Garbage Collection in B-Prolog

Neng-Fa Zhou
Department of Computer and Information Science
CUNY Brooklyn College
New York, NY 11210-2889, USA
zhou@sci.brooklyn.cuny.edu

Abstract

We present the garbage collector (GC) implemented in B-Prolog. It is based on the incremental GC algorithm proposed for the WAM by Older and Rummell, but modifications are made to adapt the algorithm to the ATOAM, the abstract machine adopted in B-Prolog. The criterion on when to invoke GC is refined to take determinism into account so that the effectiveness of GC is enhanced and the penalty on execution time is minimized. In addition, special efforts are made to handle those extensions of Prolog, such as delaying and constraint solving, that are supported in B-Prolog.

1 Introduction

B-Prolog was released in 1994. From the beginning, it was realized that a GC would be indispensable. However, it was postponed again and again because there were many other projects (delaying, constraint solving, a debugging tool, external language interfaces, and even a command editor) that were thought to be more interesting and urgent. Arguably, GC for CLP systems becomes less urgent nowadays than, say, ten years ago because computers nowadays have an order of magnitude more memory than ten years ago. Also, backtracking can sometimes be used effectively to reclaim space taken by garbage. Nevertheless, a GC is indispensable especially for those data-intensive applications such as CAD and language processing, and systems that run perpetually such as real-time applications. The DJ system[1], which is an interpreter for a Java-based constraint language implemented in B-Prolog, reveals that constraint solving may require far more memory than expected.

There are two types of heap GC algorithms, namely mark-and-compact [1] and copying [2, 3, 4] for Prolog. The incremental algorithm by Older and Rummell[2] is

[1] http://www.sci.brooklyn.cuny.edu/ zhou/dj.html

[2] Similar idea had been pursued by Touati and Hama [5]

especially interesting because it is much simpler than a global copying GC. It is a one-pass algorithm that collects only garbage in the top-most heap segment, i.e., the terms created after the latest choice point was created. There is a trade-off between time and space. On the one hand, if GC is not invoked often, some garbage may be buried under a choice point and may never be collected. On the other hand, if GC is performed frequently, the algorithm may significantly slow down the execution of programs. However, if the frequency of GC is adjusted properly, the algorithm can be as effective as a global GC and the time penalty can be kept very low. Another advantage of the algorithm is that the order of segments is preserved [3] without any extra cost.

The algorithm has to be adapted to the ATOAM, the abstract machine adopted in B-Prolog. The ATOAM, as a Prolog machine, is not much different from the WAM if only GC is concerned. Nevertheless, the non-standard functionalities such as delaying and constraint solving provided by B-Prolog need special attention. For example, frames on the stack in ATOAM no longer comprise a linear chronological chain if there are calls being delayed. This makes it more difficult to identify what terms are useless in ATOAM than in the WAM.

In Section 2, we briefly review the characteristics of the ATOAM memory architecture. In Section 3, we review the incremental algorithm by Older and Rummell and adapt it to ATOAM. In Section 4, we discuss criteria for determining when to invoke GC, and in Section 5 we evaluate the criteria and justify the one used in the implementation.

2 Memory Architecture of the ATOAM

The ATOAM, as a Prolog machine, is a variant of the WAM. It uses all the data areas used by the WAM. The *program area* stores the byte code instructions of loaded programs, the symbol table, and dynamic clauses created during program execution. The *heap* stores terms, mostly structural terms, created during execution. The register `H` points to the top of the heap. The *trail* stack stores those updates that must be undone upon backtracking. For each update, the address of the memory cell that was updated and the old content of cell are stored. The register `T` points to the top of the trail stack. The *control* stack stores frames associated with predicate calls. Unlike in the WAM where arguments are passed through argument registers, arguments in the ATOAM are passed through stack frames and only one frame is used for each predicate call. Each time when a procedure is invoked by a call, a frame is placed on top of the control stack unless the frame currently at the top can be reused. Frames for different types of predicates have different structures. Currently, predicates are classified into the following types: *determinate*, *nondeterminate*, *delayed*, and *tabled*. The register `AR` points to the current frame that is being accessed in execution and the register `B` points to the latest *choice point*, i.e., the frame for a nondeterminate predicate.

