TCPW/CW/K. INT.

# BIM-PROLOG

Joint Project between
BIM
and
Department of Computer Science
Katholieke Universiteit LEUVEN

Sponsored by DPWB/SPPS
under grant nr KBAR/SOFT/1

A Real Time Garbage Collector
for Prolog

by
Edwin PITTOMVILS **
Maurice BRUYNOOGHE **

Internal Report
BIM-prolog IR5

November 1984

* BIM
Kwikstraat 4
B-3078 Everberg Belgium
tel. +32 2 759 59 25

** Katholieke Universiteit Leuven
Department of Computer Science
Celestijnenlaan 200A
B-3030 Heverlee Belgium
tel. +32 16 20 06 56

DPWB = Diensten van de eerste minister : Programmatie van het Wetenschapsbeleid.
SPPS = Services du premier ministre : Programmation de la Politique Scientifique.

# A real time garbage collector for PROLOG

**Edwin Pittomvils and Maurice Bruynooghe**

Department of Computer Science
K.U.Leuven
Celestijnenlaan 200A
B 3030 Heverlee
Belgium

## 1. Abstract

Inspired by the work of [Lieberman 83] which describes a garbage collector which takes into account the lifetimes of objects, we were able to improve the garbage collection algorithm as described in [Bruynooghe 82a]. The improved algorithm reduces the time needed to mark and compact the storage area by limiting its activity to relatively small segments of memory. Moreover it can easily be extended to a real-time garbage collector. Contrary to the real-time garbage collector described in [Bekkers et al 84], it preserves the important properties of increasing the locality of references and allowing for the recuperation of memory during backtracking.

## 2. Introduction

Very little investigation has been done in the field of garbage collection for large memories. With the constant evolution of ever larger memories for ever lower prices, one might think there does not exist a garbage collection problem anymore. However, even for relatively small applications in the domain of artificial intelligence, one runs out of memory very rapidly.

This paper describes a new garbage collection algorithm for a sequential PROLOG processor.
We will follow mainly the model proposed in [Tick & Warren 84], because it has some very interesting properties which makes it well suited to graft our garbage collector upon it and, not in the least, because it tends to be a very promising model on its own.

In [Bruynooghe 82a], the garbage collection process was improved by reducing the number of starting points for the marking algorithm. Only those variables that are really needed for further computation are marked. This results in a more complete recovery of useless memory. However, no attention is paid to the time dimension of the problem i.e. there are no indications on how to perform this algorithm quickly.

In [Tick & Warren 84], one obtains a better memory management on the local stack by performing a generalised tail recursion optimisation. To this end, the variables in the environments have to be arranged in a special way. This arrangement will help us to perform the marking algorithm in [Bruynooghe 82a] more quickly.

In [Bekkers et al 84] a further improvement was made by taking the reset information into account. This will be realised in our model by what we call "virtual backtracking" (also described in [Bruynooghe 84]).

They propose a real time garbage collector for PROLOG by making use of Dijkstra's on-the-fly garbage collector [Dijkstra 78]. For large memories, this algorithm seems to have many drawbacks.
Dijkstra's garbage collection algorithm consists of a two-processor model with one processor (the mutator) doing the proper computation and the other processor (the collector) performing garbage collection all the time. The collector's activity consists of a number of batches, each batch consisting of a marking phase followed by an appending phase. In the appending phase, non-marked cells are incorporated into a free list.
For large memories, marking all accessible structures takes a long time and the possibility exists that the free list is exhausted before the marking phase comes to an end. In other words, there may be long suspensions of the mutator, waiting for the collector to terminate.
A second drawback is, that memory cells are allocated randomly throughout memory. Hence, one cannot guarantee localness of pointers. In a virtual memory environment, this can considerably degrade the mutator's performance.
A third important drawback is, that space recovery on backtracking is impossible. Because the heap does not operate as a stack, there is no memory recovery possible by simply popping the stack. Therefore, the cells which become inaccessible on backtracking have to be recovered by the general marking and collecting mechanism.

In [Lieberman 83] the garbage collection problem is stated in a more general setting. The heap is divided into a number of regions. The object of this division is to vary the degree of garbage collection for each region. The following strategy is used:

- Recent (created) regions are supposed to contain a high proportion of garbage. These regions are collected frequently.

- Older regions are supposed to contain a low proportion of garbage. They are collected less frequently.

This strategy results in a higher efficiency of the garbage collector. The cost per collected cell is less than in a strategy, where the age of the regions is not taken into account.

