UPMAIL Technical Report No. 37B

A Garbage Collection Algorithm for Tricia

Jonas Barklund

Uppsala Programming Methodology and Artificial Intelligence Laboratory
Computing Science Dept., Uppsala University
P. O. Box 520, S-751 20 Uppsala, Sweden
+46–18–18 25 00

Electronic mail: JONAS@AIDA.UU.SE

Abstract. We present a new algorithm for garbage collection of the term stack in David H. D. Warren's abstract Prolog machine (WAM). The algorithm exploits the possibilities of using a large address space, with special attention paid to virtual memory behaviour. The time required for the algorithm depends linearly on the size of the areas in which garbage is collected. Pseudo-code is given for the whole algorithm. It has been fully implemented in Tricia, a Prolog developed at UPMAIL for the DEC-2060 computer.

The research reported herein was supported by the National Swedish Board for Technical Development (STU).

1. PROBLEM

In languages with dynamic memory allocation it often happens that an instance of a data structure is not needed for the rest of the computation. Assuming that virtual memory is a limited resource (although it may be very large) it is desirable to be able to re-use such garbage structures. Placing the responsibility of returning garbage on the programmer is a too heavy burden for non-trivial programs so it should be mechanized and invisible to the programmer.

In modern abstract Prolog machines, such as WAM [19] or ZIP [8] some garbage collection is built into the machine itself. On backtracking, allocated memory in the environment and term stacks is immediately reclaimed. In addition, environment stack memory is reclaimed during deterministic execution and tail-recursive calls. These and other peculiarities of Prolog machines have to be considered when writing a garbage collector, or it is likely to be inefficient in time and/or space.

A garbage collection algorithm for WAM or a similar machine must have the following properties:

1. The garbage collected term stack must be compactified.
2. The order of objects in the term stack must be preserved, at least within each term stack segment. (A *term stack segment* is defined as the part of the term stack allocated between two subsequent choice points. *Environment stack segment* and *trail segment* are definied analogously.) This criterion rules out many LISP garbage collectors.
3. Non-atomic objects may be of arbitrary size.
4. There may be pointers with different tags to the same address. The standard example is a list whose head is an unbound variable. Both variable and list-tagged pointers to the first word of the list object may appear. This can be avoided, at a price, by always moving unbound variables out of lists and other compound terms. There may be other reasons than garbage collection to do this [7].

The ideal garbage collection would progress in parallel with the computation, without interfering with it. This is sometimes referred to as real-time garbage collection. It requires using several processors or simulation by time-sharing on one processor [10, 18]. The algorithm described below is not a real-time garbage collector. Pittomvils' discussion [15] about the modifications needed for real-time use mostly carries over to our algorithm.

One aspect of garbage collection which is rarely discussed is when to do garbage collection. To keep allocating memory until it runs out, and then do garbage collection is a simple way, but the following garbage collection may take very long time for a large virtual memory. Also locality of references may be low while data is spread over a large area. In section 11 we discuss another alternative for when, and where, to do garbage collection which takes the life time of data into account [12].

2. SOLUTION

We define a mark-compact-update algorithm for garbage collection in a Prolog machine. The garbage collector works in three phases: marking, compacting, and updating. In the terms of Cohen's survey of garbage collection algorithms [9] our garbage collector could be characterized as a 'sliding, compacting algorithm for varisized cells without using a break table'.

The idea is to take advantage of a large virtual memory during garbage collection by using a temporary area, as large as the area being garbage collected. This area is used to mark objects in the term stack, by storing pointers to them. These pointers are then used during the subsequent compacting and updating phases, and finally removed. Assuming that we have this perhaps large amount of virtual memory available, we can make the algorithm more efficient, since we do not have to worry about some complicated and/or time-consuming techniques (e.g., for compaction to keep track of pointers [13, 17]). In Cohen's survey, the technique to use a congruent temporary area for marking and updating does not seem to be represented, probably because there are few algorithms written to benefit from the use of a large memory.

Since both the temporary area and the memory being garbage collected can be expected to be in virtual memory, it is important to localize references as much as possible. We think we achieve this, at least in the compacting and updating phases.

3. THE MACHINE

The machine for which this algorithm is designed is a somewhat revised version of WAM [3, 6]. The revisions relevant to the term stack GC are as follows:

- An extra register F is added, containing flags. This register is always tested when a predicate is entered and an exception occurs if any flag in it is set. One of the flags is *GCFLAG*. When the machine is almost out of memory this flag is set. Then on the next predicate entry garbage collection will occur. The advantage is that garbage collection will only start when the machine is in a well-defined state. The disadvantage is that there is a slim chance that a huge data structure gets allocated before the next predicate entry, so there is no room for it. To our knowledge, this has never happened in our implementation and we do not consider it a serious problem.
- Tagged pointers (as opposed to tagged structures) are used. This is not a revision of WAM but certainly affects the GC algorithm. It is possible to extend this algorithm to cope also with untagged pointers. We have done this extension but do not present it here because the algorithm becomes uglier and because we do not use untagged pointers anymore in our implementation.
- The arity of a choice point is always accessible from the choice point. Let us assume, for the sake of simplicity, that choice points contain a new field R which is the number of temporary variables stored in the choice point. In some implementations this number may be available somewhere else, the algorithm can be trivially modified to take this into account.
- Environments contain a field N, for marking and updating. It is used to keep track of the number of variables in the environment which have been marked so far. An 8-bit field should suffice for reasonable programs. (In our implementation we could overlay the left half of the *CE* field with the N field. It normally contains a value k, so all values for the N field in the algorithms below were incremented by $k + 1$.)

