Copy_term/2 and garbage collection

Bart Demoen, Phuong-Lan Nguyen, Ruben Vandeginste

Report CW 329, January 2002

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

Copy_term/2 and garbage collection

Bart Demoen, Phuong-Lan Nguyen, Ruben Vandeginste

Report CW 329, January 2002

Department of Computer Science, K.U.Leuven

Abstract

Copying terms has a central role in every Prolog system as some predicates must use it. Apart from copy_term/2 itself, there is at least findall/3 and throw/1 that (can) use it and in many implementations, also some form of globale variables uses an underlying copying routine. On the other hand, most Prolog systems use a garbage collector that moves terms on the heap, either by a copying or a sliding algorithm. Copy_term/2 and garbage collection are clearly related: *findall/3* is even advertised sometimes as a substitute for true garbage collection. Starting from a simple experiment that establishes the relative performance of copy_term/2 and garbage collection , it is described how the garbage collection algorithms can be used for (non-destructively) copying terms, and how the essentials of copy_term can be used to implement a series of garbage collectors. Experimental results show good speedups in the latter case. Information on the cache behaviour of all variants is provided and partly explains the observed performance.

Copy_term/2 and garbage collection

Bart Demoen * Phuong-Lan Nguyen † Ruben Vandeginste ‡

January 31, 2002

Abstract

Copying terms has a central role in every Prolog system as some predicates must use it. Apart from copy_term/2 itself, there is at least findall/3 and throw/1 that (can) use it and in many implementations, also some form of globale variables uses an underlying copying routine. On the other hand, most Prolog systems use a garbage collector that moves terms on the heap, either by a copying or a sliding algorithm. Copy_term/2 and garbage collection are clearly related: *findall/3* is even advertised sometimes as a substitute for true garbage collection. Starting from a simple experiment that establishes the relative performance of copy_term/2 and garbage collection , it is described how the garbage collection algorithms can be used for (non-destructively) copying terms, and how the essentials of copy_term can be used to implement a series of garbage collectors. Experimental results show good speedups in the latter case. Information on the cache behaviour of all variants is provided and partly explains the observed performance.

1 Introduction

We assume knowledge of Prolog and its implementation. For a good introduction to the WAM, see [1, 18]; the SICStus implementation is described in [5]. About Yap one can find implementation details in [9]. hProlog is a successor of [12] and is available from the author. hProlog is meant to become a back end to HAL [11]. The ultimate documentation of these systems - their source code - is accessible, either for free or at a small cost for academia.

The start of this research was two-fold: firstly the realisation that the hProlog system contained several term *copying* routines, and secondly the trade-off between copying a term and collecting it in an alternative implementation of the built-in predicate *findall/3*, based on hProlog's global variables:

- There are term copying routines in hProlog for *copy_term/2*, for *findall/3*, for global variables, for stack expansion and for garbage collection. They all have different characteristics: copying can be form heap to heap, from heap to some external contiguous

*Dept. of Computer Science, K.U. Leuven, Belgium, bmd@cs.kuleuven.ac.be
†Inst. de Mathèmatiques Appliquées, UCO, Angers, France, nguyen@ima.uco.fr
‡Dept. of Computer Science, K.U. Leuven, Belgium, ruben@cs.kuleuven.ac.be

or discontiguous area, back from these areas to the heap. Also, these copying routines need to satisfy different constraints: we'll come back to that point in section 4. At some point, we wanted more uniformity in all these copying routines.

- As explained in more detail in section 4, the alternative implementation of findall/3 trades copying a term (by means of a copy_term/2 like primitive) for a collection of the term [1]

The latter observation prompted a small experiment in which the performance of a garbage collector was compared with copy_term/2. Section 4 gives more details.

The usual implementation of copy_term/2 is by a recursive C-function that visits the term in a depth-first manner. In hProlog - as in many other systems - this is improved on by explicitly managing a stack of nodes still to visit, instead of relying solely on recursion in C. The first idea was to change that implementation of copy_term/2 to use an algorithm similar to what typical Prolog garbage collectors use: they are usually based on marking the live data and the algorithm of Morris [14] as in SICStus Prolog or Cheney [7] as in hProlog. This is investigated in Sections 5 and 6 and turned out to be not successful. In that context we have implemented an optimization we name *forward optimization*, we was also tested later in variants of the garbage collectors.

Finally, we set out to re-implement the garbage collector and base it on the same algorithm as copy_term/2. The relation between copying terms and garbage collection is actually quite old: findall/3 (which uses a form of copying) has sometimes been advocated as *the poor man's garbage collection* because often a real garbage collector can be avoided using findall/3. Basing a collector on the copy_term/2 routine - or a small variant of it - means that no marking prior to the copying will be performed. However, [4] explains the *need* for marking before copying: cells that belong to a structured term but are copied before the term is copied, are copied more than once and it is in this way possible that the collected heap occupies more space than the original heap. Still, there are at least two systems that use copying without prior marking: both [13] and [19] report on an implementation which avoids the multiple copying of the same cell by postponing the copy of a cell that looks dangerous. They both report that the set of postponed cells is small in their benchmarks. However, [13] collects only small portions of the heap and [19] collects only the last segment. Neither of them have tried this postponing schema for major collections. On top of that, postponing of dangerous cells in the context of hProlog is not an option as explained in Section 7. This section describes three other ways to deal with dangerous cells while using the copy_term/2 approach to garbage collection. All three were implemented in hProlog and experimental results are compared with the existing garbage collector in Section 8.

Section 9 shows cache performance results which can explain the performance results in Section 7.

Since this research started partly because of the need for hProlog to support non-backtrackable global variables - as a back end to HAL [11] - Section 2 shortly describes the implementation of such global variables in hProlog.

[1] we abuse terminology a bit; by *collecting a term* we actually mean *keeping it alive during a garbage collection* - this usually involves copying it in some way

The experimental evaluation was performed on a Pentium III 866MHz with 256Mb RAM. Timings are given in seconds, space measurements in bytes or cells. We used SICStus Prolog 3.8.6 and Yap 4.3.20 to show in Section 3 that the original collector in hProlog has already a decent quality which means that reported improvements are relevant.

The Appendix contains more information on the benchmarks themselves and on some issues in benchmarking garbage collectors.