Our garbage collection algorithm is an adaptation of ideas in [Lieberman 83] to the particularities of a PROLOG implementation. The heap is divided logically into a number of segments, each segment corresponding to a backtrackpoint (choicepoint). We are able to compact one segment at a time by what we call "incremental marking". We do not have to mark all accessible cells in order to "know" all accessible cells! This can make the marking phase very short.

In the following sections, we consider three marking strategies:

i. optimal total marking

ii. optimal incremental marking

iii. quasi(non)-optimal incremental marking

Optimal marking means that a theoretically minimum number of cells is marked. In the first two strategies, we perform a full "virtual backtracking" allowing optimal marking. The third strategy performs only a partial virtual backtracking and therefore is non-optimal with regard to the recovered memory.

In our terminology, the garbage collector developed in [Bekkers et al 84] is a real time garbage collector with optimal total marking. For large memories, we expect that an optimal incremental- or a quasi-optimal marking will give considerably better timing results.

We obtain localness of pointers by dividing the heap into a number of segments. The number of inter-segment pointers is relatively small. Most pointers are internal segment pointers (i.e. pointers pointing into the segment itself) which link datastructures together. Moreover by compacting the segments we obtain increased locality of pointers.

Note that each segment on itself operates as a stack.

## 3. Execution of PROLOG programs

The concrete run-time structures of the PROLOG processor (interpreter) consist of a number of stacks and registers. They are the concrete representations of the abstract run-time structures: namely a search tree (OR-tree) and a proof tree (AND-tree). The search tree constitutes the solutionspace (the initial goal and the goals deducible from it through unification). The proof tree describes a path in the search tree. Goals are deduced one from another by a "depth first, left to right" computation rule. More details about the working of PROLOG interpreters can be found in [Bruynooghe 82b] and [van Emden 84].

## 4. Data areas and registers

The data areas are mostly identical to those proposed in [Tick & Warren 84]. Only the heap and the trail will be organised in a different way. The description of some datastructures will be simplified in order to enhance the understanding of the developed algorithms.

i. local stack (environment stack):
The local stack is the concrete representation of the proof tree. The local stack contains environments and choice points.

environment (env) :

call _of_ env : continuation program pointer (= next call to be executed in this environment)
father _of_ env : pointer to the fatherenvironment
visit _of_ env : boolean field (only needed in an optimal total marking strategy)
number _of_ env : indicates how many variables (Y1, Y2, ... ,Ynumber) have already been marked in env (only needed in an optimal total marking strategy)
Y1,Y2, ... ,Yn _of_ env : permanent variables

choice point (chpt) :

reset _of_ chpt : saved register TR′
heap _of_ chpt : saved register H
back _of_ chpt : saved register B ; pointer to the previous choice point
alternative _of_ chpt : pointer to the alternative clauses to match the call
cont _of_ chpt : saved register CP
env _of_ chpt : saved register E
A1,A2, ... ,Am _of_ chpt : saved argument registers

ii. the heap:
The heap consists of a number of segments. Each segment corresponds to a choice point. This is a one-to-one relation. Placing a choice point on the local stack closes a segment on the heap and opens a new one.

iii. the trail
On the trail, we place information about the bindings to be undone on backtracking. The logical division of the trail into segments is analogous to that of the heap.

iv. registers
We only mention those registers needed further in the text.

P : program pointer (to the code area)
CP : continuation pointer (next call to be executed)
E : topenvironment on the local stack
B : last choice point on the local stack
H : top of the most recent heapsegment (= opened heapsegment which is the current creationzone)
TR : top of the most recent trailsegment
HL : pointer to a choice point ; this is a history pointer needed by the garbage collector for incremental marking
A1,A2, ... , Am : argument registers

5. <u>Optimal total marking algorithm (first strategy)</u>

This section is more of a lead-up to the following sections. We explain briefly how the proof tree can be traversed efficiently and what virtual backtracking is all about. For some details, we refer to [Bruynooghe 84] and [Bekkers et al 84].

The algorithm can best be explained on the basis of an example. Consider the following proof tree and the corresponding data areas (fig1):