In our implementation, the following restrictions have also influenced the garbage collector:

- A machine word can contain two disjunct fields: a tag and an address.
- The memory has no internal structure (e.g., segments of some fixed size) but is just a sequence of words with monotonically increasing addresses. To be fully utilised, Pittomvils' algorithms require a segmentation of memory [15].

An *object* is represented by a word containing a *tag* and a *value*. The interpretation of the value depends on the tag. It is usually an address but is sometimes data. The following are the objects which may appear in the term stack and their internal structure.

- *Variables.* If the tag is `$SVA`, the value is an address in the environment stack. Such objects cannot appear in the term stack and the algorithm does not handle them. If the tag is `$VAR`, the value is an address in the term stack. The word at this address contains an object.
- *Closures (constrained variables).* If the tag is `$CLO`, the value is an address in the term stack. The word at this address contains an object which is the value of the constrained variable and the second word contains a goal, represented as an ordinary term.
- *Small integers.* If the tag is `$SIN`, the value is data. Usually the lower 18 bits is a signed (two's complement) integer datum, but sometimes all 30 bits are interpreted as an unsigned integer.
- *Big integers.* If the tag is `$INT`, the value is the address of a word containing a 36-bit signed integer datum.
- *Floating point numbers.* If the tag is `$FLO`, the value is the address of a word containing a 36-bit floating point number.
- *Characters.* If the tag is `$CHR`, the value is data: an ASCII character code and font information.
- *Atoms and functors.* If the tag is `$SYM` or `$NIL`, the value is an address in the code area.
- *Lists.* If the tag is `$LST`, the value is an address of a two-word block in the term stack, each containing an arbitrary object.
- *Other compound terms.* If the tag is `$STR`, the value is an address of a $(n + 1)$ word block in the term stack. The first word contains a functor (an object whose tag is `$ATM`) and the following n words contain arbitrary objects.
- *Strings.* If the tag is `$SST`, the value is data. Moreover, the k next words also contain data, representing an ASCII string. The size k is contained in the data.

An object with any other tag is treated exactly like an atom.

4. NOTATION

We will use the same names of registers in the machine and naming of fields in choice points and environments as Warren [19].

By $@A$ we denote the contents of the word pointed to by A.

Let GCB be either the choice point which was most recent at the previous garbage collection or the oldest choice point reached since by a `trust`, `trust_me_else` or *cut* operation. Let LB be the choice point for which we will collect garbage in the term stack above $LB.H'$. We define the *white* area as the part of the term stack below (and including) $GCB.H'$. We define the *dark* area as the part of the term stack above $GCB.H'$. The dark area is divided into a *grey* area below (and including) $LB.H'$ and a *black* area above $LB.H'$. (Intuitively, no garbage can be collected in the white area, but the dark area contains garbage. Garbage is collected in the black area, but the grey area will still contain garbage after the collection.)

We assume that the following primitive predicates and functions are defined:

- $permanent\_variable(Env, I)$ returns the address of the permanent variable I in environment Env.
- $env\_size(Cont)$ returns the number of living variables in the environment that $Cont$ will return to.
- $argument(Chpt, I)$ returns the address of the saved temporary variable I of choice point $Chpt$.
- $size\_of\_object(P)$ returns the size in words of the surface level of the object pointed to by the tagged pointer P (e.g., 2 for lists and $n+1$ for other compound terms of arity n).
- $tag(P)$ returns the tag of the tagged pointer P.
- $address(P)$ returns the address part of the tagged pointer P.
- $typed(Tag, Address)$ returns a tagged pointer with tag field Tag and address field $Address$.
- $temp\_area(P)$ where P is a pointer to a word in the dark area, returns a pointer to the corresponding word in the temporary area.
- $arity(F)$ returns the arity of the functor F.
- tb is the address of the first word in the temporary area.
- tz is the address of the first word after the temporary area.

5. TOP LEVEL

The algorithm collects garbage in the part of the term stack allocated since the creation of some arbitrary choice point LB. This flexibility is used when implementing 'garbage cut'. It will perform garbage collection of cells allocated since the creation of an arbitrary choice point.

Before the actual garbage collection starts, the current state is saved in a choice point. Because we know that the current state is in the most recent choice point there is no need to treat the topmost segments and the current goal specially. After the garbage collection it is important that the current state is restored from the choice point, since the new term stack and trail pointers are only reflected in the choice point.

