A. Voronkov (Ed.)

Logic Programming

First Russian Conference on Logic Programming
Irkutsk, Russia, September 14-18, 1990
Second Russian Conference on Logic Programming
St. Petersburg, Russia, September 11-16, 1991
Proceedings

Springer-Verlag
Berlin Heidelberg New York
London Paris Tokyo
Hong Kong Barcelona
Budapest

Series Editor

Jörg Siekmann
University of Saarland
German Research Center for Artificial Intelligence (DFKI)
Stuhlsatzenhausweg 3, W-6600 Saarbrücken 11, FRG

Volume Editor

Andrei Voronkov
European Computer-Industry Research Centre (ECRC)
Arabellastraße 17, W-8000 München 81, FRG
and
International Laboratory of Intelligent Systems (SINTEL)
Universitetski Prospect 4, 630090 Novosibirsk 90, Russia

CR Subject Classification (1991): F.4.1, I.2.3

ISBN 3-540-55460-2 Springer-Verlag Berlin Heidelberg New York
ISBN 0-387-55460-2 Springer-Verlag New York Berlin Heidelberg

Printed in Germany

Typesetting: Camera ready by author
Printing and binding: Druckhaus Beltz, Hemsbach/Bergstr.
45/3140-543210 - Printed on acid-free paper

Preface

The Russian Conferences on Logic Programming were organised with the aim of bringing together researchers from the Russian and the international logic programming communities. The first conference was planned to be held on the shore of Lake Baikal. However due to some local problems it was held September 14–18, 1990, in Irkutsk — a pleasant city in the Eastern part of Siberia. The number of participants was 71 from the Soviet Union and 11 from the other countries. The second conference was held September 11–16, 1991, on the board the ship "Michail Lomonosov", named after the founder of the Russian Academy of Sciences. The ship started from St.Petersburg and sailed along the River Neva, Lake Ladoga and Lake Onega. This time there were 125 participants from the former Soviet Union and 32 from other countries.

This volume contains the selected papers presented to these two Russian Conferences on Logic Programming.

The idea to organise the conference on the ship proved very successful. The next conference will be held on the same ship July 15–20, 1992 in the famous period of white nights. For several reasons it has been decided to change the name of the conference to LPAR — Logic Programming and Automated Reasoning.

I wish to thank all the people who did a lot to organise these conferences in the time when Russia was in total disorder. Special thanks are due to Victor Durasov and Nelya Dulatova, who made it possible to rearrange the first conference in one day. Further thanks are due to Tania Rybina, Yuri Shcheglyuk, Yulya Mantsivoda, Lena Deriglazova, Maxim Bushuev, Vladimir Bechbudov, Lena Shemyakina and Andrei Mantsivoda for the first conference. For the second conference special thanks are due to Eugene Dantsin, Robert Freidson, Per Bilse and Valeri Shatrov. Further thanks are due to Robert Kowalski, Cheryl Anderson, Herve Gallaire, Tania Rybina, Oleg Gussikhin, Nikolai Ilinski, Arkadi Tompakov, George Selvais, Yuri Shestov, Michael Simuni, Vladislav Valkovski, Oleg Alekseev and Edward Yanchevsky.

Munich, March 1992 Andrei Voronkov

Conference Sponsors

RCLP'90

International Laboratory of Intelligent Systems (SINTEL)

RCLP'91

Association for Logic Programming

SRI Opyt

Per Gregers Bilse

Prolog Development Center A/S

Applied Logic Systems Inc.

Logic Programming Associates Ltd.

SICS Sweden

St. Peterburg Institute of Electrical Engineering

Conference Organizers:

RCLP'90:

International Laboratory of Intelligent Systems (SINTEL)

Irkutsk State University

RCLP'91

Russian Association for Logic Programming

Association for Logic Programming

Eurobalt Inc.

International Laboratory of Intelligent Systems (SINTEL)

St. Peterburg Institute of Electrical Engineering

IRI Inc.