A determinate frame has the following structure:

```
A1..An     Arguments
AR:        Pointer to the parent frame
CP:        Continuation program pointer
BTM:       Bottom of the frame
TOP:       Top of the frame
Y1..Ym     Local variables
```

Where `BTM` points to the bottom of the frame, i.e., the slot for the first argument, and `TOP` points to the top of the frame, i.e., the slot just below that for the last local variable[3]. The field `BTM` was not in the original version [6]. It was introduced for handling delaying. Without delaying, the slot `BTM` would be useless since each time a call is returned, the top of the stack can be set to be either the top of the caller's frame or the top of the latest choice point. With delaying, however, the situation becomes a little more complicated. The chain of active frames connected by their `AR` slots may not be chronological. For this reason, only the top-most frame that is not a choice point can be released after the call is returned. The `BTM` becomes the new top of the stack when the current frame is released.

A choice point frame contains, besides the slots in a determinate frame, three other slots:

```
H:     Top of the heap
T:     Top of the trail
B:     Parent choice point
```

The slot `H` points to the top of the heap when the frame is allocated. As in the WAM, a new register, called `HB`, is used as an alias for `B->H`.

Figure 1 illustrates a situation in which the chain of active frames becomes non-chronological. The frames `f1` and `f2` are for delayed calls, `f3` is the latest nondeterminate frame, and `f4` is a determinate frame. The execution of `f4` was interrupted by an event that woke up `f1` and `f2`. The figure is a snapshot taken immediately after the two woken frames were added into the chain of active frames and `f2` became the current active frame.

3 The Incremental GC Algorithm

The algorithm by Older and Rummell is said to be incremental since it only collects garbage that resides in the top-most segment between `HB` and `H`. In this section, we review the algorithm, but in ATOAM's rather than WAM's terms, and discuss how to adapt the algorithm to handling delaying and constraint solving.

We use the terms *stack top-segment*, *trail top-segment*, and *heap top-segment* to refer to those frames between the top-most frame and `B`, those updates between `T`

[3]It is a convention in the literature that the stack is assumed to grow downwards

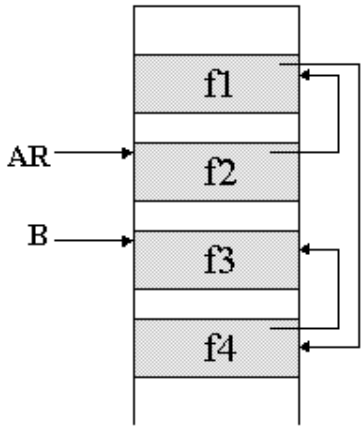

Figure 1: A non-chronological active frame chain.

and `B->T`, and those terms on the heap between `H` and `HB`, respectively. We will just use the term *top-segment* if the stack it refers to is clearly understandable from the context.

The algorithm looks like:

```
For each cell that refers to the heap top segment
   rescue the cell by copying its referred term to the to-area;
   redirect the cell to the eventual position of the referred term;
copy the terms in the to-area back to the heap;
```

The cells that have to be rescued include the argument slots and local variable slots in the stack top-segment, the local variable slots of the latest choice point, and the updated cells trailed in the trail top-segment. We assume that GC will be done at the entry points of nondeterminate predicates. Therefore, no temporary registers need to be rescued.

3.1 Correctness conditions

This algorithm assumes that the memory architecture meets the following requirements:

- All local variable slots must be initialized. Every GC requires the initialization of slots regardless of its type. Since in ATOAM arguments are passed through the stack and argument slots need not be initialized, the overhead of initialization is constantly lower than that in the WAM. Consider, for example, the following clause:

  ```
  p(X,Y):-q(X,Z),r(Z,S),t(S,Y).
  ```

In the WAM, three local variable slots, namely `Z`, `S`, and `Y` are necessary, but in the ATOAM, only one local variable slot, i.e., `Z`, is adequate since `X` and `Y` are arguments and the argument slot used by `X` is reused for the local variable `S`.