The marking phase is a recursive algorithm, adapted to the peculiarities of Prolog in a way similar to that of Pittomvils [15]. First, every term accessible from goals older than LB is marked. This information is available on the trail. Secondly, terms accessible from more recent goals, beginning with the most recent, are marked. Apart from the exception noted above, we think the algorithm is correct and optimal in the sense that it marks all objects needed later in the computation, but does not mark any objects to which there are no references, or to which all references will disappear on backtracking. To mark even fewer objects it would be necessary to analyze the code to be executed and/or improve compilation techniques further. That is beyond the scope of this paper.

The compacting phase scans through the temporary area, moving all living objects to lower addresses while storing forward pointers to the moved objects in the temporary area.

The updating phase replaces all addresses of objects which have been moved by their new addresses. This involves another scan through the temporary area.

Algorithm G (*Garbage collection*). The current goal/state is saved in a choice point. If garbage collection is only started when a predicate is entered, all information needed about arity, etc., should be available. The marking, compacting, and updating phases are applied and finally the state is restored from the most recent choice point.

G1. [Initialize.] Choose LB, allocate a new choice point and save the current state in it.

G2. [Mark live terms.] Mark all live terms with a pointer from the temporary area (algorithm M).

G3. [Compactify live terms.] Copy all live terms towards the beginning of the heap, keeping them in order (algorithm C).

G4. [Update pointers.] Update all pointers to copied terms to reflect their new addresses (algorithm U).

G5. [Reset temporary area.] Set every word in the temporary area to 0.

G6. [Restore state.] If LB was chosen as GCB in step G1, then set $GCB \leftarrow LB$. Restore the state from the choice point allocated in step G1 and deallocate it. Finally terminate the algorithm.

Program G (*Garbage collection*).

```
procedure garbage collection:
  choose_lb;        -- See section 10.
  save_state;       -- "Try" this and then "trust" true.
  marking_phase;
  compacting_phase;
  updating_phase;
  for i:= tb to tz-1 do @i := 0;
  if gcb_chosen then gcb := lb;
  restore_state.    -- "Fail" to saved state above.
```

6. MARKING PHASE

In this algorithm, *marking an object* means to place a tagged pointer to its first word from the corresponding word in the temporary area. The purpose of the marking phase is to mark every object in the term stack segments between $LB.H'$ and H which is accessible from some living goal. When marking a compound term of arity k, pointers are placed to the term itself as well as to the following k words. The latter pointers will be tagged as variables.

If the head of a list is an uninstantiated variable, its address will be the same as that of the surrounding list. If there are references both to the list and the variable in its head, the pointers marking them in the temporary area will share the same word. We solve this by always letting a list pointer (in fact any pointer) take precedence over a variable pointer. If we attempt to mark a list and the temporary area word is already occupied by a variable pointer, we overwrite it with a list pointer and mark the parts of the list. If we attempt to mark a variable and the temporary area word is already occupied by a list pointer, we consider the variable to be already marked.

Where to start marking

All living goals are represented by argument registers and environments saved in choice points. The terms contained in a goal must have been created before the creation of the goal itself. However, if a goal contained logical variables when it was created, these variables may subsequently have been bound to terms created after the goal itself. But because of the way the WAM is constructed, all such instantiations of variables to more recent terms are recorded in the trail.

As a consequence of this, to mark all terms between $LB.H'$ and H accessible from living goals, we should scan the trail from $LB.TR'$ to TR, marking from all variables below $LB.H'$ which are bound to terms above $LB.H'$. Moreover, we should mark from all saved goals created after LB.

The most recently saved goal (actually the current goal, since it is saved when the garbage collection is started) is in B. Saved argument registers can be found in $B.A_1$ to $B.A_r$ where r is the current value of $B.R$. The saved environment is in the word $B.BCE$, with number of living variables accessible through $B.BCP$. Successively older goals are saved in $B.B'$, $B.B'.B'$ etc.

Marking a goal

Saved goals will share parts of their environment, but may have different ideas about the number of living variables in it. Because of this, it is not necessary to mark from the whole environment every time it is entered by the marking process.

To keep track of which parts of an environment have been marked from, each environment contains a field N which should be initialized to -1 when an environment is created. This field has the same purpose as

the `visit` and `number` fields in the OTM algorithm [15]. A negative value of N indicates that the environment has not yet been marked. Zero or a positive value indicates that the environment has been visited, and is the number of variables which have been marked so far. When the marking process enters an environment Env with a corresponding continuation $Cont$, it examines the value of $Env.N$ and computes the number of living variables in Env, as described above.

1. If $Env.N < 0$, all $env\_size(Cont)$ living variables in Env are marked recursively. $Env.N$ is set to $env\_size(Cont)$ and the marking process proceeds with the environment pointed to by $Env.CE$.
2. If $Env.N \geq 0$ it means that we have (partially) marked the environment before. It may be the case that the more recent goal which previously marked from the environment had fewer living variables. If $Env.N < env\_size(Cont)$ this was the case; the extra $env\_size(Cont) - Env.N$ variables are marked from and $Env.N$ is set to $env\_size(Cont)$. Marking of the goal can then terminate, since the other environments of the goal have already been fully marked.

