit usually hidden assumptions about a garbage collector, e.g. that each object, even if reachable from more than one root cell, should end up only once in the WAM heap after collection. The above specification does not allow for what is called semantic garbage collection, neither for term compression during the collection. But the constraint where this restriction is put is clearly visible.

The use of the abductive specification is currently restricted to checking instead of generating: indeed, current abductive execution mechanisms - such like SLDNFAC - are probably not able to compute new heaps from the above specification in a reasonable time. Also in this setting, we can wonder whether classical algorithms can be derived. The in-place sliding collector requires that it is possible to put a constraint on the order in which new_heap facts are abduced: indeed, the idea would be to let a new_heap fact reuse the space occupied by an old_heap fact; so the constraint on the order of deriving the new_heap facts should certainly include that the corresponding old_heap fact is no longer needed. For deriving a copying compacting collector - which leaves forwarding pointers in the old heap - it would be necessary to let the forwarding pointers - which reside in the transtable/2 facts - reuse the old_heap facts. Apart from the fact that such reuse reasoning is not straightforward in general, there is the extra obstacle that within an abductive theory, there are no ways to guide the abductive process. Still, we think these issues are worth to investigate further.

6 Conclusion

Apart from an aid during the debugging of the garbage collectors, Prolog dumps of the stacks have helped us many times in discovering bugs in the implementation of builtin predicates, like e.g. copy_term/3, and in finding memory leaks in builtin predicates. So their use is quite wide.

It is clear that we have simplified certain issues: we have not dealt with early reset, variable chain collapsing, trail compaction during garbage collection etc.. But it is clear that these can be treated in Prolog as well as in the abductive approach.

During an informal meeting at ILPS'91 with other implementors we suggested that the garbage collector (of a particular Prolog system) could have been written in Prolog itself. The idea caused laughter: writing a garbage collector for Prolog in Prolog was clearly absolutely, totally and in all other ways, inconceivable [9]. But ever since then, we have wondered whether it really isn't the right thing to do. The disadvantage is obviously performance and also, how can one be sure that the garbage collector is not going to overflow the heap, causing itself to be called ? So there is definitely a challenge there. However, the Prolog version has a bigger chance of being correct. It will provide more insight in the garbage collection process and it will perhaps help in understanding how seemingly totally imperative algorithms can be reconstructed from within the LP framework, be it Prolog or abduction. We will continue research in this direction.

Acknowledgements

We want to thank Geert Engels, Paul Tarau and Kostis Sagonas for the pleasant collaboration while working on the garbage collectors of BinProlog and XSB respectively. The BIM_Prolog experience was also essential for this paper, but rather painful, so we'll omit names.

References

1. H. 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.
2. 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.
3. 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.
4. BinProlog: see http://www.binnetcorp.com/BinProlog
5. C. J. Cheney. A Nonrecursive List Compacting Algorithm. Communications of the ACM, 13(11):677–678, Nov. 1970.
6. B. Demoen, G. Engels, and P. Tarau. Segment Preserving Copying Garbage Collection for WAM based Prolog. In Proceedings of the 1996 ACM Symposium on Applied Computing, pages 380–386, Philadelphia, Feb. 1996. ACM Press.
7. B. Demoen, G. Janssens, H. Vandecasteele. Executing Query Flocks for ILP. Proceedings of BENELOG'99, Maastricht, 5 November 1999
8. B. Demoen and K. Sagonas. Memory Management for Prolog with Tabling. In Proceedings of ISMM'98: ACM SIGPLAN International Symposium on Memory Management, pages 97–106, Vancouver, B.C., Canada, Oct. 1998. ACM Press.
9. B. Demoen, K. Sagonas. Heap Garbage Collection in XSB: Practice and Experience Proceedings of the Second International Workshop on Practical Aspects of Declarative Languages (PADL'00), Boston, Jan. 2000, pp. 93–108
10. M. Denecker, and D. De Schreye. SLDNFA: an abductive procedure for normal abductive programs J. Logic Programming 34 (1998), no. 2, 111-167.
11. Mercury: see http://www.cs.mu.oz.au/research/mercury/.
12. F. L. Morris. A Time- and Space-Efficient Garbage Compaction Algorithm. Communications of the ACM, 21(8):662–665, Aug. 1978.
13. XSB: see http://www.cs.sunysb.edu/~sbprolog/.

[9] thank you William Goldman