

**Respecting the variable order
in a copying garbage collector
for the WAM.**

Bart Demoen, Phuong-Lan Nguyen

Report CW 334, March 2002



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

Respecting the variable order in a copying garbage collector for the WAM.

Bart Demoen, Phuong-Lan Nguyen

Report CW 334, March 2002

Department of Computer Science, K.U.Leuven

Abstract

One of the main objections against copying collectors for Prolog is that they do not retain the variable ordering as *defined* by the built-in predicate `compare/3`. Solutions to this problem exist already for a long time, but seem not widely known. So there is a point in presenting and evaluating them. The first solution re-uses an old idea in garbage collection and is a simple add-on to any mark© collector. Its cost is $O(V \log(V))$ where V is the number of live variables at the moment of the garbage collection. The mutator incurs no cost. The second method is just an old - but never before implemented - method which has a constant overhead on each variable-variable comparison and a $O(V)$ overhead on the garbage collector. It is also very easy to implement. The third method presented is a mixture of the previous two.

Respecting the variable order in a copying garbage collector for the WAM.

Bart Demoen

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

Phuong-Lan Nguyen

Institut de Mathematiques Appliquées
Université Catholique de l'Ouest
49000 Angers, France
nguyen@ima.uco.fr

Abstract

One of the main objections against copying collectors for Prolog is that they do not retain the variable ordering as *defined* by the built-in predicate `compare/3`. Solutions to this problem exist already for a long time, but seem not widely known. So there is a point in presenting and evaluating them. The first solution reuses an old idea in garbage collection and is a simple add-on to any `mark©` collector. Its cost is $O(V \log(V))$ where V is the number of live variables at the moment of the garbage collection. The mutator incurs no cost. The second method is just an old - but never before implemented - method which has a constant overhead on each variable-variable comparison and a $O(V)$ overhead on the garbage collector. It is also very easy to implement. The third method presented is a mixture of the previous two.

1 Introduction

We will assume working knowledge of Prolog and its implementation without further explanation. For a good introduction to Prolog see [3]; to the WAM [10], see [1]. hProlog has a predecessor [7] and is available at <http://www.cs.kuleuven.ac.be/~bmd/hProlog>.

Even though the ISO standard does not define the behaviour of comparing free variables (with the built-in `compare/3` or the `@`-family of predicates), and even though most systems do not define ¹ such behaviour, such systems feel often obliged to continue supporting code that counts on a particular behaviour. This prevents the wider acceptance of copying garbage collectors, as they typically do not preserve the address order on which comparison of free variables relies. When a copying garbage collector for the WAM was proposed in [2], the authors also described a method for dealing with this problem. The method is however not very attractive as an add-on to existing code.

¹in the ISO sense of the word

[9] is probably the first paper that mentions a garbage collection method that has complexity $O(U \log U)$ where U is the number of live cells. This method was adapted in [5] and later in [4] and it seems practical. The same idea can be used for preserving the order of cells during a mark© garbage collector and it has essentially the same complexity. The adaptation of this idea to just preserve the order of variables during a copying collection, is also easy: it is presented in Section 3 and it can easily be implemented as an add-on to an existing mark© garbage collector. However, since this method has a worst case complexity of $O(V \log V)$ with V the number of variables that are live at the moment of garbage collection, and since empirical evidence shows that V can be very large, we are not in favour of this method.

In [6] - which was partly written in reaction to [2] - a different method was proposed for solving the problem that a copying collector messes up the order of variables: we review it shortly in Section 4. Its constant overhead on each variable-variable comparison is shown in Section 8. During garbage collection, its cost is $O(LCV)$, i.e. the number of compared variables live at the moment of garbage collection. Its implementation is discussed in Section 4 and its overhead on the garbage collector of hProlog is also shown in Section 8.

Section 5 shows how the two ideas can be merged to provide a potentially better performing method.