2 hProlog implementation of garbage collection and global variables

The garbage collector in hProlog is a quite straightforward implementation of [4]. Perhaps it is worth mentioning the use of the mark bits for other purposes than marking itself: first of all, we have chosen to keep the mark bits separately from the heap cells. In this way, it is possible to inspect mark bits without fetching heap cells, and we believe that this is beneficial. Also, we do not want the mark bit(s) to restrict the address range of the heap. Secondly, we have chosen to always keep the array for the mark bits allocated, instead of allocating it at the moment of garbage collection: this has several advantages. When it is allocated only when needed, one must make sure the mark bits are all zero, which can incur an extra cost. Also, this mark array can be used for other purposes as well, for instance for implementing a fast *term_size/2* function. Next, we have to admit that we allocated one byte for each mark bit: we never got down to efficiently implementing packing the bits - which is also advantageous for other reasons than space. So, we have 8 bits to play with, only one of which is in use for traditional marking: we use one more bit that indicates whether some cell is to be considered for relocation or not; indeed, a cell containing an atom, an integer or a functor, needs no relocation; neither does our representation of doubles. This bit is set during marking. Later, when the cell is forwarded, a corresponding bit is set in the position of the final destination of that cell, so that during scanning, a simple test avoids relocation.

During marking, we use also a bit for detecting multiple trailings of the same heap cell: since hProlog allocates free variables on the heap only, trail entries always point to the heap.

We have also used a bit in the mark array for an optimization named the *forward optimization* described in Section 6.

Global variables in other systems (the blackboard for instance) are usually kept outside of the Prolog heap. This involves copying each time the value is retrieved. In hProlog we have decided for a radical approach: all Prolog terms will reside always in the heap. This opens possibilities for avoiding to copy the value whenever it is used. This will probably be of major importance once hProlog is used as a back end to HAL. The only problem with keeping such terms on the heap is that they must be protected against backtracking. It turns out that this is quite simple and costs very little - the mechanism is similar to the one used in XSB ([15]) to freeze consumers. We simply introduce a new machine register named *heap freeze* register and name it FH. It is initially equal to the start of the heap. Whenever a term is made global, FH is set to the current H (top of heap) and the H field in the top choice point is set to H as well. After discarding one or more choice points (on cut or trust) and before global registers are reset, the following code is executed:

```
if (FH > B[H]) B[H] = FH
```

On garbage collection FH must be appropriately set, but no other dealings with it are necessary. In particular, FH does not occupy an extra slot in the choice points.

Keeping the global variables in the heap and freezing the heap, makes it more urgent to implement a generational garbage collector for hProlog.

3 A short comparison between the garbage collector in hProlog and in other systems

It is difficult to compare correctly garbage collectors which have different policies or use a different basic algorithm. So this section should not be taken as an absolute assessment of the relative qualities of the collectors in hProlog, SICStus Prolog and Yap. It merely serves as an indication that the first garbage collector of hProlog is not too far off other systems and that the improvement we are reporting about later, is meaningful.

The collector in hProlog is a rather straightforward implementation of [4] and thus uses a Cheney algorithm [7] with a marking phase. It is non-generational. The collector in SICStus Prolog is an implementation of [3]: it uses a mark&slide algorithm based on Morris [14]. It also implements variable shunting. More importantly, it is 2-generational - but when the collector is called explicitly (with the built-in garbage_collect/0), it performs always a major collection. Yap has a similar implementation to SICStus, but it uses *easy shunting*: see [6].

In the benchmarks, we have tried to exclude the effect of heap expansion, i.e. we have given enough heap so that no expansions (should) occur. Also, we have given similar amounts of heap space to all systems for a particular benchmark: this is not always possible, as SICStus rounds heap sizes to the nearest size it likes, and Yap seems to use the size specified with its -s option for the total of global and local stack (which has both environments and choice points). So we decided to use the following procedure to decide on how much space to give each system:

1. SICStus Prolog gets enough global stack space so that no expansion occurs

2. Yap gets enough space so that it has exactly the same amount of garbage collections; this is reasonable because SICStus Prolog and Yap both follow WAM in their allocation of free variables

3. finally, hProlog gets exactly the same amount of heap as SICStus: hProlog indeed allows to specify up to a cell (4 bytes) its initial heap size

This can result in a higher number of garbage collections for hProlog than for SICStus Prolog, because hProlog allocates all free variables in the heap. But the generational aspect (of SICStus Prolog) can in general lead to more collections as well.

The appendix describes the benchmarks in a bit more detail.

The benchmarks in the table have been ordered in such a way that the lower the benchmark in the table, the more we think it benefits from generational garbage collection. This is most pronounced for the mqueens benchmark.

	hProlog	SICStus	Yap	
emul	1.85/24.21	2.04/41.40	0.74/24.66	gc time/runtime
	75	42	42	# garbage collections
	3098074456	223179672	257424496	# bytes collected
tspgc	0.02/24.10	0.33/54.62	0.03/44.33	gc time/runtime
	8	6	6	# garbage collections
	42255172	31687108	29191880	# bytes collected
dnamatchgc	0.14/3.46	0.82/5.94	0.24/4.46	gc time/runtime
	15	15	15	# garbage collections
	78565940	76766780	78379180	# bytes collected
chess	3.62/12.83	3.44/23.14	2.36/12.96	gc time/runtime
	3	3	3	# garbage collections
	14646248	6075380	10188472	# bytes collected
boyergc	4.19/14.04	2.85/19.54	3.65/12.09	gc time/runtime
	5	5	5	# garbage collections
	1033713000	133572332	80463556	# bytes collected
serialgc	3.80/11.02	4.10/18.25	4.76/13.97	gc time/runtime
	3	3	3	# garbage collections
	1040245812	140274784	112856100	# bytes collected
mqueens	39.71/97.50	18.12/141.95	42.42/86.83	gc time/runtime
	41	38	38	# garbage collections
	15013225540	1399909612	1388072412	# bytes collected

Table 1: Comparing the mark&Cheney copying collector of hProlog with others

4 Comparing copying and collecting a term

Using the global variables as described in Section 2, we re-implemented findall/3: we name this *new_findall* as opposed to *old_findall*. In a goal like $?-findall(Template, Goal, List).$, old_findall copies the Template solutions twice: once to a memory place outside the heap, and once back into it. New_findall copies the Template solutions only once - from heap to heap - and then freezes the heap. This looks like a pure gain, but this is not so: freezing the heap means that the Template now has a higher chance to be subject to garbage collection while new solutions to the Goal are being computed. In fact, with new_findall a Template can be subject to garbage collection an arbitrary number of times before the Goal is finished, while during old_findall, garbage collection is not triggered at all. With a reasonable generational collector, this number is limited to one. So, after all with new_findall have traded in one sure copy, for one maybe garbage collection of the Template.

It follows that the relative performance of copying and collecting a term is worth looking into. To this effect, we performed a very simple experiment: a reasonable big term is created (in hProlog it is close to 1 Mb heap cells) and we measure the time to construct it, to collect it and to copy it. Since the fact that the term is constructed depth first or breadth first can influence the timings, we have done the measurements for both. The term is a 12 deep tree

of the form f(A,B,C) where A, B and C are of the same form. The leaves of the term are distinct free variables. The results are in table 2: the timing is the result of performing each operation 10 times.

		hProlog	SICStus	Yap
depth-first	construct	0.34	0.39	0.40
	collect	1.91	2.09	1.74
	copy	1.14	2.53	0.75
breadth-first	construct	0.39	0.71	0.72
	collect	2.10	2.29	1.96
	copy	1.18	2.40	0.72

Table 2: Comparing construction of a term, collecting it and copying it

All systems got enough initial heap so that garbage collection was only triggered explicitly.