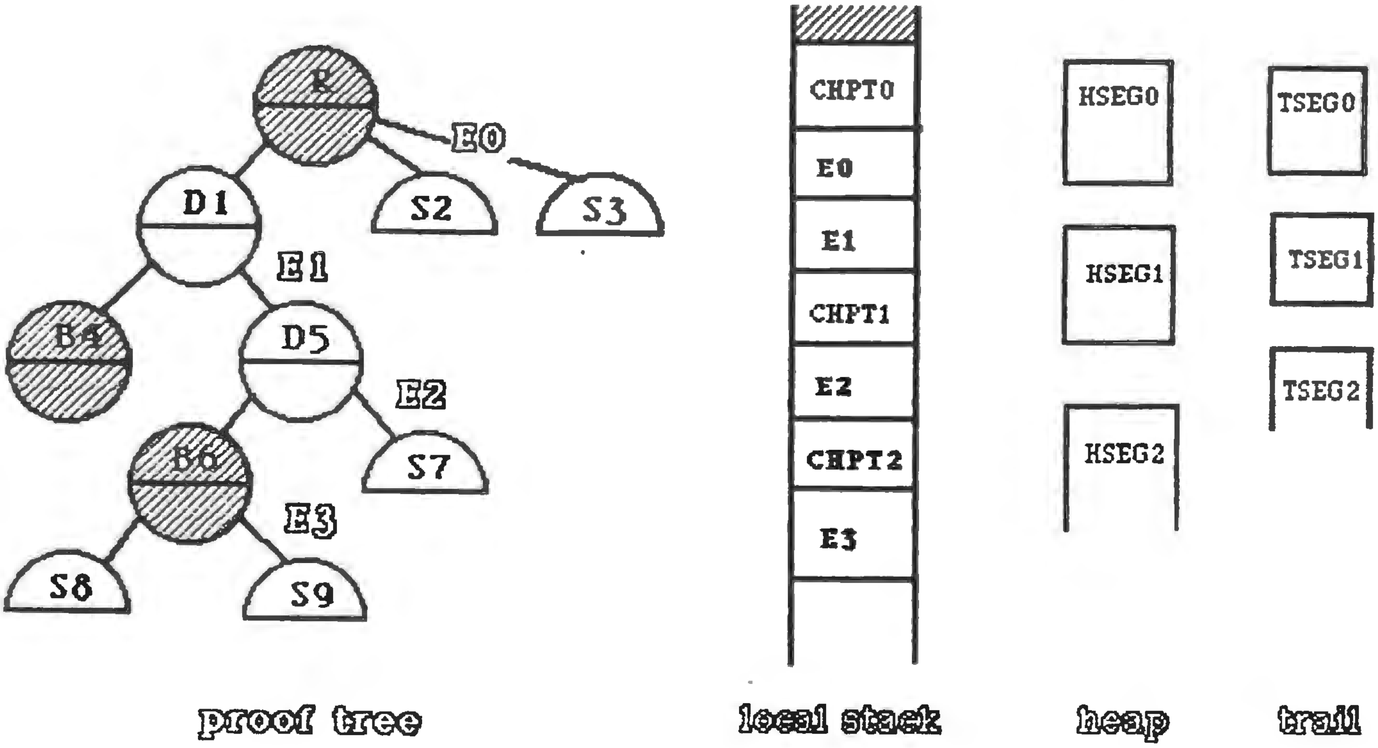


figure 1

In fig_1 the abbreviations have the following meaning:

- Si : call still to be executed
- Di : deterministic call
- Bi : non-deterministic call ( choice point )
- R : root (this is a sentinel)
- CHPTi : choice point on the local stack
- Ei : environment on the local stack
- HSEGi : heapsegment i
- TSEGi : trailsegment i

Placing CHPTi+1 on the local stack closes the segments HSEGi and TSEGi, and opens the segments HSEGi+1 and TSEGi+1. The marking algorithm starts from all living goal statements [Bruynooghe 82a]. In general, there is the current living goal and a living goal corresponding to each backtrackpoint. Here, we have three living goal statements (enumerated from the most recent to the oldest):

- current living goal: S8,S9,S7,S2,S3.
- goal corresponding to B6: S6,S7,S2,S3.
- goal corresponding to B4: S4,S5,S2,S3.

The marking proceeds from the most recent to the oldest living goal in order to perform virtual backtracking. First the variables occurring in the calls S8,S9,S7,S2,S3 are marked. Then we do virtual backtracking by resetting (virtually) the variables encountered in TSEG2. Indeed, for the marking of the next (older) living goal, these bindings do not logically exist.
During the treatment of the following living goal (corresponding to B6), we only need to mark the variables occurring in S6. Those occuring in S7,S2 and S3 have already been marked. Then we do virtual backtracking again by resetting the variables encountered in TSEG1.
At last the variables occurring in S4 and S5 are marked. Notice that TSEGO is always empty.