Partial backtracking

Before we start marking a goal saved in a choice point $Chpt$, *partial backtracking* to $Chpt$ is done. This means to reset the variables pointed to by trail entries above $Chpt.TR'$, unless they have already been marked. When a goal was saved in $Chpt$, these variables were uninstantiated, so if no more recent goal references them, the structures they are instantiated to can be reclaimed. Since the variables will not be referenced until backtracking reaches $Chpt$, any legal object could be left in them, but if they are changed into uninstantiated variables (i.e., self references in our implementation), the trail entry can also be removed. This is similar to 'virtual backtracking' [5] and equivalent to 'early reset of variables' [1].

As mentioned above, marking terms referenced from below $LB.H'$ is also done by scanning the trail. Unfortunately, it turns out that partial backtracking and marking from below $LB.H'$ cannot be done simultaneously. The partial backtracking would reset variables referred from parts of goals created before LB. (This observation was pointed out to us by Mats Carlsson before we experienced the problem ourselves).

Algorithm M (*Marking phase*). All terms between $LB.H'$ and H are marked by placing a tagged pointer in the corresponding word of the temporary area. The links between choice points are reversed. The N field of each environment above LB is set to the maximum of the number of live variables in the environment accessible from any goal. Some details (e.g., marking for each type of object) are omitted but can be found in program M.

M1. [Mark from old goals.] For each trail entry from $LB.TR' + 1$ to TR, if it points to a term stack variable below $LB.H'$ or an environment stack variable below LB, then mark from the bound variable.

M2. [Mark from more recent goals.] Set $(Chpt, Chpt', B.B') \leftarrow (B, B.B', LB.B')$.

M3. [Termination?] If $Chpt \leq LB$ the marking phase terminates.

M4. [Mark subgoal.] Mark from $argument(Chpt, I)$, for $I = 1, \ldots, R$ where R is the arity of the saved subgoal. Set $(Env, Cont) \leftarrow (Chpt.BCE, Chpt.BCP)$.

M5. [Mark rest of goal.] If $Env \leq LB$, then go to step M7. Otherwise, set $M \leftarrow Env.N$, and set $M' \leftarrow env\_size(Cont)$.

M6. [Mark environment.] If $M < 0$ then mark from $permanent\_variable(Env, I)$ for $I = 1, \ldots, M'$, set $Env.N \leftarrow M'$ and $(Env, Cont) \leftarrow (Env.CE, Env.CP)$; then return to step M5. (We have not visited the environment before and mark from all live variables.) Otherwise, mark from $permanent\_variable(Env, I)$ for $I = M + 1, \ldots, M'$, and set $Env.N \leftarrow M'$. (We have visited the environment before but there may be more live variables. We do not have to continue further in the chain.)

M7. [Partially backtrack.] For each trail entry from $Chpt.TR'$ down to $Chpt'.TR' + 1$, if it points to an unmarked variable above $LB.H'$, make it uninstantiated and reset the trail entry.

M8. [Proceed to next goal.] Set $(Chpt, Chpt', Chpt'.B') \leftarrow (Chpt', Chpt'.B', Chpt)$, and return to step M3.

Program M (*Marking phase*).

```
procedure marking_phase:
  mark_old_goals;
  mark_young_goals.
```

```
procedure mark_old_goals:
  for J := LB.TR'+1 to TR do
   {Aj := address(@J);
    case tag(@J) of
     {$SVA:       if Aj ≤ LB then mark(@Aj);
      $VAR, $CLO: if Aj ≤ LB.H' then mark(@Aj)}}.

procedure mark_young_goals:
  Chpt := B; Chpt' := B.B'; B.B' := LB.B'
  while Chpt > LB do
   {-- Marking saved argument registers.
    for I := 1 to Chpt.R do mark(@argument(Chpt,I));
    -- Marking rest of goal.
    mark_environment(Chpt.BCE,Chpt.BCP);
    -- Partial backtracking.
    for J := Chpt'.TR'+1 to Chpt.TR' do
     {Aj := address(@J);
      if black(Aj) then
        if @temp_area(Aj) = 0 then {@Aj := @J, @J := 0}}
    -- Flip B'-pointers and process next environment.
    exchange(Chpt,Chpt'.B');
    exchange(Chpt,Chpt')}.

procedure mark_environment(Env,Cont):
  while Env > LB and Env.N < 0 do
   {for I := 1 to env_size(Cont) do mark(@permanent_variable(Env,I));
    Env.N := env_size(Cont);
    Cont := Env.CP; Env := Env.CE};
  if Env > LB then
   {for I := Env.N+1 to env_size(Cont) do mark(@permanent_variable(Env,I));
    Env.N := env_size(Cont)}.

procedure mark(P):  -- Mark object P.
  Ap := address(P);
  case tag(P) of
   {$INT, $FLO, $SST: if black(Ap) and @temp_area(Ap) = 0 then temp_area(Ap) := P;
    $VAR:             if black(Ap) and @temp_area(Ap) = 0 then
                       {temp_area(Ap) := P; mark(@Ap)};
    $CLO, $LST:       if black(Ap) and @temp_area(Ap) ≠ P then
                       {temp_area(Ap) := P; mark(@Ap); mark(typed($VAR,Ap+1))};
    $STR:             if black(Ap) and @temp_area(Ap) ≠ P then
                       {temp_area(Ap) := P; for I := 1 to arity(@Ap) do
                                              mark(typed($VAR,Ap+I))}}.

function black(A): -- Return true if address in black area.
  LB.H' < A and A ≤ H.
```

7. COMPACTING PHASE

This phase copies all living term stack objects above $LB.H'$ towards $LB.H'$, removing space between objects while keeping their order.