Each method has advantages and drawbacks: in the concluding section 9 we group them systematically and indicate which is our preferred choice. Since variable-variable comparison is *a bad thing*, every method we consider must meet the following basic requirements:

- as long as variable-variable comparison is not used, the basic execution performance should not be tampered with; this excludes solutions that time stamp all variables
- supporting a reasonable view on variable-variable comparison should not require complicated and error-prone changes to the (WAM) implementation
- overhead imposed by the mechanism to deal with variable-variable comparison during garbage collection, should be small ²

There are basically two kinds of methods: those that impose no overhead at all on compare/3 (and friends) and the mutator, but which have a potentially large overhead on the collector. And those that penalize compare/3 (but not the other parts of the mutator) and to a smaller extent the collector. The beginning of Section 3 goes into more detail.

Section 2 first shows by example what is current practice and what can reasonably be expected from the presented techniques.

The experiments were performed on a Pentium III, 500MHz, 128 Mb. Timings are reported in milliseconds. We used version 1.5 of hProlog. Benchmarks using variable-variable comparison are hard to get, so we cooked up an artificial one that will show better any overhead on comparison.

²this excludes the use of a sliding collector :-)

2 What exactly are we aiming for ?

Loosely speaking, we aim at techniques by which the garbage collector does not mess up the order of variables: garbage collection is transparent to the user, so it should not alter the behaviour of a program, perhaps not even when non-standard features are used. We do not want to overshoot this aim, so we will use some examples to narrow down the aim.

Example 1 Consider the program:

```
run(X,Y) :-
  (
    X @< Y, b1, fail
  ;
    Y @< X, b2, fail
  ).
```

and the query ? – $run(-,-)$.. Is it acceptable that both $b1$ and $b2$ are executed ? Is it acceptable that neither is executed ?

According to the ISO standard, both questions deserve an affirmative answer. Slightly more contrived examples³, (but in the same spirit) show that in SICStus Prolog and Yap, sometimes both and sometimes neither of the two branches are executed⁴. In any case, if local variables are globalized at some point (maybe during comparison, or during the construction of a term containing the variable or as it happens even during writing a variable !), it is difficult to remember this globalization over backtracking. It just seems that one should not count on a consistent behaviour when backtracking over the comparison itself is involved. Moreover, it is quite easy to rewrite the body of the predicate into an if-then-else if that was the intention.

So, none of our methods will attempt to give the more intuitive meaning to the above piece of code, which is that either $b1$ or $b2$ will be executed.

Example 2 Consider the following example:

```
run(X,Y) :-
  X @< Y,
  do_things_not_involving_X_Y,
  X @> Y.
```

and the query ? – $run(-,-)$.. Can we now expect that the query fails ?

The ISO standard says one cannot expect that. However, most implementations comply with most users' expectations on this query. So, our methods should also comply with that. In terms of the WAM, this means that once two variables have been compared, and if this comparison was based on the order of the addresses of the end of the reference chains - the cells holding the UNDEFs - the order of these cells should be preserved during a collection. Note, that this requirement precludes already easy shunting as in [4].

³see the appendix

⁴but the same examples behave *better* in SICStus when the code is consulted instead of compiled

Example 3 Consider the following example:

```
run(X,Y) :-
    X @< Y,
    (
        X = a, Y = b, b1, fail
    ;
        X @> Y, b2, fail
    ).
```

and the query $? - \text{run}(-, -)$. Can we now expect that *b2* is not executed ?

This example is a bit different from examples 1 and 2, because now the comparisons $X@ < Y$ and $X@ > Y$ are performed in conjunction *after backtracking*. Again, the ISO standard does not promise anything, so perhaps we should not bother with this behaviour either. Still, implementations with a sliding collector like SICStus get this *right*. In systems with a copying collector like hProlog, it is very well possible that a garbage collection is invoked during the execution of *b1*. This could switch the order of the cells with *a* and *b*, so that after backtracking the test in the second branch also succeeds. A similar thing also applies to example 2.

One could argue that the code in example 3 can easily be rewritten to

```
run(X,Y) :-
    compare(Res,X,Y),
    (
        X = a, Y = b, b1, fail
    ;
        Res = (>), b2, fail
    ).
```

so the user can avoid this problem. So, we feel that it is reasonable to devise methods which do not comply to the intuition and find out what the performance cost is for complying. In Section 3 we will indicate clearly what it takes to preserve the order of cells which are no longer UNDEF at the moment of the garbage collection.