The *construct* and *copy* row show the time for constructing and copying the term: there was no garbage collection during this. The *collect* row shows only the garbage collection time: the rest of the runtime is 0 or negligible anyway. Garbage collection is explicitly called and at that point the root set is small: there are no choice points and there is just one environment - except the ones that result from the boot/toplevel procedure of the system. hProlog and SICStus preserve sharing between subterms while copying a term. Yap does not [2] SICStus also avoids copying ground subterms, which in this case leads only to overhead and no savings.

Even though there are faster ways to construct the term we used for the test it is clear already from these figures that constructing a term can be much faster than copying it. The interesting point is however the difference in speed between collecting a term and copying it: while copying is faster than collection in hProlog and Yap, it is slower in SICStus. Copying and collection have of course quite different characteristics, but one can wonder whether it would be sensible to base the collector in hProlog or Yap on copying, and the copy routine in SICStus on the collector !

It is worth stating at this point explicitly the differences between copying a term with copy_term/2 and collecting it. Table 3 shows these differences; they hold for compacting collectors (non-moving collectors have even more different characteristics):

	collecting a term	copying a term
shorten chains	conditional	preferably
preserve original	optional	mandatory
preserve sharing	usually	preferably
copies ground terms	mandatory	optional

Table 3: Differences between copying and collecting a term

The entries are self-explaining, except perhaps the one with *usually*: all collectors we are aware off preserve sharing, and this is probably considered mandatory. However, a

[2] Yap has an installation option for preserving sharing, but its default is off

technique like hash-consing during garbage collection (see for instance [2]) actually improve sharing during garbage collection. New insights also indicate that *changing* the sharing (i.e. destroying existing sharing and introducing a new one) can be beneficial for garbage collection.

It will turn out that the obligation to preserve the original term, is the major obstacle for garbage collection algorithms to be applied efficiently for copying a term.

5 Copy_term/2 based on a sliding collector

Table 2 indicates that the collector in SICStus is faster than the copy routine. SICStus' collector [3] is based on Morris' algorithm [14]; it performs variable shunting [17] and has other overhead that just copying the term does not need to incur, so it is tempting to try to use a variation of the garbage collector for copying a term. Adapting it to copying a term goes as follows: use a marking phase which starts from the root of the term to copy. At the end of this marking phase, the size of the term is known, so it can now be checked easily that there is enough spare space on the heap. Now we enter the sliding phase: Morris starts with an upward phase, from the top of the heap. Here is the first obstacle for using the plain algorithm: if the upward phase really starts at the top of the heap, it will have a worst case O(sizeofheap) cost. So, we want to start at the highest address any cell in the term has. This requires during marking keeping a high water mark. But that is not enough: if the term is dispersed over the heap, the worst case complexity will remain the same. So one has to resort to a technique like described in [8] or [16]. This will make copying in the worst case O(NlogN) where N is the number of cells in the term. That seems worse than it really is: avoiding to copy ground subterms has the same complexity in the implementation of SICStus.

There are still some technicalities to clarify: Morris' algorithm uses a *to* and a *from* pointer. When we use Morris' algorithm for copying a term, the *to* pointer must always point into the not-yet used portion of the heap where the copied term eventually will show up. Normally, the two phases together move the term, while adapting its internal pointers. We have to make sure that the original term is perserved. This could be done as follows:

1. during the upward phase, relocate upward pointers
2. during the downward phase, relocate downward pointers, but do not move any data
3. push all the changes made by relocation on a trail stack, in the spirit of the usual value trail
4. perform one more pass over the original term (which has all its internal pointers now relocated) and copy it verbatim to the new region
5. untrail all the changes made to the original term

Finally, there is the issue of short-circuiting reference chains: the usual copying of terms makes sure that reference chains are at most of length one. One way to achieve this, it during

the marking short-circuit the reference chains and also value trail these changes. Avoiding to copy ground subterms can also be incorporated in the marking phase.

All in all, it seems that there is little chance to make this work as fast for copying a single term, as by the current copy_term implementation, while retaining its properties.

6 Copy_term/2 based on a copying collector

Table 2 also indicates that the hProlog collector is faster than the copy routine in SICStus. So we set out to try to use the basic algorithm of the hProlog collector for copying. We expect both some gain and some loss in doing so: in comparison with the collector algorithm, for copying we have some opportunities to specialize it. We do not need to allocate a temporary heap (see [10] for more details) neither do we need to copy back after the collection. Also, the *distance* over which copying takes place, is zero (this is related to the lack of need for copying back). Marking becomes less involved than during garbage collection since the root set is just one cell: the top entry point of the term to copy. Moreover, apart from adapting the mark&Cheney-copy implementation to do plain copying, we have also implemented plain Cheney copying, i.e. without marking, but with shortening of chains. So quite optimistically, we implemented these variants for copy_term/2. The outcome of the experiment is in Table 4.

hProlog	depth-first	breadth-first
construct	0.34	0.39
collect	1.91	2.10
depth first rec copy	1.14	1.18
mark&Cheney based copy	2.69	2.84
plain Cheney based copy	2.21	1.87

Table 4: Basing a copy_term/2 implementation on Cheney

The result is rather disappointing: copying a term based on mark&Cheney even takes more time than collecting it. And Cheney copying it without marking is in the same order as collecting. The difference in root set cannot be the reason, as during the benchmark the garbage collections are always called when the stacks are very small. The marking costs the same in during collecting and during the mark&Cheney based copy. The untrailing of the forwarding pointers at the end of the copy is expensive and need not be done by the collector, because it can destroy the original. Therefore it might pay off to lower the number of trailings that are performed. Cheney's algorithm leaves a forwarding pointer for each forwarded cell. However, this is not always necessary. During marking it can be decided for each cell whether it will need a forwarding pointer or not: only cells that are referenced directly from another cell need it, because they will be referenced while scan catches up with next. The marking phase can be made to set an extra bit indicating which cells need a forwarding pointer. We name this the *forward optimization*. The effect of the forward optimization on the artificial copy benchmark can be seen in Table 5: the savings is mostly in the untrailing phase and a bet less in the marking phase which now needs to trail less.