After the marking phase, each word in the temporary area is either zero or contains a tagged pointer into the term stack. The address contained in such a pointer is redundant, since it can be computed from the address A of the temporary area word containing the pointer, as $A - tb + LB.H'$.

Let a temporary segment be the part of the temporary area corresponding to one heap segment. Each temporary segment, beginning with the oldest, is scanned, copying the top-level of each object pointed to as close to $LB.H'$ as possible. This preserves the order between the objects. When an object is copied, the pointer in the temporary area is updated with the new address of the object. When there is a pointer to a compound term as well as to some subterm properly contained in it (e.g., a variable), it is important to

copy only the compound term. The pointer to the subterm is updated with the new address of the subterm but the subterm itself is not copied again. After the compaction of one temporary segment, the term stack pointer for the segment is updated to reflect the new size and position of the term stack segment.

Algorithm C (*Compacting phase*). The term stack pointer s is set to $LB.H'$. For each temporary segment, beginning with the oldest, the following actions are taken. The term pointer t is set to the beginning of the segment. t is advanced until a tagged pointer is found in $@t$. The word pointed to is pushed on s and the address part of $@t$ is set to the address part of s. If $@t$ points to a string (our only uninterpreted data occupying more than one word) of length k, the words $t+1$ to $t+k-1$ are also pushed on s. t is incremented by 1 or k and the process is repeated until t reaches the end of the segment. The saved heap pointer is set to s and the process continues with the next temporary segment, until the end of the temporary area is reached.

C1. [Initialise.] Set $S \leftarrow LB.H'$, $Chpt \leftarrow LB.B'$, and $Chpt' \leftarrow LB$.

C2. [Termination?] If $Chpt' < B$, set $t \leftarrow temp\_area(Chpt'.H'+1)$; then proceed to step C3. Otherwise, the compacting phase terminates.

C3. [Find non-garbage.] If $T > temp\_area(Chpt.H')$, then set $(Chpt.H', Chpt, Chpt') \leftarrow (S, Chpt.B', Chpt)$ and return to step C2. Otherwise, if $@T = 0$, increment T by 1 and repeat this step.

C4. [Copy non-garbage.] If $tag(@T) =$ \$SST, then set $K \leftarrow size\_of\_object(@T)$, otherwise set $K \leftarrow 1$. Set $@(S+I) \leftarrow @(@T+I-1)$, for $I = 1, \ldots, K$. Set $address(@T) \leftarrow address(S+1)$, increment S and T by K and return to step C3.

Program C (*Compacting phase*).

```
procedure compacting_phase:
  S := LB.H'; Chpt := LB.B'; Chpt := LB;
  while Chpt' < B do
   {T := temp_area(Chpt'.H'+1);
    while T ≤ temp_area(Chpt.H') do
      if @T = 0 then T := T+1 else
       {if tag(@T) = $SST then K := size_of_object(@T) else K := 1;
        for I := 1 to K do @(S+I) := @(@T+I-1);
        address(@T) := address(S+1);
        S := S+K ; T := T+K};
    Chpt.H' := S; Chpt' := Chpt; Chpt := Chpt.B'}.
```

8. UPDATING PHASE

The purpose of this phase is (i) to modify the address part of all pointers to objects in the black area (we call this *updating* a pointer) and (ii) to clean up the state of the machine after the two previous phases.

Pointers which need to be updated could be found in exactly the same way as in the marking algorithm. Since, however, we have already done the job of finding all referenced objects in the black area, and can access all of them from the temporary area, it is not necessary to use a recursive algorithm again. Another reason not to use a recursive algorithm is that we cannot tell if a pointer has already been updated.

We also have to do the following to clean up the state of the machine:

- Set $Env.N$ to -1 for every environment Env.
- Reverse the chain of choice points again. Actually, the chain was reversed in the marking phase because we want to access the choice points starting with the oldest during this phase. This is to simplify the excising of the trail below.
- Remove trail entries to variables which have been collected. We copy trail entries downwards, beginning from the oldest choice point and finishing with the most recent. This implies that it is necessary to update the $Chpt.TR'$ field of each choice point $Chpt$ above LB.

$Chpt.H'$ has alrady been updated, for all choice points $Chpt$ above LB.

Algorithm U (*Updating phase*). All pointers to the black area are updated, environments and choice points are reset to their normal state, the trail is excised and the temporary area is cleared. To 'update A' means: if $LB.H' < @A \leq H$ then set $address(@A) \leftarrow address(@temp\_area(@A))$.

