Garbage Collection for Prolog based on Twin Cells

Joachim Schimpf
European Computer Industry Research Centre (ECRC)
Arabellastraße 17, D-8000 München 81, FRG

October 9, 1990

Abstract

A garbage collection algorithm for a WAM-based Prolog system is presented. It uses a classical mark-and-compact approach, but requires less passes through the data areas than previous algorithms. It is applicable when the smallest garbage-collectable entity occupies an amount of space which is sufficient to store two pointers plus 2 bits with each pointer. This collector has been implemented in the SEPIA Prolog system.

1 Introduction

In state-of-the-art LISP systems, the method of choice for garbage collection is *copying*, which has superseded the *mark-and-compact* approach. Unfortunately, copying collectors usually change the order of memory cells which makes them difficult to use for Prolog. The main resons are:

- The Prolog terms are allocated in a stack like fashion in order to be able to pop this stack on failure (This space reclamation on failure makes it even possible to write memory-intensive Prolog programs that don't need garbage collection at all).
- In WAM-based Prolog implementations, compound objects consist of a sequence of simple objects that are adjacent in memory. A special (typed) pointer references the beginning of the compound object. However, the components can also be referenced independently, which requires special treatment to prevent a copying collector from taking apart the compound object.

Many recent Prolog garbage collectors ([1], [2]) therefore use a mark-and-compact scheme. This can be given a certain amount of incrementality by considering segments of the copy stack, defined by choicepoints. For compiled Prolog this was first suggested by [6].

The other important feature of this class of Prolog garbage collectors is that they take the reset information (available on the trail stack) into account. This is known as *virtual backtracking* [3] or *early reset of variables* [1]. This technique improves the precision of the marking phase in determining those objects that have to be preserved for backtracking.

Recently, Touati [7] has proposed a scheme for exploiting copying where possible, i.e. in deterministic execution sequences. However, a compacting algorithm is still needed as a fall back method.

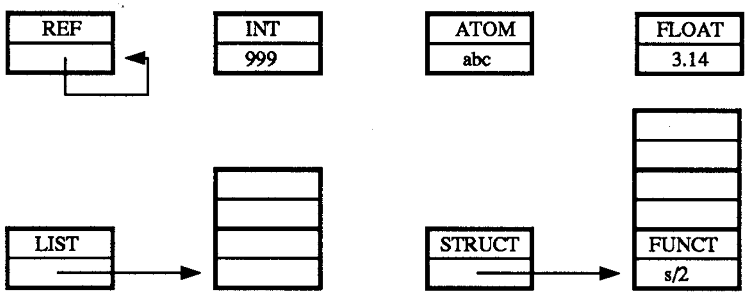

Figure 1: Representation of Prolog data

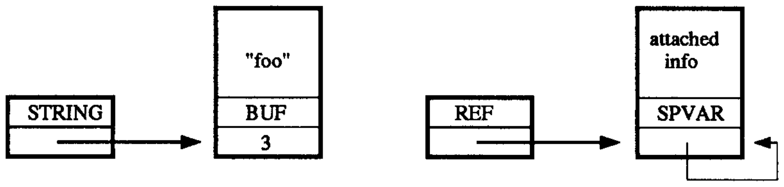

Figure 2: Non-standard data types: a string and a special variable

2 Preliminaries

Our garbage collection algorithm assumes a WAM-like abstract machine [8]. All data is represented in tagged form. The important assumption we make is that the smallest independently collectable data item occupies an amount of space which is sufficient to store two pointers plus 2 bits with each pointer. We will call such an elementary data item a *twin cell*. Before we discuss the benefit of twin cells for garbage collection, we shortly recall their more obvious advantages.

Many current Prolog implementations pack a Prolog object into a single 32-bit cell (we will refer to this as the *unit-cell* model). It involves stealing some of the 32 bits to represent the tags. The size of the value part is then restricted accordingly. Such a scheme has several drawbacks. In particular, the representation of Prolog values (like integers, floats and pointers) is incompatible with the one in commonly used statically typed languages (such as C).