The timings represent the total of 10 runs. For comparison, we have also given the timings for Cheney without marking.

		marking	copying (incl trailing)	untrailing	total
depth-first	mark&Cheney	0.47	1.57	0.65	2.69
	mark&Cheney + forward opt	0.48	1.46	0.49	2.43
	plain Cheney	(none)	1.59	0.62	2.21
breadth-first	mark&Cheney	0.50	1.69	0.65	2.84
	mark&Cheney + forward opt	0.51	1.64	0.50	2.65
	plain Cheney	(none)	1.33	0.54	1.87

Table 5: Copying based on Cheney with and without the *forward optimization*

It seems clear that the savings of plain Cheney as compared to mark&Cheney come from excluding the marking phase. Note that the plain Cheney copy routine corresponds to the *dangerous* copying method used in Section 7 for garbage collection.

The same forward optimization can be also used during garbage collection : we adapted the mark&Cheney collector of hProlog and on the set of benchmarks we get the results as in Table 6 where only the garbage collection times are given.

	hProlog	forward opt
emul	24.21	24.18
tspgc	24.10	24.07
dnamatchgc	3.46	3.44
chess	12.83	12.88
boyergc	14.04	13.99
serialgc	11.02	10.97
mqueens	97.50	97.51

Table 6: Assessing the effectiveness of the *forward optimization* during garbage collection

Clearly, for a garbage collector, the savings are smaller and often even negative.

As a side note: shortening the reference chains is easy in Cheney; one must merely take care to use a special purpose dereferencing function which also *stops* when a forwarding pointer is found. Avoiding to copy ground subterms, can be solved in a similar way as for the sliding algorithm during the marking phase. It seems not to fit at all in the plain Cheney algorithm.

As in Section 5, the outcome is quite disappointing and here even shown with some hard figures: the fact that we actually set out this piece of work trying to show that Cheney's algorithm can be used for copy_term without speed loss, only makes it worse. The fact that we used only one benchmark - and an artificial one for that - does not change our conclusion: we should give up on this and put our hopes on using the recursive copying algorithm from copy_term/2 in the garbage collector.

7 Garbage collection based on copy_term/2.

In [4] it is argued why it is important to mark before (garbage collecting by) copying. However, at least two garbage collectors for Prolog systems are based on copying without marking: the problem of internal cells that are copied twice is solved by postponing to copy cells that look dangerous. This happens in [19, 13]. Postponing during copying is not a good option for hProlog: it allocates all free variables on the heap, which means that one expects that most root cells point to the heap and will look dangerous.

Another approach advocated by Paul Tarau - but not implemented as for as we know - is to use *dangerous* copying. i.e. not mark and take the risk of copying twice the same cell - of course forwarding pointers are set and taken into account. It is expected that double copying will not happen often in practice. The fact that both [19] and [13] report that the stack of postponed cells is small, seems to support this claim. However, [13] collects always very small portions of the heap, the measurements of [19] were done for small benchmarks only and neither of them performs global collections. Dangerous copying is even more dangerous than one realises at first: it turns out that without extra precautions, some cells can even be copied more than twice, actually in the order of the root set. Indeed, consider the following piece of code:

```
run :-
        f([X|Y],Z),
        A1 = Z, A2 = Z, ... , An = Z,
        copy_in_order(Y,X,Z,A1,A2,...,An).

f(A,A).
```

The meaning of copy_in_order/* is that there is a garbage collection which visits the root set in the order of the arguments of the predicate. Assuming that the variables reside in the environment, there are before copying only 2 cells on the heap and both contain a self reference. This can be seen at the left of Figure 1: the **L** in some cells, indicates that the cell contains a list-tagged pointer.

Dangerous copying does not postpone dangerously looking cells, so after Y and X are copied, they appear in the *wrong* order on the new heap for the lists Ai. So, after copying, there are on the heap in total n*2+2 cells, as in the right of Figure 1. Note that the fact that X and Y are free variables, is not essential: even if they had been instantiated, the same problem would remain. Postponing as in [19] on the other hand applies only to cells containing an undef, because garbage collection is always restricted to the most recent segment.

It is clear that if the order of copying the cells X and Y would have been the other way around - i.e. in the *right* order as far as list elements is concerned, either by a goal like `copy_in_order(X,Y,Z,A1,A2,...,An)` or `copy_in_order(Z,A1,A2,...,An)` - then it is possible to limit the duplication: copying X and Y leaves forwarding pointers that differ by 1. This can be easily checked before subsequent copying. Figure 2 shows this. We name this *optimistic* copying, because the idea is that hopefully the cells of a list are copied as tuple together before any duplication occurs.

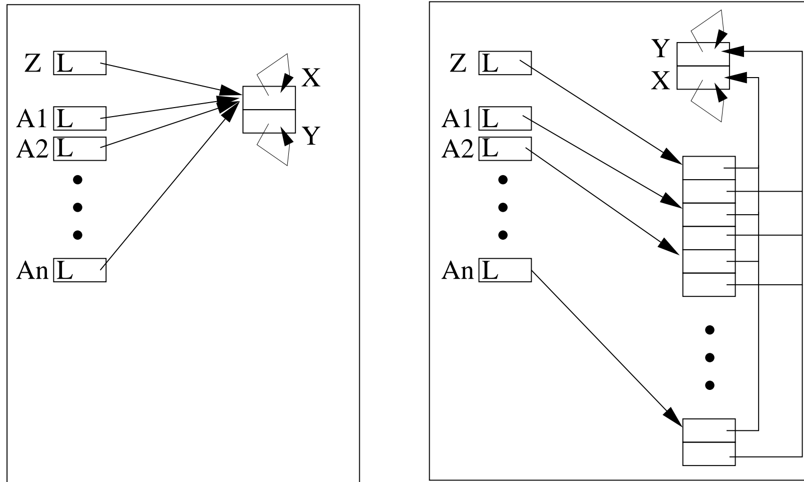

Figure 1: Arbitrary duplication by dangerous copying

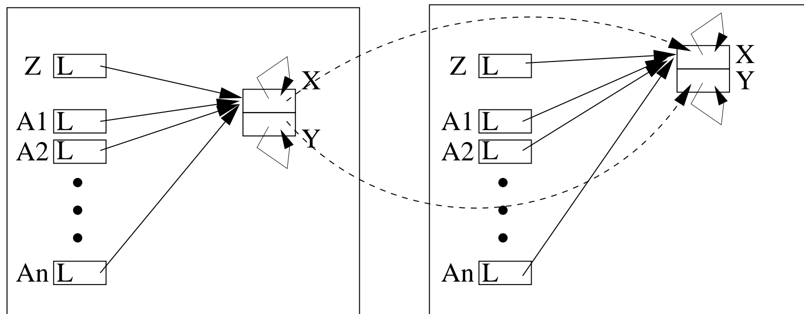

Figure 2: No duplication during optimistic copying: because we were lucky

The dotted arrows in Figure 2 are the forwarding pointers.