3 A $O(V \log V)$ method

In this section we are aiming at a solution that does not require a change to the implementation of `compare/3`. From this premise it follows that one cannot distinguish variables that have taken part in a variable-variable comparison and those that have not. It follows that the collector must preserve the address order of **all** variables. This can be achieved as follows - remember we are doing a mark© collector: during the marking, all cells with an UNDEF (self-reference) are saved in the to-space. If the to-space is as large as the from-space, it is definitely large enough to hold all the UNDEF cells. After the marking

phase is finished, the UNDEF cells are sorted and then used in the copying phase before any other root pointer for forwarding in the order from small to large UNDEF address.

Let V be the number of UNDEF cells live at the moment of garbage collection. It is clear that the cost of the proposed method is dominated by the $O(V \log(V))$ complexity of sorting. It thus is informative to know V during some typical benchmarks or applications. Table 1 contains for a series of benchmarks V and the number of garbage collections that occurred during a run of the benchmark, starting with 4 Mb of heap⁵ space. V is reported in the form $a - b$ with a the minimum number of variables during a collection and b the maximum, or when there are very few collections (see chess for instance) as $a/b/c$ with a , b and c the number of UNDEFs at each collection.

benchmark	# of UNDEFs	# gc's
updown	4-5	17
mqueens	4-11	186
browsegc	5	1
dnamatch	6-6	21
tspgc	6-10	11
emul	11-17	101
barnes_hut	12-22	26
boyergc	10-26	31
xsbcamp (apc)	223-1922	6
hPrologcomp (xsbcamp)	4480/6298	2
hPrologcomp (apc)	6459-31688	14
chess	7/325493/809257	3

Table 1: Counting the number of UNDEF cells in the heap during garbage collection

The benchmarks have been split up in three categories: the first one has benchmarks that are commonly used for measuring the performance of the mutator or the collector; they show a small number of UNDEFs which does not raise second thoughts on the $V \log(V)$ component. The second category consists of Prolog compilers: xsbcamp is the XSB compiler (adapted slightly) and it was given the Aquarius compiler as input. hPrologcomp is the compiler in hProlog and it was run on xsbcamp and on the Aquarius compiler: V can be much higher here. The last category consists of just one benchmark chess, which is admittedly badly written, but nevertheless shows that user programs can have a huge number of active free variables. Even though most of these result from the allocation strategy of hProlog (no variables in the local stack), it seems that this method will sooner or later bite the user, because of its high complexity, even if her program performs only one variable-variable comparison. We therefore dismiss it and have not even bothered to implement it completely. Still, it is a useful method if accompanied by the possibility for the user to declare in some way that during the execution of her program, variable ordering is unimportant - which should perhaps also result in a warning when a variable-variable comparison is attempted.

We will later refer to this method as *preserve_all*. As described above, it would not necessarily give the intuitive result for Example 3. The following addition is required to the

⁵also named global stack

trail traversal during the marking phase:

- if the trail entry points to a non-marked cell, perform early reset as usual (and remove the trail entry), or if that is not possible (because of generational collection for instance)
- add the address of the pointed to cell to the set of cells that need their order preserved (and perform all other usual actions on this cell of course)

Variants of this adaptation to the trail treatment will pop up in later sections. Their effect is similar: with it, Example 3 works as expected, otherwise all bets are off. We will distinguish variants that have it or not by a suffix: *preserve_all(undef)* as opposed to *preserve_all(allvar)* up to now.

4 An old idea regenerated

In [6] we have already suggested the following idea: when two variables X and Y are involved in a comparison, then create a new data structure $\text{foo}(\text{NewX}, \langle \text{NX} \rangle)$ on the top of the heap and bind X to NewX; do a similar thing for Y. For NX and NY choose two new numbers: these are the stamps on which the comparison of X any Y are based. When X (or Y) is involved in a new variable-variable comparison, check whether its dereferenced value points one cell further than a $\text{foo}/2$ structure and if so, pick up NX, otherwise do the above construction. This shown in Figure 1: X is a variable that was previously involved in a variable-variable comparison, while Y is not - see Figure 1(a). Then the goal $X@ < Y$ is executed, resulting in Figure 1(b).