U1. [Update goals.] Set $(Chpt, Chpt', LB.B') \leftarrow (LB.B', LB, B.B')$ and $(S, T) \leftarrow (Chpt'.TR' + 1, LB.TR')$.

U2. [Update subgoal.] If $Chpt' \geq B$, set $I \leftarrow tb$ and go to step U7. Otherwise, update $argument(Chpt, I)$ for $I = 1, \ldots, B.R$ and set $Env \leftarrow Chpt.BCE$.

U3. [Update environments.] If $Env.N \geq 0$, update $permanent\_variable(Env, I)$ for $I = 1, \ldots, Env.N$, set $Env.N \leftarrow -1$ and $Env \leftarrow Env.CE$; then repeat this step.

U4. [Update trail segment, at end?] If $S > Chpt.TR'$, set $Chpt.TR' \leftarrow T$ and $(Chpt, Chpt', Chpt.B') \leftarrow (Chpt.B', Chpt, Chpt')$; then return to step U2.

U5. [Removed entry?] If $@S = 0$, then increment S by 1 and return to step U4.

U6. [Copy entry, maybe update.] Increment T by 1 and set $@T \leftarrow @I$. If $@T$ points to the environment stack and $@T \leq LB$, or if T points to the term stack and $T \leq LB.H'$, then update $@T$. Otherwise, if T points to the term stack and $T > LB.H'$, then update T. In any case, increment S by 1 and return to step U4.

U7. [Update internal term stack pointers.] If $I \geq tz$ then terminate the compacting phase. Otherwise, if $@I = 0$, then increment I by 1 and repeat this step.

U8. [Update object.] Inspect $tag(@I)$. If its value is `$VAR`, update $@I$ and increment I by 1. If its value is `$LST`, update $@I$ and $@I + 1$ and increment I by 2. If its value is `$STR`, update $@I + J$ for $J = 0, \ldots, arity(@@I)$ and increment I by $arity(@@I)$. Finally, if the tag is `$INT`, `$FLO`, or `$SST`, then increment I by 1. In any case, return to step U7.

Program U (*Updating phase*).

```
procedure updating_phase:
  update_goals;
  update_term_stack.

procedure update_goals:
  Chpt' := LB; Chpt := LB.B'; LB.B' := B.B';
  S := Chpt'.TR'+1; T := LB.TR;
  while Chpt' < B do
   {-- Update argument registers.
    for I := 1 to Chpt.R do update(argument(Chpt,I));
    -- Update rest of goal.
    update_environment(Chpt.BCE);
    -- Update trail segment.
    update_trail(Chpt.TR',S,T);
    Chpt.TR' := T;
    -- Flip B'-pointers and process next goal.
    exchange(Chpt,Chpt');
    exchange(Chpt,Chpt'.B')}.

procedure update_environment(Env):
  while Env.N >= 0 do
   {for I := 1 to Env.N do
      update(permanent_variable(Env,I));
    Env.N := -1; Env := Env.CE}.

procedure update_trail(End, var S, var T);
  -- Update trail entries and trailed variables pointing to black area.
  while S ≤ End do
   {if @S > 0 then
     {T := T+1; @T := @S;
      case tag(@T) of
       {$SVA:       if address(@T) ≤ LB then update(@T);
        $VAR, $CLO: if address(@T) ≤ LB.H' then update(@T) else update(T)}};
    S := S+1}.

procedure update_term_stack:
```

```
  I := tb;
  while I < tz do
    if @I = 0 then I := I+1 else
      case tag(@I) of
       {$VAR:             update(adress(@I)); I := I+1;
        $INT, $FLO, $SST: I := I+1;
        $CLO, $LST:       for J := 0 to 1 do update(adress(@I)+J); I := I+2;
        $STR:             for J := 1 to arity(@@I) do update(adress(@I)+J);
                          I := I+arity(@@I)}.

procedure update(A): -- Update the pointer at address A.
  if black(@A) then address(@A) := address(@temp_area(@A)).
```

9. PROPERTIES OF THE ALGORITHM

Our marking algorithm is like a mixture of the QOIM and serial OIM algorithms [15]. It differs from the (optimal) OIM algorithm in not marking objects below $LB.H'$. Still, there are pointers from below $LB.H'$ to above. The only possible such pointers are variables that have become bound after the creation of LB. However, the trail rules of WAM guarantee that all such bindings must have been recorded in the trail. So all pointers from below $LB.H'$ are in turn pointed to from trail entries above $LB.TR'$. Since, (as Pittomvils notices [15]) when the term stack segments below $Chpt.H'$ have been garbage collected, they cannot get more garbage until backtracking (or a cut) reaches below $Chpt$. This means that everything referred to from below $LB.H'$ must be considered non-garbage. This makes the algorithm non-optimal, since some parts of the goals below $LB.H$ may not be accessible.