Optimistic copying goes wrong of course when the cells X and Y are first copied in the wrong order, or if only one of them was copied before the two cells are copied as a head-tail pair of a list. Fortunately, even in that situation, we can limit the duplication to two, by using the tag field in the forwarding pointers. Figure 3 shows an initial situation in the upper left part, in which there are 3 root pointers, pointing to the first element of a list pair, a cell containing the atom c, and a list pointer pointing to the list pair. The numbers next to the 3 root pointers indicate in which order the copying routine visits them.

The upper right part of Figure 3 shows the situation after the first two root pointers have been copied. Again, forwarding pointers are indicated with dashed lines. In Figure 3 the *from* and *to* space are now explicit. The second root pointer and the cell with the atom **c** only serve the purpose to make sure that the cell with **b** cannot be copied just after the cell with a anymore. When the third root pointer is treated by the copy routine, it creates a list-pair in the *to* space and copies the contents from the list-pair in the *from* space to it. It then leaves the usual forwarding pointers, but the one for the first cell of the list-pair is tagged with the tag D which indicates that this cell contained a forwarding pointer before: the previous forwarding pointer can still be retrieved. On following a subsequent list pointer

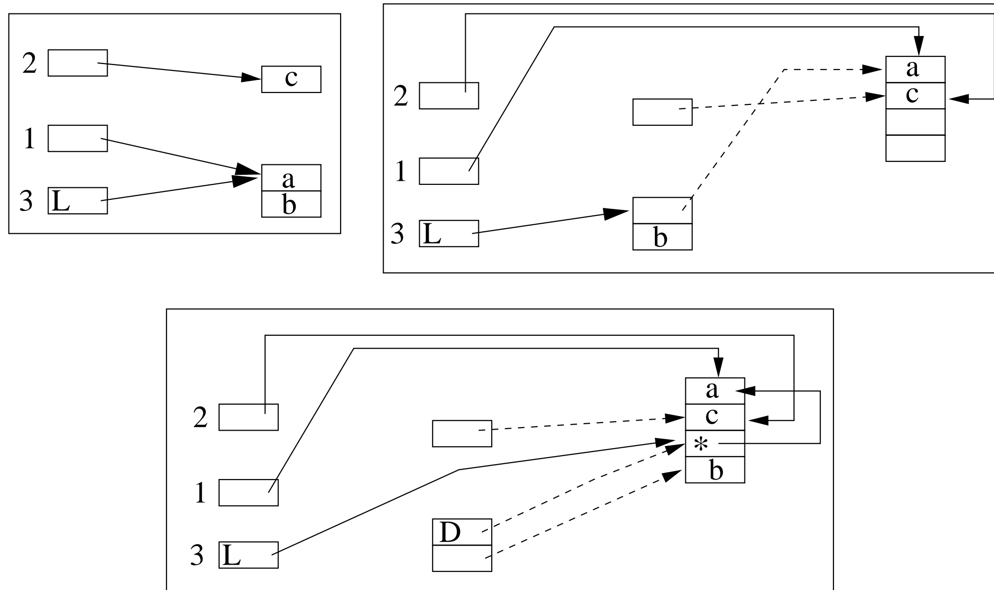

Figure 3: Limiting the duplication by cautious copying

to the list pair, the copy routine can easily recognize this situation as an already copied list-pair, because (after D-tag has been stripped if needed) the two forwarding pointers in the pair are subsequent. On following any other pointer directly to the first cell of the list pair, the D-tagged forwarding pointer is just stripped from its tag and treated as usual.

The above does not show the need for the D-tag yet: it has not been used in an essential way. The trail however needs the D-tag: indeed, when a trail entry points to a D-tagged cell, the relocated pointer in the trail must be the *original* forwarding pointer – and it can be retrieved by following the D-tagged pointer.

There remains to argue why the above is correct and optimal in some sense: first one can wonder why the cell marked with (*) contains a pointer to the cell with the atom **a**, instead of the atom **a** itself. Such would be an application of shunting ([]) but for the above situation, shunting is only allowed if the cell with atom **a** was not trailed: in the absense of this information, the safe way is not to shunt. Secondly, the above following of the D-tagged forwarding pointer when the cell is reached (directly) from the trail, makes sure the correct cell is trailed after the collection. Finally, it is true that we have introduced an extra reference to the atom **a**, but as far as forward execution is concerned, this is correct.

The example of course does not cover all possible situation (e.g. the cell with b could also have been copied already before a list pointer is found that points to the list pair), but the principle remains the same: as soon as two cells are copied as a list pair, the list tagged forward pointer will be written in the first cell of the pair.

In the implementation, we have used the for D the same tag as for floating point numbers, because that tag occurs least in practice. We name this variant *cautious* copying.

While adapting the recursive copying routine to garbage collection, we need to be aware of the fact that reference chains cannot be shortened (barring perhaps easy shunting as in [6]) and that the copy can be destructive.

Doing early reset is in general not mandatory and usually is performed during the marking phase. In the schemes without marking, we make sure that the root set is visited in the same order as the one marking must obey for making early reset possible. Then the forwarding pointers act like marks. In the schema which postpones dangerous cells (like [13, 19]), the order of copying becomes different and either one should empty the work list (with the postponed cells) before closing a segment with the treatment of its trail, or one should mark in some way the postponed cell. The alternative is to give up on early reset as soon as a cell is put on the work list.

A final note about the three variants of copying: unbounded duplication of cells during copying without marking or postponing happens only to lists; for other structured terms, the WAM uses a header which can easily be used to indicate when the term is copied as a whole, so this happens never more than once in all three variants, and by using the same mechanism.

Taking into account the above considerations, we used the hProlog copy_term/2 implementation as the start for a new garbage collector in hProlog and we did that for the above mentioned variants dangerous, optimistic and cautious.

8 Experimental results

Table 7 reports on the results of optimistic copying for garbage collection on the artificial benchmark of Section 4.

		hProlog	optimistic
depth-first	construct	0.34	
	collect	1.91	0.74
	copy	1.14	
breadth-first	construct	0.39	
	collect	2.10	0.82
	copy	1.18	

Table 7: Optimistic copying versus the original garbage collector for the artificial benchmark

The gain is enormous. It is also interesting to see the difference between original copying and optimistic copying collection: the fact that copying must preserve the original term is largely responsible for it.

Table 8 contains a comparison between all implemented variants on the benchmark set. The results are less pronounced, but reasonably consistent.