As opposed to that, in the SEPIA [4] system, the basic Prolog object is in fact represented as a twin cell. It consists of two 32-bit words, the first holding the value, the second holding the tag. The standard Prolog data types are conveniently represented in this form (figure 1) and new types can be easily added. Obviously, the space consumption is higher, on the other hand tagging and untagging overhead is saved during execution.

For various extensions of the basic system, we allow bigger objects to be stored on the *global stack* (the *heap* in terms of [8]). These objects have a size of (2+n) 32-bit words. One example is the string data type, for which an arbitrary sequence of characters has to be stored (figure 2, left hand side). It consists of a twin cell followed by the number of unit cells necessary to store the string proper.

An even more important example of complex objects are special variables. While a standard variable is simply represented as a self reference with an appropriate tag, special variables may have attached information of various sizes. Due to the size of the attached information, special variables can only be stored on the global stack.

This flexibility is a problem for the garbage collection algorithm described in [1]), since the global stack can no longer be traversed from top to bottom (unless the objects are tagged at both ends).

It was said above that it must be possible to store 2 additional bits together with a pointer. This is easily possible on byte-addressing machines, when an addressing granularity of 32 bits is used in the Prolog system. Then the two least significant pointer bits are meaningless and can be used to store the additional information.

3 The Algorithm

The basic structure of our collector is similar to the one that has been used successfully in [6], [1] and [2]:

- It is a compacting collector
- It is incremental, based on choicepoint segments
- It performs *early reset* of variables

Since these features have been described in the works cited above, we will concentrate here upon how the availability of the twin cells can be exploited to improve the algorithms mentioned previously.

We will therefore ignore the details of virtual backtracking and just assume that we have a known set of Prolog objects that can serve as the root set for determining accessibility of objects in the stack segment that we are going to garbage collect (called the *collection segment*).

To account for the incrementality we assume that there is an abstract machine register GCB which denotes the youngest choicepoint that has already existed during the last collection. This choicepoint virtually separates all stacks into an old and a new part, The part of the global stack newer than this choicepoint is the collection segment that the collector works on (figure 3).

We will also assume a split-stack variant of the WAM, i.e. choicepoint and environment stack are separated, but this does not affect the algorithms in any way.

3.1 Calling the collector

The garbage collector has to be called in a well-defined state of the abstract machine. To guarantee this, the abstract machine is always in a state that resembles the state immediately after a `Call` instruction (i.e. just before entering a predicate). In this state the machine's arguments registers hold a known number of valid arguments. This number is given as a parameter to the garbage collector. Note that this is a situation where the machine could create a choicepoint in normal execution. Following a suggestion of [2], we create such a choicepoint before starting the actual garbage collection. This has the effect of storing the current machine state (its registers) in memory locations, which makes the subsequent algorithms more uniform because no special handling for the current state is needed (as is in [1]). After the collection, the updated registers (argument registers, global and trail stack pointers) are restored from this choicepoint, and the choicepoint is popped. The associated overhead is negligible.
The top level procedure of the collector looks like this:

```
collect_stacks(arity)
{
    save the machine registers in a new choicepoint;
    mark accessible objects inside the collection segment and build
            relocation chains;
    compact the global stack, on the fly updating pointers to
            the relocated objects;
    compact the trail and update trail pointers in choicepoints;
```

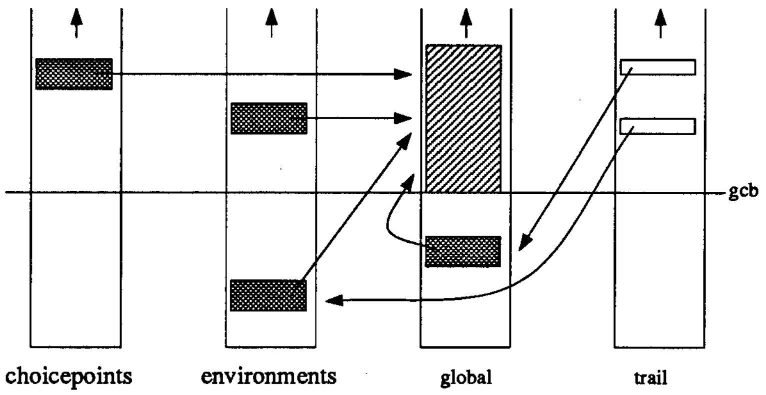

Figure 3: Overview of the stack areas (collection segment hatched, roots grey)

```
    restore the new machine state from the choicepoint and pop it;
}
```