The garbage collector is called in case of heap overflow. The test on overflow takes place just before the execution of one of the following instructions: call proc/ar,n , execute proc/ar or proceed (see [Tick & Warren 84] for more details on those instructions).
In the following procedures, comments are placed between brackets.

```
procedure marking
begin
    { marking of the argumentregisters A1,A2, ... ,Aar
      in the case of a call proc/ar,n or execute/ar instruction }
    markargregisters( P );

    { the first living goal is determined by CP and the
      environment corresponding to CP }
    active_call := CP ;
    active_env := E ;

    marklivinggoal( active_call, active_env );

    next := B ;
    while next <> root do
         { determine the next living goal }
         active_call := cont of next ;
         active_env := env of next ;

         { resetting of the variables in the trailsegment
           pointed to by reset of next }
         virtualbacktrack( reset of next );

         { marking of the argumentregisters stored in the
           choice point next }
         markchptargregisters( next );

         marklivinggoal( active_call, active_env );

         next:= back of next ;
    od
end marking
```

```
procedure marklivinggoal( active_call, active_env )
begin
     end_of_goal := false ;
     repeat
          if visit of active_env
          then begin
               markvariables( number of active_env,
                  numbervar(active_call), active_env );
               number of active_env := numbervar(active_call);
               end_of_goal := true
          end
          else begin
               markvariables( 0 , numbervar(active_call),
                  active_env );
               number of active_env := numbervar(active_call) ;
               visit of active_env := true ;
               if father of active_env = nil
               then end_of_goal := true ;
               else begin
                    active_call := call of active_env ;
                    active_env := father of active_env ;
               end
               fi
          end
          fi
     until end_of_goal
end marklivinggoal


procedure markvariables( nbl, nbh, active_env )
begin
     { marking of the variables Ynbh, Ynbh-1, ... , Ynbl+1
       in active_env }
     ...
end markvariables


function numbervar( active_call ) returns (integer)
begin
     { This function returns the number of variables needed
       in active_env to execute active_call and its righthand
       brothers. This information can be found in the code
       by going back one instruction from the place pointed
       at by active_call. There you find the instruction
       call proc/ar,n. In the case of an execute instruction
       n=0. }
     ...
     return( n );
end numbervar
```

## 6. Incremental marking

Placing a choice point (backtrackpoint) on the local stack saves (freezes) all existing structures on the heap. More technically: placing CHPTi+1 closes HSEGi and opens HSEGi+1 which becomes the new creationzone on the heap. All the segments HSEG0,HSEG1, ... , HSEGi are saved.

All structures that are "no garbage" (i.e. structures needed for further computation) at the moment of saving will remain "no garbage" until we undo this saving by backtracking. Indeed, after backtracking to a backtrackpoint, we must reinstate exactly the same state as just before placing that backtrackpoint. In other words, if we have compacted a saved segment on the heap once, this segment cannot contain inaccessible cells until we open it again (undo the saving).

We can now see more clearly where the first strategy fails. The second, third, ... time we call the garbage collector, a total marking algorithm will mark and compact many segments that have no garbage at all. In fact, it suffices to compact only those segments which have not been compacted before or which have been opened on backtracking.
It should now also be clear that we must only mark a restricted number of active goals.

## 7. Optimal incremental marking (second strategy)

### 7.1. The marking strategy

As stated above, "optimal" refers to a theoretical minimum of marked cells. The second strategy will be optimal and give much better timing results than the first strategy (also being optimal).

We give a real-time model, making use of two processors being a working processor (worker) and a garbage collector processor (collector). The model is easily expandable to a multiple-processor model with one working processor and one or more garbage-collector processors (see also [Lieberman 83]).
Incremental marking will also give much better timing results for the classical one- processor model (= the worker doing both the computation and the garbage collection).

Only one segment on the heap will be marked and compacted at a time. This can be done by marking the goal corresponding to CHPTi+1 , where CHPTi is designated by HL. HL is a history pointer and points to the oldest choice point for which the heapsegment designated by heap of HL has not been compacted in previous garbage collections. HL must be adjusted on backtracking or in the case of a cut operation as follows:

```
if HL is more recent than B
then HL := B
fi
```

( initially we have HL = B = CHPT0 )

Also after compaction of the segment CHPTi , HL must be adjusted and point to CHPTi+1 (= following choice point).

Consider the following example (fig_2):