- All those cells that are older than the latest choice point `B` (including the arguments stored in `B`) and that refer to the heap top-segment must be trailed in the trail top-segment.
- No cell can be rescued more than once. This condition is important since after a cell is rescued, it will be set to point to the eventual position of the referred term, i.e., the position of the term after the term is copied back to the heap from the *to-area*.

To have the GC meet the third condition, we have to make sure that in the trail top segment all cells are different. This is guaranteed for standard Prolog programs since no variable can be instantiated more than once. For systems such as B-Prolog that allows non-logical updates of compound terms, constraints lists, and finite-domains, we have to pay special attention. We can use a time stamp for each cell to make sure only the first update with respect to a choice point is trailed. If we allow multiple updates for a single cell to be trailed, then we have to compact the trail every time GC is done.

3.2 Traversing stack frames

To rescue the cells in the frames in the top-segment, we can start from the current frame pointed to by the `AR` register and traverse the chain of active frames until we reach a frame that is not younger than the latest choice point. This method works for standard Prolog programs.

As mentioned above, with delaying, the chain of active frames connected by their `AR` slots may not be chronological. A frame's `AR` slot may point to a frame that happens to be younger than the frame.

There are two methods for traversing frames in the top-segment. One method is to add an extra slot into each frame and let it point to its previous frame. In this way, all frames in the top-segment can be reached from the top-most frame. The other method is to traverse the whole chain of active frames including those frames that are older than the latest choice point. To make this operation fast, we can introduce a tag, which is one bit, to each frame to tell where we can stop.

In addition to traversing the active frames, we must also traverse suspension frames. Since there is another chain that connects all such frames [7], it is not difficult to scan those suspension frames in the top-segment.

3.3 Rescuing terms

In BNR-Prolog [4], structures and lists are represented in contiguous memory cells, possibly with continuations. Thus, an iterative algorithm can be designed to copy

```
rescueTerm(addr,term){
   if (term is in heap top-segment){
        if (term has been rescued){
             *addr = eventualPos(*term);
         } else if (term is a reference but not free) {
             rescueTerm(addr,*term); /* dereference */
         } else if (term is free) {
             postpone rescuing it;
         } else {
           ....
         }
     else  /* term is not in heap top-segment */
         *addr = term;
     }
}
```

Figure 2: Rescuing terms.

terms from the top-segment to the *to-area*. For B-Prolog where terms are represented in the same way as in the traditional WAM, we need a recursive algorithm. Figure 2 shows the scheme of the algorithm. In the real implementation, a resizable stack is used instead of the stack in C.

The function call `rescueTerm(addr,term)` rescues a cell, where `addr` is the address and `term` is the content of the cell. If `term` has been rescued, i.e., it is a pointer to the *to-area*, then the cell is set to point to the eventual positon of the term. If `term` is a reference but not free, then dereference takes place. This is the so-called garbage-collection time dereference. Since term is guaranteed to be young that the latest choice point, it is safe to cut off reference chains.

As mentioned in [2, 3], the heap may grow because of GC if inner variables in structures are copied before the structures. Consider, for example, the call `p(X,f(X))` where the first argument of the call points to the argument of the structure. If the argument `X` is copied before `f(X)`, then the heap will grow by one cell after GC. To remedy this, some garbage collectors, such as [2, 3], use another bit for each heap cell to indicate whether the cell is in a structure. We take a different approach. Instead of using another bit for each heap cell, we postpone the copying of free variables until all structures and lists are copied. In this way, the heap is guaranteed not to grow after GC.

4 When Should GC be Done?

It is a daunting task to find a good answer to this question because the performance fluctuates for not only different programs but also for different settings for the system. There is a trade-off between time and space. If GC is not invoked often, some garbage may be buried under choice points and may never be collected. On the other hand, if GC is invoked too often (e.g., before each choice point is created), then GC would considerably slow down the execution of programs. A good criterion should lie in between these two extremes.