3.2 Mark&Link Phase

The marking phase consists of finding all accessible objects in the collection segment. These objects can be referenced from:

1. arguments saved in choicepoints
2. permanent variables stored in environments newer than GCB
3. permanent variables stored in environments older than GCB
4. global stack cells older than GCB

This is the root set from where our marking phase starts. Figure 3 gives an overview of the stack areas. The roots are displayed in grey, the collection segment is hatched.
The Mark&Link phase does two jobs. It can be done in one or two passes (cf. section 5):

1. marking the accessible objects
2. building *relocation chains*

For marking, we reserve one bit in the tag cell of every object inside the collection segment, called the MARK bit. This bit is used by the compaction phase to tell garbage from non-garbage. Before and after a garbage collection, all these bits are zero.

The purpose of *relocation chains* is to be able to update pointers to data objects which are moved during compaction. All cells containing a pointer to a certain object in the collection area are linked into a chain starting at this target object (cf. figure 4). The chain pointer of course overwrites at least a part of the original target object, so this has to be moved elsewhere. The only place that is available is the last cell of the relocation chain. It originally contained a pointer, and can now be used to preserve the overwritten part of the target object.

This is the point where the twin cells become essential. In a unit-cell system, it is not possible to build all relocation chains at once. This is because an object can not be the head of its own relocation chain and at the same time be a member of its target's chain. Morris' algorithm [5] solves this problem by doing two passes over the collection segment, in opposite directions.

In the twin-cell model, this restriction does not exist. There we can use the following convention:

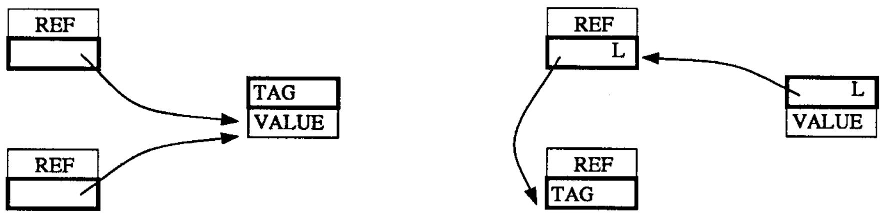

Figure 4: Building a relocation chain

- the *tag* cell of an object is used to store the head of the objects's relocation chain, linking all references to this object.
- the *value* cell may be the member of another chain, starting from the tag cell of the object it originally pointed to. When the value was a constant rather than a pointer, it remains unchanged.

Figure 4 shows a situation with two references to a simple target object (left hand side). At the end of the Mark&Link phase there is a relocation chain starting from the target's tag and connecting all value cells which previously held a pointer to the target. The last cell of the relocation chain preserves the original tag of the target. Obviously, it is necessary to have an indicator to distinguish a tag from a relocation link. This is accomplished by reserving a second bit, the LINK bit. When set, the cell holds a relocation link, when reset (the default) it holds a tag.

After the marking phase the situation is as follows:

- The reachable objects in the *collection segment* have their tag's MARK bit set.
- The tag's LINK bit is set if moving the object requires updating references. In this case the tag cell holds a relocation chain.

We end up with 4 possible bit combinations, having the following meaning:

MARK	LINK	meaning
0	0	garbage object
0	1	referenced garbage object, tag cell holds relocation chain
1	0	useful object, but not directly referenced
1	1	referenced object, tag cell holds relocation chain

The second combination may not seem useful. It is needed for the global stack pointers that are saved in choicepoints. They may reference garbage cells, but have to be updated when the stack is compacted.

3.3 Compact&Update Phase

What is left for the compaction phase is to move the marked objects to the bottom end of the collection segment, keeping their order, but removing gaps of unused space between them. Additionally, all references to the relocated objects have to be updated and the marking bits must be reset.