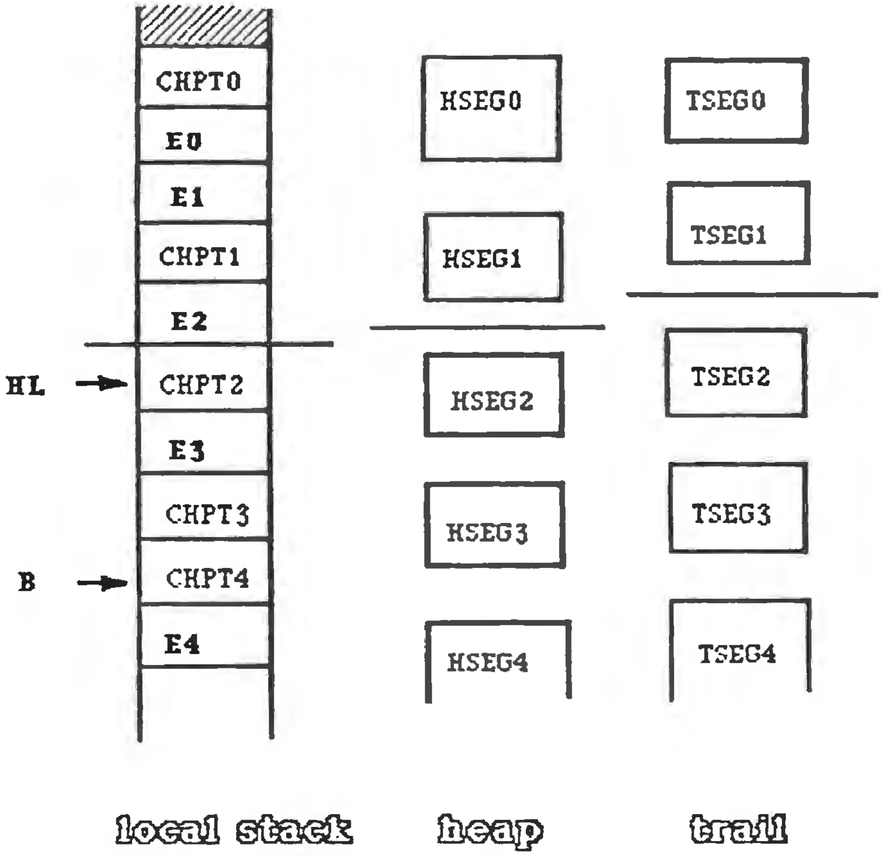


figure 2

The segments HSEG0 and HSEG1 have been compacted in previous garbage collections (see HL in fig_2). HSEG2 is the oldest segment that has not yet been compacted. The marking of HSEG2 proceeds as follows.

First determine the goal to be marked by calculating active_call and active_env. We have:

```
     active_call = cont of CHPT3
     and  active_env = env of CHPT3
```

Then we must perform virtual backtracking by resetting virtually the variables noticed in TSEG3 and TSEG4. Indeed, the heapsegments HSEG3 and HSEG4 do not logically exist when marking the goal corresponding to CHPT3. The marking starts from the argumentregisters stored in CHPT3 (markchptargregisters( CHPT3 )). At last we must mark the rest of the goal (marklivinggoal( active_call, active_env) ).

We now have the following modified marking algorithm:

```
procedure marking1
begin
     ptchpt := B ;
     while ptchpt is more recent than HL do
          virtualbacktrack( reset of ptchpt );
          hptchpt := ptchpt ;
          ptchpt := back of ptchpt ;
     od

     { marking the argumentregisters stored in the
       choice point hptchpt }
     markchptargregisters( hptchpt );

     active_call := cont of hptchpt ;
     active_env := env of hptchpt ;
     marklivinggoal1( active_call, active_env)
end marking1


procedure marklivinggoal1( active_call, active_env )
begin
     { Because we only have to mark one living goal,
       the use of visit of active_env and number of active_env
       is not necessary anymore. No double marking is possible. }
     end_of_goal := false ;
     repeat
          markvariables( 0, numbervar(active_call) , active_env );
          if father of active_env = nil
          then end_of_goal := true
          else begin
               active_call := call of active_env ;
               active_env := father of active_env
          end
          fi
     until end_of_goal
end marklivinggoal1
```

Before we start executing marking1, every cell has to be "unmarked" in the segment under compaction and in the older segments. There are some alternatives to solve this problem. We could use a bitmaptable to mark the cells and after the marking reset all the marked cells, by cleaning up the whole bitmaptable.

Another alternative is to keep a table of marked entries. This table is released when the marking is finished.

In fact, for real time working, the marking of cells in the segment under compaction (HSEGi) is replaced by copying. Instead of marking the cells, they are directly evacuated from HSEGi to a new segment (HSEGi') which will become the compacted segment (fig_3).

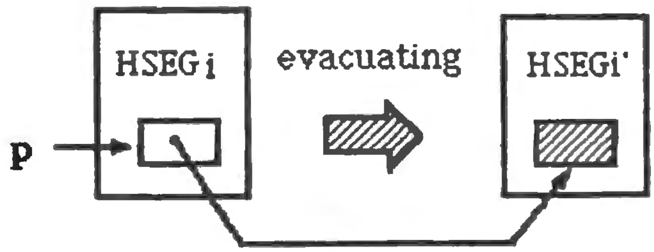


figure 3

References to HSEGi will be handled by placing in HSEGi pointers to the evacuated object in HSEGi'. For details see [Lieberman 83]. After marking, the place occupied in memory for HSEGi can be released.