The following are worth nothing:

- cautious copying does rarely collect more garbage than optimistic copying: only in chess; optimistic copying performs better overall; cautious copying collects slightly less in mqueens than optimistic: we have as yet no good explanation for that
- dangerous copying is significantly slower than optimistic copying in the benchmarks with many shared lists; it collects the same amount of garbage in most benchmarks; for

	hProlog	dangerous	optimistic	cautious
emul	1.85/24.21 75 3098074456	1.91/24.16 75 3098074348	1.89/24.15 75 3098074348	1.86/24.21 75 3098074348
tspgc	0.02/24.10 8 42255172	0.03/24.07 8 42255100	0.02/24.12 8 42255120	0.02/24.12 8 42255120
dnamatchgc	0.14/3.46 15 78565940	0.11/3.47 15 78564528	0.10/3.51 15 78565684	0.11/3.47 15 78565684
chess	3.62/12.83 3 14646248	2.67/12.79 3 10986964	2.53/12.90 3 14427476	2.43/12.84 3 14529664
boyergc	4.19/14.04 5 1033713000	3.23/13.97 5 1033711960	3.13/13.94 5 1033711960	3.26/13.89 5 1033711960
serialgc	3.80/11.02 3 1040245812	2.75/10.84 3 1040248968	2.68/10.71 3 1040248968	2.74/10.77 3 1040248968
mqueens	39.71/97.50 41 15013225540	23.09/98.69 45 15017356516	18.34/97.32 41 15013223616	19.57/97.42 41 15013223576

Table 8: Comparing the hProlog collector with 3 plain copying ones

chess, it collects significantly less garbage; for mqueens, garbage collection is triggered more often, which can only be explained by the fact that more multiple copying goes on; because of the higher number of garbage collections, the amount of collected garbage is higher

- the original mark© collector in hProlog performs worse than optimistic collection; it collects only a tiny fraction more garbage, except in chess, where the difference is about 1.5%: in chess, the chess board represented by a list containing structured terms: these structured terms are often accessed separately from the board itself which leads to the small increase

- boyer does hardly use any lists, so even dangerous copying does not collect much less garbage than hProlog; but even for four other benchmarks that use a mix of lists and other structured terms – dnamatchgc, emul, serialgc, tspgc – the difference is tiny; only in chess is it larger and really large for mqueens: the latter uses lots of sharing between sublists

We think it is safe to conclude that the collector based on optimistic copying comes out as a clear winner.

9 Cache behaviour of the different collectors

In Tables 9 and 10 we show the results that we obtained by running the various combinations of benchmark/collector in a cache simulator. We used 'cacheprof', which is an open-source cache profiler. Cacheprof is written by Julian Seward and available from http://www.cacheprof.org. The program traps all memory references during execution of a benchmark and feeds them to its cache simulator. Afterwards it can print summaries for each function and annotate the corresponding source code. We processed the acquired data and for all implemented collectors we compared the memory behaviour for collector and mutator. The cache modelled for these tests was the L2 cache of a Pentium II with the following properties: 512 kb in size, associativity 4, 32 bytes per line. The timings were taken from table 1. This machine has a different cache configuration (256 kb in size, associativity 8, 32 bytes per line), but we expect the global picture to be the same as for the cache modelled.

benchmark		hProlog Cheney	dangerous	optimistic	cautious
emul	cache misses collector (K)	11698075	11333002	11331540	11333547
	cache misses mutator (K)	14157210	14259390	14257765	14259643
	cache misses total (K)	25855285	25592392	25589305	25593190
	total # of instructions (M)	17386323323	17343359130	17343410430	17344912678
	total # of refs in collector (M)	**209930170**	276537087	276570687	277587042
	total # of refs in mutator (M)	8943320131	8943685521	8943685746	8943685746
	total # of refs (M)	9153250301	9220222608	9220256433	9221272788
	time in collector (ms)	1850	1910	1890	1860
	time in mutator (ms)	24210	24160	24150	24210
tspgc	cache misses collector (K)	153124	125536	125070	125515
	cache misses mutator (K)	1495427	1503233	1501598	1503167
	cache misses total (K)	1648551	1628769	1626668	1628682
	total # of instructions (M)	23113133954	23108763636	23108770261	23109039599
	total # of refs in collector (M)	6700037	3876126	3885785	4115143
	total # of refs in mutator (M)	13680844638	13680845426	13680844798	13680844798
	total # of refs (M)	13687544675	13684721552	13684730583	13684959941
	time in collector (ms)	20	30	20	20
	time in mutator (ms)	24100	24070	24120	24120
dnamatchgc	cache misses collector (K)	593043	451481	451807	451531
	cache misses mutator (K)	2676447	2674511	2674334	2674435
	cache misses total (K)	3269490	3125992	3126141	3125966
	total # of instructions (M)	2639472902	2619718891	2619197678	2626733127
	total # of refs in collector (M)	22120409	11445034	12194503	16378416
	total # of refs in mutator (M)	1544528227	1544772338	1544528596	1544528596
	total # of refs (M)	1566648636	1556217372	1556723099	1560907012
	time in collector (ms)	140	110	100	110
	time in mutator (ms)	3460	3470	3510	3470

Table 9: Relating the cache behaviour and # of instructions to performance

The table tells us the following:

- the memory behaviour of the copyterm based collectors in general seems to be better than for the Cheney based collector; this can be seen in the number of cache misses and references inside the collector; we expect the new collectors to have less memory references because they don't need a marking phase; note however that we cannot hope for only half as much references: while every cell in the heap is only touched once instead of twice, we walk through the root set (choicepoints, environments, trail) twice instead of thrice

benchmark		hProlog Cheney	dangerous	optimistic	cautious
chess	cache misses collector (K)	14265875	9408136	9250673	9251531
	cache misses mutator (K)	3371439	3371078	3369552	3369811
	cache misses total (K)	17637314	12779214	12620225	12621342
	total # of instructions (M)	13776615178	13180178554	13161438831	13203586977
	total # of refs in collector (M)	768825225	466086150	462420912	485952297
	total # of refs in mutator (M)	6910577149	6910576840	6910576840	6910576840
	total # of refs (M)	7679402374	7376662990	7372997752	7396529137
	time in collector (ms)	3620	2670	2530	2430
	time in mutator (ms)	12830	12790	12900	12840
boyergc	cache misses collector (K)	23957597	17621188	17620380	17621056
	cache misses mutator (K)	10351578	10350610	10351824	10350741
	cache misses total (K)	34309175	27971798	27972204	27971797
	total # of instructions (M)	12849356086	12434890558	12434891498	12447596447
	total # of refs in collector (M)	648300200	558874717	558875407	568361918
	total # of refs in mutator (M)	6493883331	6493888169	6493888184	6493888399
	total # of refs (M)	7142183531	7052762886	7052763591	7062250317
	time in collector (ms)	4190	3230	3130	3260
	time in mutator (ms)	14040	13970	13940	13890
serialgc	cache misses collector (K)	22078051	14817196	14817199	14817213
	cache misses mutator (K)	23901301	22301138	22301293	22300767
	cache misses total (K)	45979352	37118334	37118492	37117980
	total # of instructions (M)	9764219089	9300051838	9306503150	9361391148
	total # of refs in collector (M)	589914103	419176203	424014678	454693026
	total # of refs in mutator (M)	**4348158739**	4348080021	4348080030	4348080030
	total # of refs (M)	4938072842	4767256224	4772094708	4802773056
	time in collector (ms)	3800	2750	2680	2740
	time in mutator (ms)	11020	10840	10710	10770
mqueens	cache misses collector (K)	183269823	77075559	62503844	62503617
	cache misses mutator (K)	48922554	48881377	48880112	48880174
	cache misses total (K)	232192377	125956936	111383956	111383791
	total # of instructions (M)	101447272125	93435708552	91377885905	95405567130
	total # of refs in collector (M)	8585223189	2985613266	2763038271	**4995962042**
	total # of refs in mutator (M)	50196319900	50303916058	50195449591	50195462594
	total # of refs (M)	58781543089	53289529324	52958487862	55191424636
	time in collector (ms)	39710	23090	18340	19570
	time in mutator (ms)	97500	98690	97320	97420