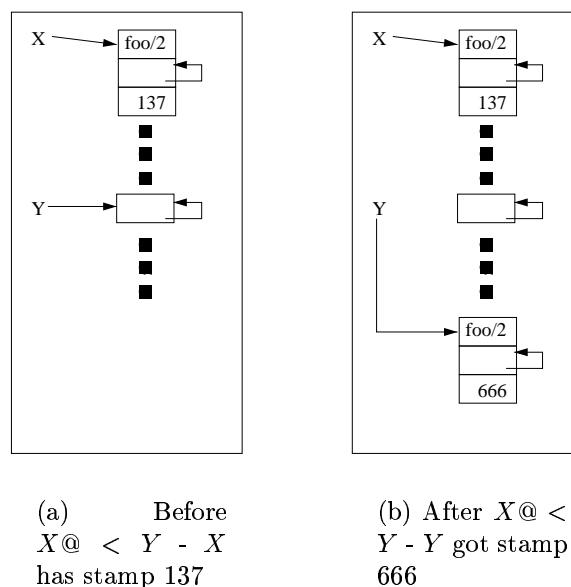


Figure 1: Illustration of stamping

Since comparison of variables is now based on the stamps, their address order can be changed freely by the collector. The garbage collector does never find direct pointers to the `foo/2` structure, but that is not a problem: when an UNDEF is found on the heap during marking, check for the `foo/2` structure just above it. We name this method *stamp_compared* and of course it comes in variants *stamp_compared(undef)* and *stamp_compared(allvar)* with obvious modifications. The trail adaptation for the `allvar` variant, can be implemented differently here: do not stamp during treating the trail, but any time a cell is reached directly by a REF, check whether it is preceded by the `foo/2` functor. This seems more economical than during the trail treatment, and is actually how it was implemented.

The characteristics of *stamp_compared* are:

- no variable-variable comparison is used, its overhead is only during the marking phase of the mark© collector; Section 8 shows how much
- it can be used even in the absence of a marking phase - see [8] for more details on copying without marking
- one needs a collector for the stamps because they can overflow, or one must keep a backtrackable stamp counter (a new field in the choice point or as a backtrackable global variable); we would prefer the former: collection of the stamps can be performed during an actual garbage collection
- comparison of terms can now put things on the heap, and potentially overflow it; the important thing to note here is that during a comparison of two terms, the stamping can happen only twice, because the result of a comparison is known when two different variables are compared - of course stamping is useless when a variable is compared with itself
- make `foo/2` impossible for the user to type in :-)
- don't do easy shunting on the NEWX
- if marking is used, the to-space need not be allocated before the marking is finished and can be smaller than the from-space

Instead of a `foo/2` structure, one could push a recognizable atom and the stamp behind the UNDEF, or any other variation on the same theme depending on what suites the implementation best.

5 Merging two ideas

We can use a combination of the ideas in Sections 4 and 3: during a variable-variable comparison in which X is involved, the variable X is bound to a structure `foo(NewX)`. During marking, only the UNDEF cells preceded by the `foo/1` functor are saved and sorted before the forwarding begins. In principle the number of such cells can be $O(H)$ with H the heap size, but no doubt is it small for almost any application. This idea has the advantage that it

uses less space than the ideas it inherits from. There is a need for marking before copying. Programs not using variable-variable comparison are not affected during normal execution and only the marking phase of the collector is slightly affected, in essentially the same way as with the `foo/2` structure. We refer to it as *merge*.

6 A different approach based on a similar idea

The idea here is to record every variable that is involved in a variable-variable comparison. If the Prolog system has a value trail, this can be done in a balanced ⁶ tree on the heap. After the marking phase (which does not consider this tree as part of the root set: the tree has only weak pointers to the compared variables), the tree is traversed and only the elements that point to a marked cell are retained and subsequently used for starting the copying process. This method also retains the order of once-compared variables which are bound at the moment of the collection even over backtracking. Here is an implementation for `compare/3` that maintains just a list using the backtrackable global variables in hProlog:

```
compare(Res,X,Y) :-
    internal_compare(Res,X,Y,Var1,Var2),
    (var(Var1) -> list_insert(Var1) ; true),
    (var(Var2) -> list_insert(Var2) ; true).

list_insert(Var) :-
    b_getvar('$comparedvars',L),
    (var_member(Var,L) ->
        true
    ;
        b_setvar('$comparedvars',[Var|L])
    ).
```

The idea is that `internal_compare/5` also unifies its last two arguments with any variables that were compared; as noted before, there can be at most two of those.