Unfortunately, Pittomvils' example [15 pp. 191–192] of a program where a OIM and a QOIM garbage collector will behave differently, does not say very much. A compiler has the choice of allocating a temporary variable (which will get bound to a term stack variable) or a permanent environment stack variable for `X` in the example. If a temporary variable is chosen the OIM and QOIM will mark the same data in the example. Even if `X` is a permanent variable, $env\_size(\mathbf{E0}) = 2$, so the term referred to by `X` is not marked.

Let C be the number of garbage words, and let D be the total number of words in the black area. Let E be the number of non-garbage words outside the black area, created since LB. Since the marking phase traverses only the non-garbage data structures of the black area, the time required for it is bounded by a linear expression $m_1C + m_2E + m_3$. The compacting phase traverses the whole temporary area, whose size is also D. The time required for it is bounded by the expression $c_1C + c_2(D - C) + c_3$. Since the time for processing a non-garbage word (which is copied) is much larger than for processing a non-garbage word (which is ignored), we have that $c_1 \gg c_2$. On the other hand, for most programs there will be much more garbage than non-garbage words, so $D - C \gg C$. The updating phase again traverses the whole temporary area but also the parts outside the black area, referring to it. It is bound by an expression $u_1C + u_2(D - C) + u_3D + u_4E + u_5$ (the term $u_3 + D$ corresponds to the time for resetting the temporary area). According to our experiments, the times required for marking and updating are roughly equal and they are both much larger than the time for updating. By adding the expressions above we find that the total time required for garbage collection is bound by the expression

$$(m_1 + c_1 + u_1)C + (c_2 + u_2)(D - C) + u_3D + (m_2 + u_4)E + m_3 + c_3 + u_5$$

which equals

$$(m_1 + c_1 - c_2 + u_1 - u_2)C + (c_2 + u_2 + u_3)D + (m_2 + u_4)E + m_3 + c_3 + u_5.$$

We find that in our implementation, $m_1 \approx m_2 \approx c_1 \approx u_1 \approx u_4$ (let this approximate value be a) and $c_2 \approx u_2 \approx u_3$ (let this be b). Also, $a \gg b$. Under this assumption the expression can be simplified to

$$3aC + 3bD + 2aE + m_3 + c_3 + u_5.$$

Since usually $D - C \gg C$, it is important to make b as small as possible. The constant terms do not usually matter very much, except when D is very small.

This garbage collection algorithm is intended for machines with a large virtual address space. According to Moon, one should pay attention to the difficulties introduced by virtual memory itself [14]. The time spent

in accessing memory increases rapidly if the page fault ratio is high. In the marking algorithm the memory references will probably be out of control and generate a quite high page fault rate. However, both the compacting and updating phases go through memory almost linearly, referencing the cache and the virtual memory in a way they are well suited for.

The space overhead for this algorithm is the N field in environments and the R field in choice points. In our implementation, the N field does not make environments larger and does not require any particular initialization. The R field was already present. For an implementation not having such a field, the algorithm could be trivially changed to compute the number of saved temporary variables in a choice point the same way as its `fail` routine does.

The algorithm itself uses an extra piece of storage as large as the black area and a recursion stack as deep as the most non-tail recursive, non-garbage term (in our machine this stack is located on the ordinary environment stack). See below for further discussion about these areas.

10. THE TEMPORARY AREA

It is often desirable to collect garbage in all of the dark area, i.e., to make the black area equal to the grey area. The temporary area must be as large as the black area.

A simple way of allocating a temporary area for garbage collection is to initiate garbage collection before the term stack occupies more than half of its maximum size. This guarantees that there is always room for the temporary area. However, if execution has been allowed to proceed further than this, the dark area may be bigger than the unused term stack area. In this case we may choose as LB the oldest choice point such that the black area is smaller than the unused area. This will add a number of words to the unused area. Then a new, lower LB can probably be chosen for a second collection. This process can be repeated until LB can be set to GCB or the reclaimed space is too small to choose an older LB.

11. WHEN AND WHERE SHOULD GARBAGE BE COLLECTED?

It is often assumed that garbage collection should occur when the term stack is full or almost full (see e.g., Pittomvils [15 p. 193]). This can often ensure that few garbage collections are made. But this does not necessarily give the shortest total execution time, because much time may be spent collecting garbage in areas containing comparatively little garbage. According to Lieberman and Hewitt, a good strategy is to reclaim garbage among recently created terms more often than among older terms [12].

We have noted one particular occasion when one can identify a part of the term stack which contains very much garbage. At the end of each loop in an iterative (here: tail-recursive and deterministic) program, almost everything created since the beginning of the loop is garbage. Reclaiming garbage in this area reclaims much storage in a short time. The overhead for starting a garbage collection makes it unrealistic to do so for every loop and therefore we suggest a new control primitive *garbage cut* ('`!!`' in programs). Its operational semantics is exactly the same as cut, but it also reclaims garbage in the part of the term stack created since the predicate in which it occurs was entered. It is intended to be placed immediately before the tail-recursive call in an iterative predicate.

