

**Marking in the presence of destructive
assignment is suboptimal.**

Bart Demoen

Report CW 302, October 2000



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

Marking in the presence of destructive assignment is suboptimal.

Bart Demoen

Report CW302, October 2000

Department of Computer Science, K.U.Leuven

Abstract

Early reset (or virtual backtracking) consists of an action performed by the collector, while normally it is performed by the mutator. Roughly speaking, the precondition for early reset is that a cell is not useful for the forward continuation. Later reference from the trail completes the picture. When the precondition for early reset does not hold because the cell is marked, and in the context of backtrackable destructive assignment by value trailing, the usual approach to early reset is safe but suboptimal. This is explained by one example. In spirit with other pathological examples that show suboptimality of supposedly well understood reachability issues, choicepoint trimming is crucial. The relation with a rejuvenating collector and an alternative implementation of backtracking destructive assignment is discussed.

Marking in the presence of destructive assignment is suboptimal.

Bart Demoen

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

Abstract

Early reset (or virtual backtracking) consists of an action performed by the collector, while normally it is performed by the mutator. Roughly speaking, the precondition for early reset is that a cell is not useful for the forward continuation. Later reference from the trail completes the picture. When the precondition for early reset does not hold because the cell is marked, and in the context of backtrackable destructive assignment by value trailing, the usual approach to early reset is safe but suboptimal. This is explained by one example. In spirit with other pathological examples that show suboptimality of supposedly well understood reachability issues, choicepoint trimming is crucial. The relation with a rejuvenating collector and an alternative implementation of backtracking destructive assignment is discussed.

1 Introduction

Knowledge of the WAM [12] is essential for reading this paper. A good introduction is in [1]. Also general knowledge of garbage collection is helpful - see for instance [8]. Early reset was introduced in [2, 10] and is also explained in [3] which contains a good overview of garbage collection issues in logic programming implementations. Many systems with backtrackable destructive assignment - whether by an untyped setarg/3 construct or the more disciplined mutable variables - have adapted early reset to that context in a safe way. However, we will show by **one** example that usual marking with just one mark bit is not enough for precise identification of useless data. This might in principle lead to unnecessary memory leaks. Therefore, the usual techniques for non-early reset are - in the presence of backtrackable destructive assignment - suboptimal. Since choicepoint trimming is crucial in this story - and because it seems not widely known nor implemented - we start by explaining it in section 2. Section 3 shows the example that clarifies the issue.

2 Choicepoint trimming

Consider the following example:

progl	
	p(X) :- do_something_with(X). p(-).

The usual WAM code (in the XSB - see [13]) - version of the WAM instructions) is

code1	
	try arity=1, @first trust arity=1, @last @first: execute do_something_with/1 @last: proceed

Since the argument to p/1 is not used in the second clause, there is no need to save the argument in the choicepoint and subsequently to restore it on the trust instruction. So, the following abstract code is also correct:

code2	
	try arity=0, @first trust arity=0, @last @first: execute do_something_with/1 @last: proceed

The difference is only in the fact that the latter code does not save any argument in the choicepoint: the choicepoint of p/1 is trimmed. It contains just enough information to install the failure continuation, and not more: the amount of data to save and restore is determined at compile time. No runtime overhead is involved, on the contrary, it saves machine instructions.

It has also another side effect: when during the execution of do_something_with/1 garbage collection occurs, the failure continuation of p/1 will not keep the argument with which p/1 was called alive, thus leading to potential great savings.

One might think that this situation does not occur often in practice, but consider the normal definition of member/2:

prog2	
	member(X,[X _]). member(X,[_ R]) :- member(X,R).

Now apply unification factoring and obtain:

prog3	
	member(X,[Y R]) :- member3(X,Y,R). member3(X,X,_). member3(X,-,[Y R]) :- member3(X,Y,R).

We now see a situation common in opportunities to apply choicepoint trimming: the second argument of member3/3 is not used on backtracking, so there is potentially some memory gain by applying choicepoint trimming.

As one can see, chances for choicepoint trimming might depend on other optimizations.

It might not be clear that choicepoint trimming is worth its while in an emulator: one might need new specialized abstract machine instructions or emit more abstract machine instructions. Both are potentially bad for performance.

Choicepoint trimming was implemented in [11] and possible elsewhere.

Note that the effect of choicepoint trimming can to some extent be obtained by the disjunction transformation and environment trimming and/or good variable classification. The latter can be seen from a transformed member3/3 predicate to:

prog3'	
	<pre>member3(X,Y,Z) :- X = Y ; Z = [A R], member3(X,A,R).</pre>

Y should be classified as temporary, meaning that it is not visible in the second branch.