This method was not implemented and experimented with, but it is clear that the cost of maintaining the tree of compared variables (or any other data structure with the desired properties), might be prohibitive. On the other hand, it is the only method we know of that puts no overhead at on programs that use no comparison at all ⁷. When needed, we refer to this method as *record_compared*. It is similar to what was suggested in [2], but easier to implement and slower.

⁶one probably wants to avoid duplicates in this data structure

⁷except for one test in the garbage collector for deciding that the compared variables data structure is empty

7 Overview of characteristics

Table 2 gives a short overview of the characteristics of the different methods. V means the number of variables (cells with UNDEF), CV the number of compared variables and U the size of the useful data. Note that V and CV can be $O(U)$.

method	overhead on compare/3	overhead on gc	requires marking	space cost
preserve_all(undef)	0	$O(V \log V)$	yes	to-space
preserve_all(allvar)	0	$O(V \log V)$	yes	to-space
stamp_compared(undef)	$O(1)$	$O(V)$	no	3 cells * CV
stamp_compared(allvar)	$O(1)$	$O(U)$	no	3 cells * CV
merge(undef)	$O(1)$	$O(V)$	yes	2 cells * CV + to-space
merge(allvar)	$O(1)$	$O(U)$	yes	2 cells * CV + to-space
record_compared(undef)	$O(CV \log CV)$	$O(CV)$	yes	$O(CV)$
record_compared(allvar)	$O(CV \log CV)$	$O(CV)$	yes	$O(CV)$

Table 2: Characteristics of the methods

What remains to be done, is the performance evaluation: see Section 8.

8 Performance

We must decide what is worth implementing and what is worth measuring ... The $O(V \log V)$ method of Section 3 is clearly not interesting. Also, it is clear what its overhead is once one knows how many variables there are. The overhead of the foo/2 and foo/1 methods on variable-variable comparisons is worth measuring though, as well as its overhead on the collector itself. The former measurement is interesting on programs using variable-variable comparisons. But such programs are rare, so we will rely on one artificial benchmark reproduced later. The other measurement will happen exclusively on programs not using variable-variable comparisons, as one is interested mostly in showing that programs not using such comparisons do not suffer unduly from this feature.

The artificial program just constructs a long list of variables and compares all of them with each other:

```

run :-
    N = 3000, M = 10,
    mklist(N, LN),
    mklist(M, LM),
    itcompareall(LM, LN).

docompare(Res, X, Y) :-
    compare(Res, X, Y).

compareall([]).
compareall([], _).
compareall([_|R], L) :-
    (
        compareall(L), fail
    );
    itcompareall(R, L).

```

```

compareall([X|R]) :-
    compareall(R,X),
    compareall(R).
compareall([X|R],Y) :-
    docompare(_,X,Y),
    compareall(R,Y).

```

Both hProlog and SICStus are given enough heap so that no garbage collection takes place during the benchmark. Also, a suitable dummy loop was subtracted from the run above.

	hProlog	SICStus
original implementation	4330	7850
with foo/2	6600	-
with foo/1	6590	-

Table 3: Overhead of two methods on variable-variable comparison

The figure for SICStus is given merely to show that the compare/3 predicate in hProlog is not implemented in a slow way. Compare/3 is an inlined predicate in both systems. The overhead is clear, but stays within reasonable limits: if comparing variables makes up 1% of all execution cost in a program, one should expect a global slowdown of about 0.5%, which seems reasonable to us.