The collector described in [1] uses a two-pass algorithm based on [5], comprising a top-down and a bottom-up pass through the collection segment. Our algorithm is different in that everything is done in a single bottom-up pass through the collection segment. This is possible as we have already built the relocation chains during the marking phase.

Note that this not only has the advantage of saving a pass, but it also eliminates the need for a top-down traversal of the global stack. As mentioned above, Prolog extensions

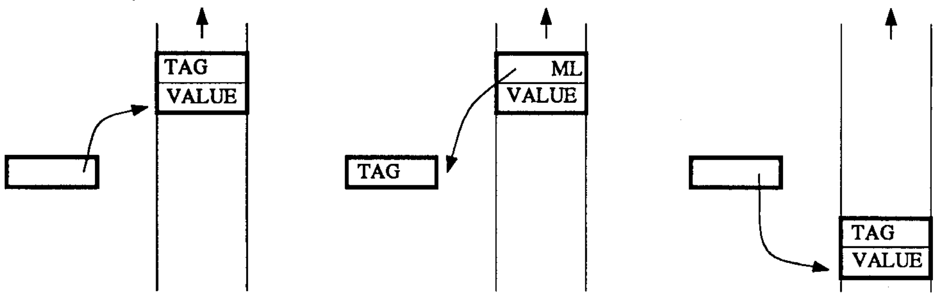

Figure 5: Updating a reference from the root set

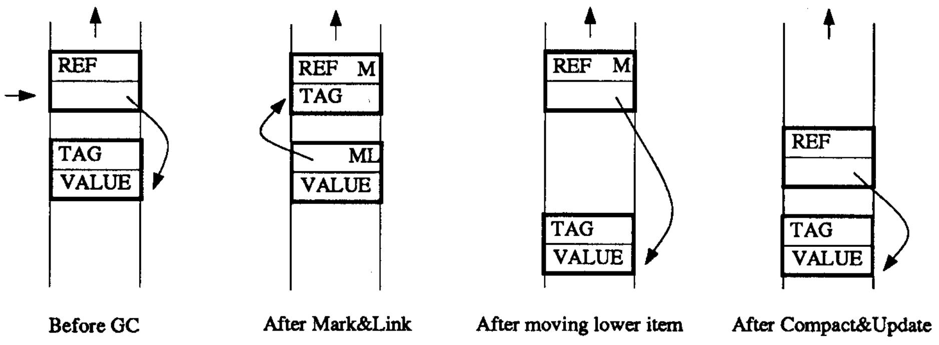

Figure 6: Updating a pointer down the global stack

often require arbitrary-sized objects on the global stack. These objects are only tagged at their lower end, making it at least difficult to traverse the stack in the opposite way.

Figure 5 shows the process of marking, moving and updating an object referenced from outside the global stack. Figure 6 shows the very similar case of a pointer internal to the collection segment where the target is older than the reference.

Pointers going from the collection segment to the collection segment in upward direction have to be handled differently. The reason is that we update the pointers at the same time as we move the target object. But in this case the reference is moved before the target is moved, which would destroy the relocation chain. The solution is to delay the building of the relocation link until after the reference has been moved to its new location. This means that we have to check for this condition in the marking phase, and if it holds, we only set the MARK bit in the target tag without replacing the tag by a link. In the Compact&Update phase, the link is created after the reference has been moved. When the target is moved later on, the reference can be updated in its proper place. The process is shown in figure 7.

4 Analysis

Being a compacting method, the complexity of the presented algorithm is proportional to the size of the whole collected area (i.e. garbage + non-garbage), while copying algorithms depend only on the amount of non-garbage. However, we will show that it considerably reduces the constant factors involved.