From our experiments we conclude that inserting a garbage cut in the main loop of an iterative program (e.g., a compiler) reduces the garbage collection time. It also reduces the number of page faults in virtual memory, because the term stack never grows large, making references more local. For further elaboration on this subject, see Barklund and Millroth [2].

12. VARIATIONS

Some variations of the algorithms may be worth investigating.

- If updated addresses in the term stack could be distinguished, the algorithm would perhaps become more efficient by updating backward references within the term stack during the compacting phase and only updating the forward references during the updating phase.
- If a backward pointer-chaining technique (in the temporary area), is used for the marking phase it is possible to dismiss the stack. This is similar to pointer-reversal techniques [1, 16].

13. RELATED WORK

The first version of this algorithm was developed in the autumn of 1985, and was inspired mostly by the work of Pittomvils, Bruynooghe, and Willems, presented at SLP '85 [15].

Appleby, Haridi and Sahlin have recently presented a garbage collection algorithm for WAM, with many similarities to ours [1]. However, they have adapted a pointer reversal technique for the marking phase, instead of using a stack [16]. Their algorithm does not reverse pointers in the choice point stack or save the current state in a choice point. Since it marks words by setting a dedicated bit to 1, it cannot handle strings in the term stack area, although it may be possible to modify the algorithm to achieve this.

ACKNOWLEDGMENTS

The author would like to thank his colleagues at UPMAIL for the stimulating environment, but particularly Håkan Millroth who implemented an earlier version of the algorithm, Mats Carlsson who has implemented large parts of Tricia, Yoshihiko Futamura and Åke Hansson who gave valuable comments on an earlier version of the paper, and Marianne Ahrne for proof-reading. He also owes much to his family.

REFERENCES

[1] K. Appleby, S. Haridi, D. Sahlin, *Garbage Collection for Prolog Based on WAM*, SICS Research Report R 86009, Stockholm, 1987.

[2] J. Barklund, H. Millroth, *Garbage Cut for Garbage Collection of Iterative Prolog Programs*, 1986 Symposium on Logic Programming, pp. 276–283, Salt Lake City, Sept. 1986.

[3] J. Barklund, H. Millroth, *Code Generation and Runtime System for Tricia*, UPMAIL Technical Report 36, Uppsala, 1987.

[4] D. L. Bowen, *DECsystem-10 Prolog User's Manual*, University of Edinburgh, Dept. of Artificial Intelligence, Edinburgh, 1981.

[5] M. Bruynooghe, *Garbage Collection in Prolog Interpreters*, in J. A. Campbell (ed.), *Implementations of Prolog*, Ellis Horwood, 1984.

[6] M. Carlsson, *Compilation for Tricia and Its Abstract Machine*, UPMAIL Technical Report 35, Uppsala, Sept. 1986.

[7] T. Chikayama, Y. Kimura, *Multiple Reference Management in Flat GHC*, 4th International Conference on Logic Programming, pp. 276–293, Melbourne, May 1987.

[8] W. F. Clocksin, *Design and Simulation of a Sequential Prolog Machine*, New Generation Computing, vol. 3, no. 2, pp. 101–120, 1985.

[9] J. Cohen, *Garbage Collection of Linked Data Structures*, Computing Surveys, vol. 13, no. 3, pp. 341–367, Sept. 1981.

[10] E. W. Dijkstra, L. Lamport, A. J. Martin, C. S. Scholten, E. F. M. Steffens, *On-the-Fly Garbage Collection: An Exercise in Cooperation*, Communications of the ACM, vol. 21, no. 11, pp. 966–975, Nov. 1978.

[11] D. E. Knuth, *The Art of Computer Programming: Volume 1 / Fundamental Algorithms*, Addison-Wesley, 1973.

[12] H. Lieberman, C. Hewitt, *A Real Time Garbage Collector Based on the Life Time of Objects*, Communications of the ACM, vol. 23, no. 6, pp. 419–429, June 1983.

[13] F. L. Morris, *A Time- and Space-Efficient Garbage Compaction Algorithm*, Communications of the ACM, vol. 21, no. 8, pp. 662–665, Aug. 1978.

[14] D. A. Moon, *Garbage Collection in a Large Lisp System*, 1984 ACM Symposium on Lisp and Functional Programming, pp. 235–246, Austin, Aug. 1984.

[15] E. Pittomvils, M. Bruynooghe, Y. D. Willems, *Towards a Real Time Garbage Collector for Prolog*, 1985 Symposium on Logic Programming, pp. 185–198, Boston, July 1985.

[16] H. Schorr, W. M. Waite, *An Efficient Machine-Independent Procedure for Garbage Collection in Various List Structures*, Communications of the ACM, vol. 10, no. 8, pp. 501–506, Aug. 1967.

[17] T. A. Standish, *Data Structure Techniques*, Addison-Wesley, 1980.

[18] G. L. Steele Jr., *Multiprocessing Compactifying Garbage Collection*, Communications of the ACM, vol. 18, no. 9, pp. 495–508, Sept. 1975.

[19] D. H. D. Warren, *An Abstract Prolog Prolog Instruction Set*, SRI Technical Note 309, Oct. 1983.