In [4], GC is invoked when the *proceed* instruction is executed that returns the call of the latest choice point and the size of the heap top-segment is greater than a threshold constant. The advantage of this strategy is that no scanning of stack frames is necessary. The disadvantage is that some garbage may never be collected even if the threshold is zero. The following example illustrates such a situation:

```
go:-g(X),t(X,Y),c(Y).
```

Suppose `g(X)` and `t(X,Y)` are determinate predicates that do not create any choice points and `c(Y)` is nondeterminate. `X` becomes garbage after `t(X,Y)` is returned but it cannot be collected because it will be buried under the choice point created by `c(Y)`.

A good criterion depends on what kinds of applications the system is designed for. For example, for time-critical applications such as real-time control system and interactive user interfaces, GC should be done frequently to ensure that the system never pause long. We discuss in the following of this section what a criterion should be used to avoid heap overflows. The goal is to use an incremental GC to achieve what is achieved with a global GC.

We use the following function:

$$H_{top} > C \times \frac{H_{avail}}{H_{max} - H_{avail}}$$

where H_{top} is the amount of memory in the heap top-segment, H_{avail} is the amount of available heap space, H_{max} is the total amount of space allocated to the heap, and C is a constant. At the entry point of a nondeterminate predicate, GC is invoked if H_{top} is greater than the threshold. This function ensures that GC will be invoked more frequently with the amount of heap space becoming less and less compared with the amount of consumed space.

Most predicates in typical Prolog programs are *globally determinate* in the sense that no call to them leaves a choice point on the stack after it returns. Choice points are necessary for them but they only survive a short time. For these predicates, GC should be done less frequently than for nondeterminate predicates since garbage is unlikely to be buried by their choice points for long. To this ends, we use a bigger threshold function for globally determinate predicates than that for nondeterminate ones.

Experiments show that GC is still done too frequently even if a big constant `C` is used in the threshold function. The culprit is in that once a segment becomes eligible for GC, it will always be so if the heap does not shrink after GC. The following example illustrates this situation:

```
p([X|Xs]):-q(X),!,p(Xs).
p([]).
```

Suppose `q(X)` is nondeterminate and the top-segment is bigger than the threshold. The cut following `q(X)` discards the choice points created by `q(X)`. Each time `q` is called, the same segment will be garbage-collected. To prevent GC from being repeatedly invoked for the same segment, we remedy the strategy so that no segment can be garbage-collected consecutively more than once. This strategy may leave some garbage uncollected, but can significantly reduce the number of GCs performed.

5 Performance Evaluation

As mentioned in [4], the choice of benchmarks for evaluating GC strategies is always problematic. Some programs are determinate for which no choice point is necessary and GC will be postponed until before the heap space runs out. Some other programs involve a lot of backtracking and thus there are tremendous opportunities for GC to be called. We choose three programs: *bp-compiler* the B-Prolog compiler, *boyer* a theorem prover, and *magic sequence* a finite-domain constraint program.

Table 1 compares three different strategies: `GC-no`, `GC-all`, and `GC-part`. In `GC-no`, no GC is done at all. In `GC-all`, GC is done at the entrance of every nondeterminate predicate. And in `GC-part`, GC is done at the entrance of every nondeterminate predicate unless the top-segment had just been garbage-collected. Undoubtedly, `GC-all` invokes GC most frequently and requires the least amount of space. The column `#GCS` shows the number of times GC is performed. For *magic sequence*, two values *H+L* are shown, where `H` is the heap space and `L` is the local stack space required.

Program	Space (bytes)			#GCS	
	GC-no	GC-all	GC-part	GC-all	GC-part
bp-compiler	926,212	383, 016	406,260	42,149	13,893
boyer	575,788	224,316	224,316	144,857	23,106
magic seqence	1,620+7,744	1,272+5,404	1,444+5,616	2,975	1,324

Table 1: Comparison of three strategies.