3 The example

Assume choicepoint trimming.

prog4	
	<pre>query :- X = f(bar(a)), b(X). b(Y) :- setarg(1,Y,foo), gc, use(Y). b(_). ?- query.</pre>

The goal gc symbolizes a computation that invokes garbage collection. Since Y is in the continuation of the collection, it is marked. Since the location that $setarg$ updates was created before the choicepoint of $b/1$, the $setarg$ assignment was trailed and there is a reference on the trail to the term $bar(a)$. Since the argument of the $f/1$ term is marked, the marking phase also must mark the term $bar(a)$. But it is obviously clear that the term $bar(a)$ will not be used ever. Still, the WAM has no way to avoid the value trailing, neither can usual garbage collection avoid to mark that term, because the situation is - with ordinary marking - not distinguishable from the one occurring in the following program:

prog5	
	<pre>query :- X = f(bar(a)), b(X). b(Y) :- setarg(1,Y,foo), gc, use(Y). b(Y) :- use(Y). ?- query.</pre>

A series of pictures referring to the above programs will further clarify this.

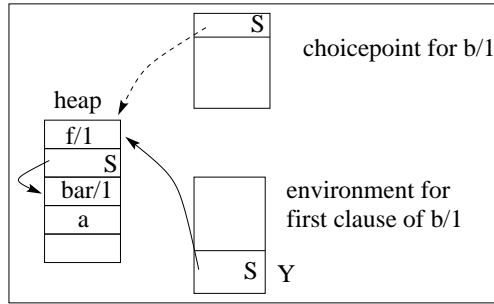


Figure 1: Just before the setarg/3 goal

Figure 1 shows the situation just before the setarg is executed. X is not in the picture, because it is already dead. There is a dashed arrow from the choicepoint of $b/1$ pointing to the term which is the argument to the call to $b/1$: with choicepoint trimming for program 4, the dashed line is not there because the argument would not be saved in the choicepoint. The term $f(\text{bar}(a))$ is kept alive by the environment variable Y . Without choicepoint trimming (or for program 5), the dashed line becomes solid and in fact keeps also the term alive.

Figure 2 shows the situation after the execution of setarg/3 and the value trail was added.

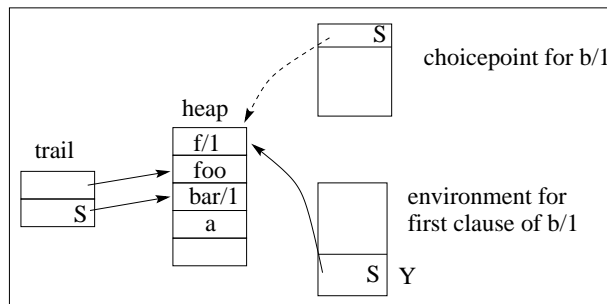


Figure 2: Just after the setarg/3 goal

To understand completely the issue, we have to look at the marking phase under different assumptions. The order of visiting root cells is environment, trail, choicepoint.

After marking from the environment, the situation is as on figure 3: we have indicated that a cell is marked by shading it.

Now we must treat the trail: since the pointer cell in the trail points to a marked cell (containing foo) no early reset can be performed. So the question is: must the value cell in the trail (it is the S pointer to $\text{bar}/1$) be used as a root for marking or not.

Consider both options:

- **do not mark from the value cell:** assume the dashed arrow is full; then later treatment of the choicepoint will use the saved argument as a root pointer, but it will immediately find a marked cell and stop; consequently, the cells containing $\text{bar}(a)$ will not be marked and collected; later backtracking will not be able to correctly restore the old value for the argument to $b/1$

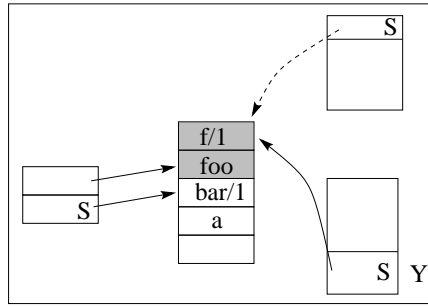


Figure 3: After marking Y from the environment

- **do mark from the value cell:** now assume the dashed arrow is not there at all; this is the case with choicepoint trimming; the cells containing $bar(a)$ will be marked; but there will be no future use of these cells: effectively, some garbage has survived a collection

The first option is plainly wrong, the second is suboptimal. The culprit is now easier to see: marking stops when seeing a marked cell. Suppose we define a new marking procedure $mark'$ which does not stop (working on a branch) when a marked cell is encountered, then one could proceed as follows.

- for each segment from new to old do
 - mark' from the environments
 - reinstall the old values from the trail while storing the new values
 - mark' from the choicepoint
- reinstall all values from the trail while storing the old values

Clearly this $mark'$ procedure which does not use the mark bit to avoid repeated marking of the same cell, has bad properties. But it solves the suboptimality problem.

4 Related issues

The situation is similar to the one in [6, 7]: several continuations - be it failure or success continuations - have their own view on the usefulness of data, but marking with a boolean is crude. The boolean says basically only

**there is some continuation which needs this location
and possibly everything that is referred by it**

while a more fine grained marking notion would also be able to tell **which** continuation needs the cell. Enhancing marking along that line makes marking lose its $O(\text{useful_data})$ property, but it would become more precise in its identification of garbage: the same was hinted at in [6, 7].