## 7.2. Updating of pointers

We will have to update all pointers to HSEGi. They can be divided into forward pointers (pointers from older segments, environments or choice points) to HSEGi, backward pointers (pointers from younger segments, environments or choice points) to HSEGi and internal pointers.

Internal pointers are handled automatically through the copying mechanism.

All the forward pointers to HSEGi can be found in TSEGi on the trail. These are the only entries to HSEGi from older segments.

For the updating of backward pointers, we associate an update table (UDTBi) with each non-compacted heapsegment (HSEGi). In UDTBi we note all the backward pointers to HSEGi. These update tables grow dynamically during the calculations of the worker, even if the corresponding segments are closed. This is illustrated in fig_4.

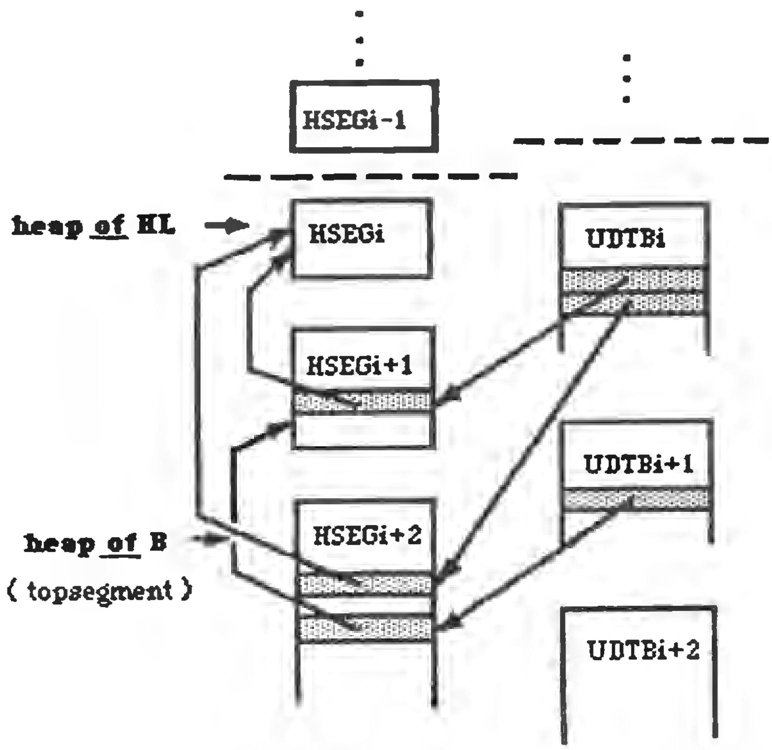


figure 4

The use of updatetables will be considerably faster than the general "scavenging technique" (= scanning more recent segments, environments looking for backward pointers to be updated) proposed in [Lieberman 83]. We can afford the use of updatetables because we only have to keep a restricted number of them. Indeed, only the non-compacted segments need updatetables, because we never have to compact segments more than once.

### 7.3. Coordination and synchronisation of the processors

The garbage collector must be synchronised in such a way that there are only a few segments (e.g 5) on the top of the heap which are not compacted. It is a good policy to always keep a few uncompacted segments on the top of the heap to ensure little mutual interference between the worker and the collector. We can motivate this as follows. The worker usually operates on structures residing in the topsegment. The probability that the worker asks access to the segment under compaction is rather small.
If there are not enough uncompacted segments, we simply suspend the collector.

The worker and the collector operate on the same datastructures. Mostly, the worker will interrupt the collector when access to the same data is required. This interruption will cause a suspension of the collector.
On interruption, the collector finishes the atomic action it is working on, and then it transfers control to the worker. More on atomic actions can be found in [Dijkstra 78].

Roughly, we have suspension of the collector on the following occasions:

- Access of the worker to the segment under compaction.

- On backtracking or when executing a cut instruction.
  It is possible that the segment under compaction has to be removed on backtracking. This gives no major problems. The collector stays suspended until enough uncompacted segments exist again.
  When the worker signals to the collector that it can continue working, the collector must decide whether to stay suspended or to continue collecting. If HL is no longer older than B after backtracking, the collector stays suspended and resumes only when a certain number of uncompacted segments are in existence again. In the other case, the collector resumes immediately.

- The updating of the pointers to the compacted segment causes some critical periods, when there must be mutual exclusion between the worker and the collector.

The worker must be suspended as little as possible. There is suspension on finishing atomic actions ( evacuation of datastructures, redirection of pointers, ... ) by the collector after an interrupt from the worker.