Table 10: Relating the cache behaviour and # of instructions to performance

- for the number of references in the mutator, we expect these numbers to be bigger for the dangerous, optimistic and cautious collectors than for the Cheney collector because they possibly copy some cells twice and in this way create chains; however the difference seems to be very small and reaches only 0.1% in the mqueens benchmark, where it is highest

- the collectors dangerous, optimistic and cautious show very similar results; we expect dangerous to have more references in the mutator because of the chains it creates while copying lists; this effect is clearly visible in the mqueens benchmark

- while in most benchmarks it seems that the number of cache misses in the collector (going from Cheney based to copyterm based) seems to be the determining factor for the time spent in the collector, also the number of references in the collector is important; the reason is probably that for each extra reference a number of extra instructions is executed, these cost extra time

- the mqueens benchmark also shows us the relation between the relative size of the root set and the number of references in the collector; in this case we see that the root set

is very small compared to the number of heap cells and the number of references in the optimistic collector is only 35% of the same entry of the Cheney collector

From these cache simulations we can conclude that our collectors based on the copyterm algorithm have a better cache behaviour and that this property leads to better performance.
However, some numbers are difficult to explain: they are indicated in the tables in bold.

- the low number of references in the hProlog collector for emul: we probably must analyse more carefully the number of references in each collector;
- the number of references in the hProlog mutator for serialgc is higher than for the others; the difference is small, but one would expect it to be in the opposite direction
- the high number for mqueens, cautious, total number of references in the collector: this can be explained by the combined effect of setting and testing for the D-tag

We intend to gather more information on the benchmarks and their execution, so that we can explain the figures better. We also might uncover some systematic performance problem in some pieces of code.

10 Recovery from overflow

Since omitting the marking phase before copying entails the risk that the copied heap is larger than the original one, an overflow during copying can occur. This means that the overflow should be caught and then recovered from. The easiest way for recovering consists in allocating an extra chunk of memory and continue copying to that extra space. This can possibly happen more than once: actually, one could optimistically allocate initially only a small chunk at the start of the collection (perhaps based on the history of previous collections) and allocate more when needed). After copying is finished completely, the total size of all the chunks is known and enough space can be allocated to contain all of them. Then the chunks are glued back together and appropriate relocation of heap and root pointers must be done: this is time consuming, but not difficult.

The benchmarks seem to indicate that such overflow is rare, and indeed, we have observed it only almost all data in the heap is live: as observed in many places, this seems to happen very rarely in real life. As a rule, more than 50% of the heap is garbage.

We have not implemented the recovery (yet), but the overflow is checked of course.

11 Related work, future work and conclusion

We have investigated the relation between the built-in predicate copy_term/2 and garbage collection. We have tried to use the usual garbage collection algorithms for building an implementation of copy_term/2 and concluded that this does not offer the desired performance. On the other hand, using the basic implementation of copy_term/2 for garbage collection works well: on average we get a nice improvement of the garbage collection time on our

set of benchmarks. Other systems use such a copying algorithm as well: they postpone the copying of heap cells that could cause double (or more) copying. Both [13] and [19] use it only for an incremental garbage collection which never performs a major collection. Those systems' policy is sufficiently different from ours that a head on comparison is impossible. Moreover, the system [13] is not available and B-Prolog [19] is no longer open-source: the information the user can get about the garbage collection process is not enough for making meaningful comparisons.

The next step in our research will be to investigate variations on the copying algorithms based on copy_term/2 or on mark&Cheney. These variations will be tuned to lower the cache misses in the mutator after a collection has taken place, while hopefully not increasing the cache misses in the collector. If this can be achieved, a global speedup is expected.

Acknowledgements

Part of this work was conducted while the first author was a guest at the Institut de Mathematiques Appliquées of the Université Catholique de l'Ouest of Angers, France. Sincere thanks for this hospitality. We also thank the (main) authors/implementors of Prolog systems that come with sources: Mats Carlsson (SICStus Prolog), David S. Warren (XSB), Vitor S. Costa (Yap) and their teams, and also Henk Vandecasteele for his work on the ilProlog compiler used within 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] A. W. Appel and M. J. R. Goncalves. Hash-consing garbage collection. Technical Report CS-TR-412-93, Princeton University, Feb. 1993.

[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] 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 Lecture Notes in Computer Science, pages 88–101. Springer-Verlag, Sept. 1994.

[5] M. Carlsson. *Design and Implementation of an Or-Parallel Prolog Engine*. PhD thesis, The Royal Institute of Technology (KTH), Stokholm, Sweden, Mar. 1990 See also: http://www.sics.se/isl/sicstus.html.

[6] 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.

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

[8] Y. C. Chung, S.-M. Moon, K. Ebcioglu, 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.

[9] V. S. Costa. Optimising bytecode emulation for prolog. In *Proceedings of PPDP'99*, volume 1702 of *LNCS*, pages 261–277. Springer-Verlag, Sept. 1999 See also http://www.ncc.up.pt/~vsc/Yap/.

[10] B. Demoen, G. Engels, and P. Tarau. Segment order preserving copying garbage collection for WAM based Prolog. In *Proceedings of the 1996 ACM Symposium on Applied Computing*, pages 380–386. ACM Press, Feb. 1996.

[11] B. Demoen, M. García de la Banda, W. Harvey, K. Mariott, and P. Stuckey. An overview of HAL. In J. Jaffar, editor, *Proceedings of the International Conference on Principles and Practice of Constraint Programming*, volume 1713 of *LNCS*, pages 174–188. Springer, 1999.

[12] 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.