Another way of viewing at these matters is as follows: assume that the code were

prog6	
	<pre> query :- X = f(bar(a)), b(X). b(Y) :- gc1, setarg(1,Y,foo), gc2, use(Y). b(_). ?- query.</pre>

where *gc1* and *gc2* symbolize computations that invoke garbage collections. In the setting of [5] the term associated to *Y* is moved to the newer segment, i.e. the segment younger than the choicepoint for *b/1*. This is correct as the only reference to this term, resides in the newest segment. Consequently, the *setarg* operation will not be trailed. As a further consequence, the term *bar(a)* will be collected during the garbage collection in *gc2*. Note that even with the rejuvenating property of [5], one garbage collection on its own is not enough to achieve the desired effect of collecting the term *bar(a)*: indeed, marking happens before rejuvenation.

Now consider the variant:

prog6'	
	<pre> query :- X = f(bar(a)), b(X). b(Y) :- setarg(1,Y,foo), gc1, gc2, use(Y). b(_). ?- query.</pre>

It now looks as if doing two garbage collection in succession, recovers more space than doing just one collection. This is in line with the findings of [4] where one of the conclusions was that marking and early reset should be performed more than once during a collection.

5 An alternative to value trailing

Suppose *setarg/3* were implemented naively in the following way:

prog7	
	<pre> setarg(N,Term,NewArg) :- young_enough(Term), !, set(N,Term,NewArg). setarg(N,Term,NewArg) :- arg(N,Term,OldArg), (set(N,Term,NewArg) ; set(N,Term,OldArg)).</pre>

where `set/3` is like `setarg/3` except that it does not trail the assignment. This implementation is not safe w.r.t. recovering heap space on backtracking, but let us forget that at the moment; without special care, it is also incorrect when a cut cuts away the disjunction in the second clause. But it serves the purpose of showing a point.

The predicate *young_enough* succeeds if its argument was created in the segment just before the current one: it is the analogue of the conditional trailing test. If it succeeds, no *trailing* is required and the assignment can be made without trailing - or in the above case, without remembering the old value in the environment of the second clause of `setarg/3`. Suppose this was the definition of `setarg/3` used in program 4. Then there is no way to collect the old argument *bar(a)* at the moment of `gc`: the term *bar(a)* is just plainly useful according to normal WAM usefulness logic. This closes the circle: the alternative and naive implementation of `setarg/3` shows the same properties as the usual one.

6 Conclusion

The example shows clearly that one bit marking is suboptimal. Also, there seems no easy and cheap fix.

Whether we should name this suboptimality a case of the WAM not being usefulness conscious, is not clear: backtrackable destructive assignment came later than the WAM. However, in this context it would mean that the WAM has no way of avoiding the value trailing in the first example program.

As far as we know, the phrase *the WAM is not usefulness conscious* was coined by Konstantinos Sagonas.

Note that the space optimization that avoids multiple value trailing within the same segment, as described in [9] does not invalidate the above argument. We have just avoided including it in the picture, as it obscures the point.

Finally, note that without destructive assignment, the issue does not arise: untrailing reinstalls a self pointer which doesn't occupy more space than already marked.

The example might look far fetched, especially since choicepoint trimming is not implemented in any popular system we know of. However, the issue of this paper was not to show that a more elaborate marking schema is needed: on the contrary, such a more elaborate schema is bad. Rather we wanted to show that the usefulness logic of Prolog - or any backtracking language - with backtrackable destructive assignment is not as well understood as one would like. And as in [4], we do not claim that we have a practical way to identify precisely the useful data.

References

- [1] H. Ait-Kaci. *Warren's Abstract Machine: A Tutorial Reconstruction*. The MIT Press, Cambridge, Massachusetts, 1991.
- [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] B. Demoen. Early reset and reference counting improve variable shunting in the WAM CW report 298, September 2000, <http://www.cs.kuleuven.ac.be/~bmd/pubs>.
- [5] 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.
- [6] 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.
- [7] B. Demoen, K. Sagonas. *Heap Garbage Collection in XSB: Practice and Experience* Proceedings of the Second Int. Workshop on Practical Aspects of Declarative Languages, Boston, Jan. 2000, pp. 93–108
- [8] Richard Jones and Rafael Lins. *Garbage Collection: Algorithms for Automatic Dynamic Memory Management* John Wiley and Sons, 1996. ISBN 0-471-94148-4.
- [9] J. Noyé. Elagage de context, retour arrière superficiel, modifications reversibles et autres: une étude approfondie de la WAM. Université de Rennes I. European Doctorate. Numero d'ordre 1275. Novembre 1994.
- [10] 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.
- [11] P. Van Roy. *Can Logic Programming Execute as Fast as Imperative Programming ?* Report 90/600, UCB/CSD, Berkeley, California 94720, Dec 1990.
- [12] D. H. D. Warren. *An Abstract Prolog Instruction Set*. Technical Report 309, SRI International, Menlo Park, U.S.A., Oct. 1983.
- [13] See <http://www.cs.sunysb.edu/~sbprolog/>.