## 8. Quasi-optimal marking (third strategy)

In the second strategy we must only mark one living goal at a time. This is a considerable improvement over the first strategy. However the marking of the one living goal can cause the marking of many structures in older compacted segments. This marking in older segments is necessary because there are forward pointers in the older segments to the segment under compaction (HSEGi).
When we mark the goal corresponding to HSEGi, all the necessary information about the (logically existing) forward pointers to HSEGi can be found in TSEGi. Not all of them will be marked in the second (optimal) strategy. Nevertheless the number of forward pointers which will not be marked seems to be very small.

The third strategy differs from the second one in that we will mark the one living goal only partially. This means that during the marking, pointers to older segments are not followed. Instead we will follow all forward pointers to HSEGi. Partial marking causes the marking of too many structures but we expect this disadvantage to be negligible compared to the time savings it allows. The difference in marking will be clarified by an example. Consider the following PROLOG clauses:

```
p(a,b).
p(c,d).

q( a.R , a.b.S ) :-
        garb_coll , r(S) , write(R) , write(S).

r(e).
```

and the query:

```
?- p(Y,Z) , q(X,X) , write(Y) , write(Z).
```

A fictive call garb_coll is inserted to clarify the example. Just before executing the call garb_coll we have the situation represented in fig_5 and fig_6.

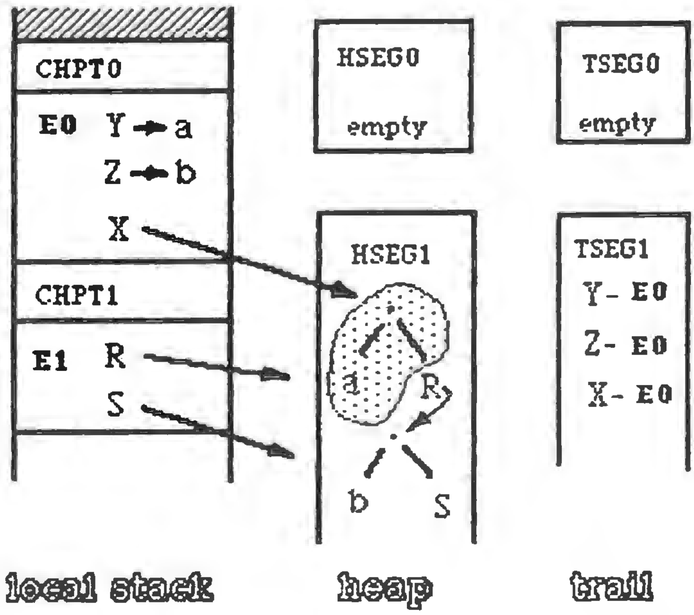


figure 5

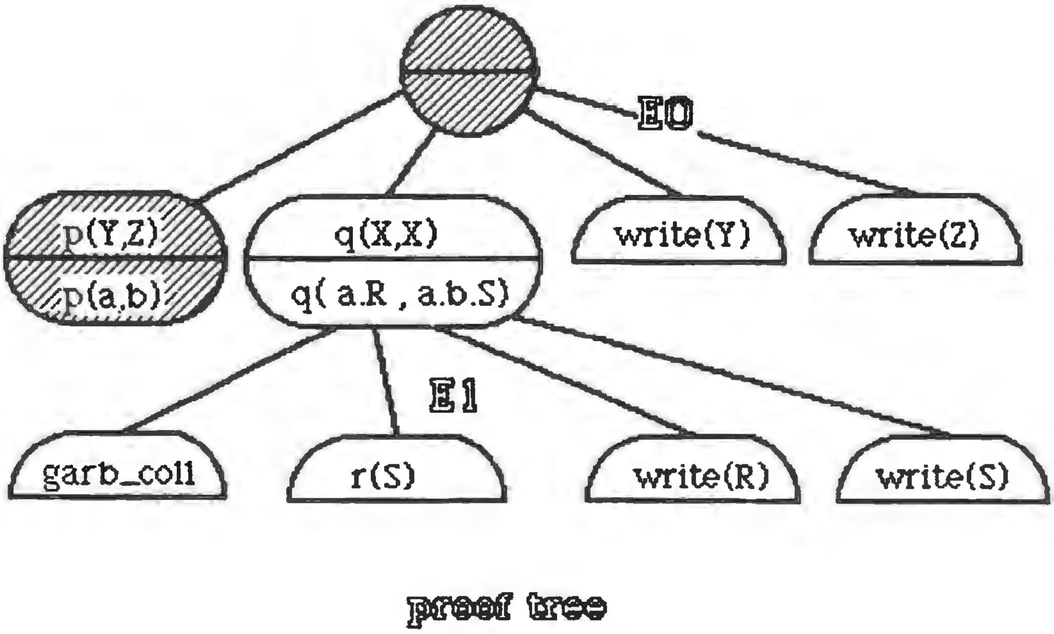


figure 6

In the optimal strategy, the variable X of E0 will not be marked wheras it is marked in the quasi optimal strategy. The shaded area in fig_5 indicates memory locations which are unnecessarily marked.
In the programs we have tested ( a.o. a concurrent PROLOG interpreter in PROLOG [Shapiro 83]), we have not found significant differences between strategy 2 and strategy 3 with respect to recovered memory. Only by testing a large number of extensive examples can one give a definite answer to the question.