[13] X. Li. Efficient memory management in a merged heap/stack Prolog machine. In *Proceedings of the 2nd ACM SIGPLAN Conference on Principles and Practice of Declarative Programming (PPDP'00)*, pages 245–256. ACM Press, 2000.

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

[15] K. Sagonas, T. Swift, and D. S. Warren. XSB as an efficient deductive database engine. In *Proc. of SIGMOD 1994 Conference*. ACM, 1994.

[16] D. Sahlin. Making garbage collection independent of the amount of garbage. Technical Report R87008, SICS, 1987.

[17] D. Sahlin and M. Carlsson. Variable shunting for the WAM. Technical Report SICS/R-91/9107, SICS, 1991.

[18] D. H. D. Warren. An abstract Prolog instruction set. Technical Report 309, SRI, 1983.

[19] N.-F. Zhou. Garbage collection in b-prolog. In B. Demoen, editor, *Proceedings of the First Workshop on Memory Management in Logic Programming Implementations, co-located with CL2000*, pages 1–10, http://www.cs.kuleuven.ac.be/~bmd/mmws.html, July 2000.

Appendix: about benchmarking and the benchmarks.

Benchmarks come in two sorts - although a particular program my be used for both.

- heap gc is triggered by *overflow*
- heap gc is triggered by calling the predicate *garbage_collect/0* or a variant of it

Both sorts have their merit, but they should not be confused and their results should be interpreted with care.

When heap garbage collection is triggered by overflow - that is, the allocator decides it is time for a memory management action - the memory management policy will influence the results of the benchmark; for instance

- is there a decision to collect or to expand ?
- is there a choice between different types of collection, as in:
  - minor or major
  - sliding or copying (at least 2 systems had or have both !)
  - perhaps a quick version of the collector that does no cleaning of the trail, early reset, variable shunting ...
- is heap garbage collection triggering memory management actions for other regions ?

Furthermore, some systems use the famous *trim_core* which at some points in time decides to *shrink* the heap - either by reallocating the heap to a smaller area, or by moving the heap limit pointer up. Such action occurs typically at backtracking - e.g. to the toplevel - but can also be taken after a collection that recovered a large fraction of the heap.

When treating a system with its garbage collector as a black box, it can be very difficult to understand such policy and this can lead to the wrong interpretation of benchmark results.

When garbage collection is called explicitly by garbage_collect/0, the situation is not better: a system can either follow the same policy as on overflow, interpreting the current heap pointer as the heap limit or not - which can lead to a different policy. SICStus Prolog performs always a major collection on calling garbage_collect/0, even though it is generational on overflow. This means that if only benchmarks with explicit triggering of garbage collection are used, the generational aspect of that system will go unnoticed. Systems might even ignore the request for garbage collection, because it *knows* there is no gain at this moment - e.g. if the heap usage is very small, or if the heap has just been collected, making a conjunction of goals like *garbage_collect, garbage_collect* meaningless.

So, the least one must do, is make sure that the characteristics of the benchmarks are clear, i.e. to which sort they belong. Other useful information about benchmarks is

- the size of the root set at gc

- the size and percentage of persistent data - that is the data that survives 2 consecutive (major) collections
- the relative amount of live versus allocated data

There is value in comparing systems as a whole, because that is what the users experience. But is is also valuable to be able to separate out issues and get an idea of policy and algorithms separately. In practice, such separation is often impossible, as systems typically don't even provide an accurate means to set the maximal heap size, or to switch off a policy. Implementors would help each other greatly by providing at least means to

- set the heap limit exactly, i.e. a limit which when crossed (or about to be crossed) triggers garbage collection; such a limit should be independent on the other stack limits
- switch off any expansion and shrinking policy
- switch off any generational aspects
- switch off extra passes for variable shunting for instance

Another problem are optimizations, especially in artificial benchmarks: terms must be kept alive, otherwise the collector has no work (at least the marking phase is short if all data is dead). The trick that most often works is writing such benchmarks according to the schema:

```
testgc(Input) :-
        generate(Input,Useful,Garbage),
        garbage_collect,
        use(Useful).
```

and add the fact

```
use(_).
```

This does not work in a system that inlines (or unfolds) goals that match only one head - and in a later phase of the compilation detects that the unification with a void variable can be removed. Such optimizations do have a large impact on non-artificial benchmarks as well.

When garbage collection is triggered by overflow, one observes usually a different number of garbage collections in different systems. One explanation is sometimes that a generational system tends to call the collector more often. Another explanation can be the allocation policy itself: hProlog for instance allocates all free variables on the heap. Also specialised built-in predicates can influence the number drastically. Finally, it is always possible that the initial heap size is just above or below a point which would have triggered one garbage collection more or less. In a system that does not allow to set the initial heap size very precisely, the difference can be even larger.

Table 11 contains for each benchmark where we obtained it from and its size (in number of program lines). Then follows the average percent of data that survives collection $(i+1)$ after it has survived collection i: a high number indicates that the benchmark can benefit from generational collection. The survival size indicates the average number of cells that survived a garbage collection . To give an idea of the size of the root set, there is a column that gives the maximal size of each stack (trail, local stack, choice point stack) during the execution of the benchmark. The last column indicates whether garbage collection was triggered by overflow or by calling the built-in *garbage_collect/0*.

benchmark	origin	# lines	survival %	survival size (cells)	max stack sizes trail-ls-cp	trigger
emul (*)	[6]	1228	98	16600	172329-3866-9841	overflow
tspgc (*)	[13]	175	98	23300	7895-83-104	overflow
dnamatchgc (*)	[13]	141	86	34471	13942-76-104	overflow
chess (**)	[6]	704	36	1746304	1805065-6276977-5214953	explicit
boyergc (*)	[13]	533	76	3176103	3985228-463-236	overflow
serialgc (*)	[13]	107	34	4042536	5882854-122-104	overflow
mqeens	(***)	119	99.9	5577525	82-129-104	overflow

Table 11: Benchmarks and some of their characteristics

(*) These are classical benchmarks.

(**) The program as used in [6] was extended with one more chess move by K. Sagonas in order to make the execution more interesting for garbage collection .

(***) was written by B. Demoen; it is a naive solution to the M-queens problem of the 7th Prolog Programming Contest.

The very large trail size might seem weird, but hProlog does not tidy the trail on cut as SICStus Prolog and Yap do. Note also that hProlog has a (non-optimized) value trail.

We have not taken some benchmarks from other papers, because of the following reasons

- boyer from [6]: too small and and all garbage collection is after the use
- browsegc from [13]: with hProlog default only one collection which takes only 140 out of 13310 msecs
- sim, nand, chat from [6]: too small in runtime and garbage collection
- the GNU-Prolog compiler from [6] (but original in the GN-Prolog distribution): uses assert which hProlog does not support
- other benchmarks from [13] were not used because they are specifically well suited for generational collection, which is not the issue here