For a realistic comparison, we consider an algorithm that is as close as possible to ours, but based on the unit-cell model. It is therefore similar to the one in [1], but using

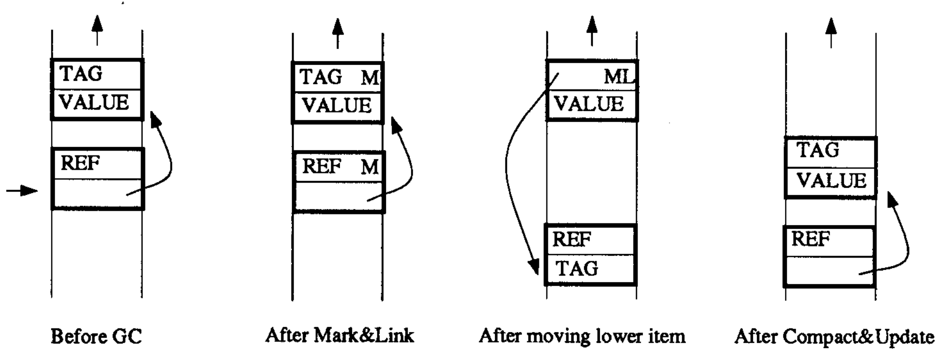

Figure 7: Updating an upward pointer in the global stack

standard recursive marking:

1. traverse all reachable objects and set mark bits in the collection segment
2. traverse root set and insert root pointers into relocation chains, resetting mark bits in environments
3. scan the global stack top-down, insert upward pointers into relocation chains and update the partial chains when passing the target
4. scan the global stack bottom-up, insert down-pointers into relocation chains and update them when reaching the target. Move non-garbage objects down the stack.

We first consider the effort for marking and updating references in terms of memory accesses. Let R_p be the number of root cells that hold pointers into the collection segment, R_s the number of other root cells, S_p the number of non-garbage pointers from the collection segment into the collection segment, and A the number of arguments of non-garbage structures. For the unit-cell model we get

- $2R_s$ read + $2R_s$ writes for marking/unmarking in the two root scans
- $4R_p$ reads + $4R_p$ writes for marking, unmarking, linking and updating pointers from the root set into the collection segment
- $3S_p$ reads + $4S_p$ writes for marking, linking and updating internal pointers in the collection segment
- A reads + A writes for marking arguments

For compacting let S be the number of surviving global stack objects and G be the number of garbage objects. Then we need

- 2S reads + S writes for scanning and moving the objects
- 2G reads to test and skip the garbage objects

This results in a total of

$$unit_r = 2R_s + 4R_p + 3S_p + A + 2S + 2G$$
$$unit_w = 2R_s + 4R_p + 4S_p + A + S$$

For the twin-cell model the analysis is similar. Note that, as above, we are counting read/write accesses for single cells, while the total data occupies twice the space that is needed by the unit-cell model.

- $2R_s$ read + $2R_s$ writes for marking/unmarking the root tags
- $5R_p$ reads + $5R_p$ writes for marking, unmarking, linking and updating pointers from the root set into the collection segment

Benchmark	#coll.	R_s	R_p	S_p	S	G
Boyer	17	766	1074	39071	1061184	681352
Browse	1	1627	721	2972	45440	20096
Edf	3	1821	3067	10203	137808	59040
Spreadsheet	1	935	145	278	5840	59712
Plm Compiler	4	238	218	6638	89080	149960
Toesp	4	103	84	97	2168	260920
Tp	21	9881	13045	24975	581896	839768
Chat	1	31	35	1923	35160	30376
Theorem Prover	8	2661	1445	3798	72336	449328

Figure 8: Root, Pointer, Survivor and Garbage Cells

- $3S_p$ reads + $3S_p$ writes for marking, linking and updating internal pointers in the collection segment
- A reads + A writes for marking the tags of arguments
- 2S reads + 2S writes for moving tag and value of the useful objects
- G reads to test the tag and skip garbage objects

This results in a total of

$$twin_r = 2R_s + 5R_p + 3S_p + A + 2S + G = unit_r + R_p - G$$
$$twin_w = 2R_s + 5R_p + 3S_p + A + 2S = unit_w + R_p - S_p + S$$

Figure 8 gives the parameters for a number of nontrivial benchmark programs[1]. It turns out that, despite the fact that all data areas are twice as large in the twin-cell model, the garbage collector is not necessarily slower than its counterpart in the unit-cell model. The most important reason being that there is only one access to every garbage object, while the unit-cell algorithm does two (thus compromising its advantage of having only half-size objects). For all benchmarks given, the twin cell algorithm needs less read accesses.