Since we interested in the overhead of the stamp_compared methods and the merge-methods on programs that do not use variable-variable comparison, and since on such programs the overhead is the same, we have implemented the adaptations to the garbage collector for the stamp-methods only. The timings are in milliseconds and only garbage collection times are shown.

benchmark	original	stamp(allvar)	stamp(undef)
updown	870	860	870
mqueens	117980	115830	116920
browsegc	140	150	150
dnamatch	210	240	240
tspgc	610	660	620
emul	2560	2550	2550
barnes_hut	60	70	60
boyergc	13610	13590	13850
xsbcomp (apc)	570	560	580
hPrologcomp (xsbcomp)	90	90	90
hPrologcomp (apc)	850	840	840
chess	5920	5840	5820

Table 4: Overhead during garbage collection

These figures seem to indicate that the overhead is within noise: probably the change in instruction cache behaviour - which can go both ways - is more important than the extra executed instructions.

9 Conclusion

The presented techniques are simple enough to be added to an existing system. The techniques with the `foo/*` wrappers have reasonable overhead on comparison and on the collector when variable-variable comparison is not used. We prefer the stamp method, because it can be combined with a non-marking copying collector. We believe that at least one objection against adopting copying garbage collectors within the WAM is not valid at all.

Acknowledgements

We thank Henk Vandecasteele (PharmaDM) for the use of the `ilProlog` compiler in `hProlog`.

References

- [1] H. Ait-Kaci. The WAM: a (real) tutorial. Technical Report 5, DEC Paris Research Report, 1990 See also: <http://www.isg.sfu.ca/~hak/documents/wam.html>.
- [2] J. Beve myr 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 Lecture Notes in Computer Science, pages 88–101. Springer-Verlag, Sept. 1994.
- [3] I. Bratko. *Prolog Programming for Artificial Intelligence*. Addison-Wesley, 1986.
- [4] L. F. Castro and V. S. Costa. Understanding memory management in prolog systems. In P. Codognet, editor, *Proceedings of the 17th International Conference on Logic Programming, ICLP'2001*, number 2237 in Lecture Notes in Computer Science, pages 11–26. Springer-Verlag, jul 2001.
- [5] Y. C. Chung, S.-M. Moon, K. Ebcio glu, and D. Sahlin. Reducing sweep time for a nearly empty heap. In *Conference Record of the 27th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 378–389. ACM Press, Jan. 2000.
- [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, Feb. 1996.
- [7] B. Demoen and P.-L. Nguyen. So many WAM variations, so little time. In J. Lloyd, V. Dahl, U. Furbach, M. Kerber, K.-K. Lau, C. Palamidessi, L. M. Pereira, Y. Sagiv, and P. J. Stuckey, editors, *Computational Logic - CL2000, First International Conference, London, UK, July 2000, Proceedings*, volume 1861 of *Lecture Notes in Artificial Intelligence*, pages 1240–1254. ALP, Springer, 2000.
- [8] B. Demoen, P.-L. Nguyen, and R. Vandeginste. `Copy_term/2` and garbage collection. Report CW 329, Department of Computer Science, K.U.Leuven, Leuven, Belgium, Jan. 2002.
- [9] D. Sahlin. Making garbage collection independent of the amount of garbage. Technical Report R87008, SICS, 1987.
- [10] D. H. D. Warren. An abstract Prolog instruction set. Technical Report 309, SRI, 1983.

Appendix

The queries in the examples are ? - *run_i*.

```
run1 :-
    f(X,Y),
    t(X,Y),
    f(X,Y).

run2 :-
    f(X,Y),
    s(X,Y),
    f(X,Y).

t(X,Y) :-
    (
        X @> Y, write(pok),nl,fail
    ;
        h(f(Y)),
        h(f(X)),
        X @< Y,
        write(asd),nl,fail
    ).

s(X,Y) :-
    (
        X @< Y, write(pok),nl,fail
    ;
        h(f(Y)),
        h(f(X)),
        X @> Y,
        write(asd),nl,fail
    ).

f(_,_).

h(_).
```

The following is similar but involves no backtracking.

```
run3 :-
    b(X,Y),
    compare(Res1,X,Y),
    b(f(X),f(Y)),
    compare(Res2,X,Y),
    b(X,Y),
    write(Res1 = Res2),nl,fail.

run4 :-
    b(X,Y),
    compare(Res1,X,Y),
    b(f(Y),f(X)),
    compare(Res2,X,Y),
    b(X,Y),
    write(Res1 = Res2),nl,fail.

b(_,_).
```