In terms of CPU time, `GC-all` is over ten times slower than `GC-no`.

In the above comparison, no threshold function is used. Table 2 compares the number of GCs performed on different settings. Each row represents a different setting for the constant `C` in the threshold function and each column represents a different setting for H_{max} (H_{max} is set to be the number times the amount required by `GC-part` shown in table 1). The constant used in the threshold function for globally determinate predicates is ten times the `C` used for nondeterminate predicates. As in `GC-part`, GC is prohibited to be done on one segment consecutively. Each entry shows the number GCs performed. It is hard to say which one is better. The overhead on execution time is too low to be measurable in all the executions. In the real implementation, `C` is set to be `1000`.

	1.1	1.5	2	4
C=500	65	53	37	11
C=1000	51	25	15	5
C=2000	23	11	9	5

Table 2: Comparison of number of GCs performed on different settings.

Table 3 shows the maximum number of postponed inner variables. It is faster to

bp-compiler	boyer	magic
4,622	30	18

Table 3: Maximum number of postponed inner variables.

postpone copying inner variables than to use a bit vector to mark them. Concerning space requirements, postponing is not worse than marking since the number of postponed variables is usually small compared with the size of the GC area.

6 Concluding Remarks

Although GC has become a mature field that is being taught in many universities, building a working GC is far more difficult than is usually perceived. There may exist bugs in the original system that appear only with the existence of GC. The conditions for correctness of GC may turn out to be false. We thought a GC based on the incremental algorithm could be completed in one month if not in one week. We eventually spent far more time than we estimated.

We presented only a GC for the heap. There is a GC for each other area in B-Prolog including the trail, the stack, and the program area. The necessity of a GC for the control stack is specific to B-Prolog since some useless frames may stay

on the stack because of delaying. All those frames that are younger than the latest choice point and are not connected by either the chain of active frames or the chain of suspension frames are useless and are collected as garbage.

The strategy for determining when to invoke GC is a central research issue. We proposed a new strategy that improves the one proposed by Older and Rummell. It uses a threshold function to judge whether GC is worthwhile and it prohibits GC from being done on one segment consecutively. This strategy cannot be claimed to be the best, but is satisfactory since for the B-Prolog compiler it collects 90 percent of the garbage with almost no overhead on execution time.

It is not difficult to find a program for which the strategy would perform badly. Consider, for example, the following predicate.

```
p([X|Xs]):-generateLittleGarbage(X),p(Xs).
p(_).
```

The call `generateLittleGarbage(X)` generates a little garbage. After each iteration, a new choice point is created that buries the garbage generated in the previous iteration. In this way, garbage will be accumulated little by little and GC will not be done until too late. For this type of programs, global GC is still inevitable.

References

[1] Appleby, K., Carlsson, M., Haridi, S., Sahlin, D.: Garbage Collection for Prolog based on WAM, *Communications of the ACM*, Vol.31, No.6, pp.719-741, June, 1988.

[2] Bevemyr, J. and Lindgren, T.: A Simple and Efficient Copying Garbage Collector for Prolog, UPMAIL Technical Report, No.87, ISSN 1100-0686, 1994.

[3] Demoen, B., Engels, G., and Tarau, P.: Segment Order Preserving Copying Garbage Collection for WAM Based Prolog, *Proc. ACM/SAG*, 1996.

[4] Older, W.J. and Rummell, J.A.: An Incremental Garbage Collector for WAM-Based Prolog, *Proc. of JICSLP*, pp.369-383, 1992.

[5] Touati, H. and Hama, T.: A Light-Weight Prolog Garbage Collector, *Proc. FGCS*, pp.922-930, 1988.

[6] Zhou, N.F.: Parameter Passing and Control Stack Management in Prolog Implementation Revisited, *ACM Transactions on Programming Languages and Systems*, Vol.18, No.6, 752-779, 1996.

[7] Zhou, N.F.: A Novel Implementation Method of Delay, *Proc. JICSLP'96*, pp.97-111, MIT Press, 1996.