On the other hand, moving the useful objects requires two consecutive writes for the twin cell, but only a single one for the unit cell. Writing consecutive memory locations, however, is usually implemented efficiently in modern hardware, thus reducing its negative impact. In summary, for the benchmarks with more than about 50 percent garbage ratio, the twin cell algorithm does less memory accesses (read and write) than the unit cell algorithm.

5 Virtual Backtracking

The marking phase as described in [1] needs two passes through the root set. This is because environment cells are marked in the first, and unmarked in the second pass. Note that these marks are not really essential since no garbage is collected in the environment stack. Their use is solely

1. to control the environment traversal algorithm by marking the already visited cells
2. to enable early reset of environment variables

The first use can be avoided by employing a different traversal algorithm, that automatically keeps track of the visited parts of environments.

[1]The garbage collector was triggered whenever 64 kBytes of global stack were used up

Benchmark	Virtual Backtracking				
	none	global only		full	
Boyer	681352	681352	0 %	681352	0 %
Browse	20096	20096	0 %	20192	+0.5 %
Edf	59024	59040	+0.0 %	64928	+10.0 %
Spreadsheet	59712	59712	0 %	64096	+7.3 %
Plm Compiler	149960	149960	0 %	152720	+1.8 %
Toesp	260920	260920	0 %	260920	0 %
Tp	828888	839768	+1.3 %	861512	+3.9 %
Chat	30376	30376	0 %	30376	0 %
Theorem Prover	449024	449328	+0.0 %	451136	+0.5 %

Figure 9: Effect of virtual backtracking on the number of collected bytes

When one decides to drop early reset of environment variables, then no marking needs to be done in the environment stack and the two passes can be collapsed into a single one. The complexity of the twin cell algorithm is reduced to

$$twin'_r = R_s + 3R_p + 3S_p + A + 2S + G$$
$$twin'_w = 3R_p + 3S_p + A + 2S$$

This will speed up the marking process, but will reduce the amount of collected garbage for some programs.

To get an idea of how much is lost by this simplification, we have measured our benchmarks with three different variants of the garbage collector (figure 9). The first one does no virtual backtracking at all, the second one uses the simplified algorithm (i.e. only variables on the global stack can be early reset) and the third one performs full virtual backtracking. Unfortunately, it seems that it is more important to do early resetting on environment variables than on global stack variables, which favors the full variant. On the other hand, for many programs virtual backtracking does not have any effect at all, and the second pass through the roots is done in vain.

6 Acknowledgement

I would like to thank Micha Meier for many fruitful discussions. Thanks are also due to Alexander Herold and Emmanuel van Rossum for reading and commenting earlier versions of this paper.

References

[1] K.Appleby, M.Carlsson, S.Haridi, D.Sahlin
Garbage Collection for Prolog Based on WAM
SICS Research Report R86009B, 1986

[2] J.Barklund
A Garbage Collection Algorithm for Tricia
UPMAIL Technical Report 37B, December 1987

[3] Y.Bekkers, B.Canet, O.Ridoux, L.Ungaro
MALI: A Memory with a Real-Time Garbage Collector for Implementing Logic Programming Languages
Proc. 3rd Symposium on Logic Programming, 1986

[4] M.Meier, A.Aggoun, D.Chan, P.Dufresne,
R.Enders, D.Henry de Villeneuve, A.Herold,
P.Kay, B.Perez, E.v.Rossum, J.Schimpf
SEPIA - An Extendible Prolog System
Proc. of the XI. World Computer Congress'89 IFIP, San Francisco

[5] F.L.Morris
A Time- and Space-Efficient Garbage Compaction Algorithm
CACM. Vol.21 No.8, pp.662-665

[6] E. Pittomvils, M.Bruynooghe, Y.D. Willems
Towards a Real Time Garbage Collector for Prolog
Proc. Symposium on Logic Programming, 1985, pp. 185-198

[7] H.Touati
A Prolog Garbage Collector for Aquarius
Report No. UCB/CSD 88/443, August 1988, University of California, Berkeley

[8] David H. D. Warren
An Abstract Prolog Instruction Set
Technical Report tn309, SRI, October, 1983