The only change to algorithms from the previous section, is the procedure marklivinggoal1. The updated version follows:

```
procedure marklivinggoal2( active_call, active_frame) ;
begin
     end_of_goal := false
     repeat
          { We make use of a procedure markvariables1. The
            difference with markvariables is, that pointers
            pointing to memory locations which are less recent
            than HL (for the local stack) or heap of HL
            (for the heap) will be neglected. }
          markvariables1( 0, numbervar(active_call) , active_env );

          { Marking of those cells in HSEGi which are accessible
            from older segments and environments (older than HSEGi).
            Therefore we must mark all the pointers noticed in
            TSEGi. }
          markoldtonew( reset of HL );

          if active_env is not more recent than HL
          then end_of_goal := true
          else begin
               active_call := call of active_env ;
               active_env := father of active_env
          end
          fi
     until end_of_goal
end marklivinggoal2
```

## 9. Conclusions

We have showed how garbage collection for PROLOG can be significantly improved. These improvements were mainly possible by taking advantage of specific properties of the language.
The indeterminism of PROLOG leads to a number of saved states. The basic idea is that the datastructures corresponding to a saved state have to be compacted only once. By incremental marking, we avoid double marking and compaction of already compacted structures, corresponding to saved states. Moreover, by using an appropriate (segmented) memory organisation, this incremental marking leads to a real-time garbage collection algorithm.
The use of a segmented heap where each segment operates as a stack guarantees localness of pointers.

In a first phase we have implemented a garbage collector for a sequential PROLOG interpreter making use of the total optimal marking strategy. In a second phase, this garbage collector was modified to the quasi-optimal incremental marking strategy [Pittomvils 84].
A real time garbage collector making use of incremental marking is under development.

## 10. Acknowledgments

M. Bruynooghe is supported as a research associate by the Belgian National Fund for Scientific Research.

E. Pittomvils is currently working on a project in collaboration with the Belgian Institute of Management and is supported by the Belgian Institute of Scientific Research in Agriculture and Industry (IWONL-IRSIA).

## 11. References

[Bekkers et al 84]
Y.Bekkers, B.Canet, O.Ridoux, L.Ungaro, A memory management machine for PROLOG interpreters, Proceedings Second International Logic Programming Conference, 1984, pp. 343-353.

[Bruynooghe 82a]
M.Bruynooghe, A note on garbage collection in PROLOG interpreters, Proceedings First International Logic Programming Conference, 1982, pp. 22-55.

[Bruynooghe 82b]
M.Bruynooghe, The memory management of PROLOG implementations, Logic Programming, Eds. K.L.Clark & S.A.Tarlund, Academic Press, 1982, pp.83-98.

[Bruynooghe 84]
Garbage collection in PROLOG interpreters, Implementations of PROLOG ed. J.A.Campbell, Ellis Horwood, 1984, pp. 259-267.

[Dijkstra 78]
Edsger W.Dijkstra, On-the-fly garbage collection: An exercise in cooporation, Communications of the ACM. Nov.1978, Vol.21, No.11, pp.966-975.

[Lieberman 83]
Henry Lieberman & Carl Hewitt, A real time garbage collector based on the life time of objects, Communications of the ACM. Jun.1983, Vol.26, No.6, pp.419-429.

[Pittomvils 84]
A garbage collector for PROLOG (in Dutch), Katholieke Universiteit Leuven, Project completed to obtain the degree of engineer in computer science.

[Shapiro 83]
E.Y.Shapiro, A subset of CONCURRENT PROLOG and its interpreter, Revised ICOT TC TR-003, JAPAN, 1983.

[Tick & Warren 84]
E.Tick & D.H.D.Warren, Towards a pipelined PROLOG processor, 1984 International Symposium on Logic Programming, IEEE Computer Society Press, 1984, pp.29-40.

[van Emden 84]
An interpreting algorithm for PROLOG programs, Implementations of PROLOG ed. J.A.Campbell, Ellis Horwood, 1984, pp. 93-110.