Contents

Real-time memory management for Prolog

Yves Bekkers, Lucien Ungaro

INRIA / IRISA
Av. du Général Leclerc
35042 Rennes-Cedex
France

Tel : (33) 99 84 71 00
E-mail : bekkers@irisa.fr, ungaro@irisa.fr

Abstract

This paper relates a long experiment on implementing real time garbage collectors for Prolog. First, the main peculiarities of Prolog memory management are briefly reviewed. The relation between non-determinism and garbage collection are explained. Early-reset and variable shunting are presented. Attributed variables, a new type of data for implementing Prolog extensions and realizing a value trail mechanism is introduced. Then, a realtime garbage collection algorithm, taking these aspects into account, is entirely presented. The synchronizing problems, including those linked to non-determinism, are discussed in details. Finally, two concrete implementations are described.

Key words : Prolog, garbage collector, realtime, implementation, abstract machine, early reset, variable shunting, attributed variable, virtual backtracking

1 Peculiarity of Prolog memory management

The efficiency of Prolog systems is due to the trailing mechanism which allows the representation of choice points without copying them. For implementing a complete garbage collection, it is necessary to find which objects belong to a choice point representation. Such a GC must interpret the trailing information [Bekkers84a], [Bekkers84b].

1.1 Early reset of variable and variable shunting

The main idea is to watch variables and interpret their correct binding through the different choice points. The following cases can be distinguished :

Keeping all information - If a variable is accessible in its bound and unbound state, the variable and its binding must be kept.

Early reset of variables - If a variable is only accessible in its unbound state then its binding is useless. In this case the variable should be unbound in order to loose access to his binding [Appleby88], [Schimpf90], [Bruynooghe84], [Pittomvils85], [Barklund87a].

Variable shunting - If a variable is only accessible in its bound state then only its binding is useful and the variable itself is useless. In this case one should replace any occurrence of the variable by its binding value, [Huitouze90].

In §4.2.4.1, a detailled implementation of these mechanisms is given.

1.2 Attributed variables

For implementing extensions to Prolog, we have designed a new type of variable called attributed variable [Huitouze88], [Huitouze90], [Brisset91]. It is like a variable with an extra term attached to it, its attribute. The main property of this object is that the attribute is only accessible when the variable is unbound. Attributed variables take all their flavor with variable shunting. The space occupied by useless attributes is automatically reclaimed by variable shunting.

Attributed variables are used in implementing PROLOGII `freeze` and `dif` primitives; it can also be used to implement other kind of constraints or *value trail* mechanisms such as described in [Carlsson87], [Turk86], [Barklund87b], [Toura88], [Neumerkel90].

1.3 The abstract machine MALI

MALI is an abstract machine which has been designed to encapsulate the memory management of Prolog systems [Bekkers86], [Bekkers88]. It offers a set of commands for creating, accessing, modifying Prolog objects such as constructed terms, logical variables, ... the backtrack stack itself is managed by MALI. In top of that, MALI offers an automatic memory management.

Many different implementations of MALI have been experimented, some with a serial GC, some with a realtime GC, some entirely in software (in C), some microprogrammed on a specialized hardware to be inserted into PCs. The same Prolog system, written in C, compatible with PrologII [Colmerauer82], uses any of these implementations.

2 The state of a Prolog system

As usual, the state a Prolog system is summarized into three informations, the current goal statement, the backtrack stack and the trail. With MALI, the dynamic space is organized as a single heap managed by the garbage collector. This must be opposed to the WAM architecture where the dynamic space is split into several spaces, stacks and heap, each subject to its own specific management.

2.1 A tagged pointer scheme

Our runtime system uses a tagged pointer scheme with an elementary type of value called a `WORD`. A `WORD` contains an information field and a tag field. The tag specifies the type of the represented value which may be atom, list, tuple, variable, trail, level, etc ....

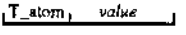
```
typedef struct {
    TAG tag ;
    INFO info ;
} WORD ;
```

From the point of view of the Garbage collector there are two types of tags, those indicating a pointer, and the others indicating small atomic values. In the two implementations of MALI that we are presenting here, the size of a referenced object, including tuples, is given by tags (see description of tuples later).

2.2 Representing Prolog terms

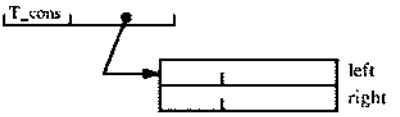
Each Prolog term, such as atom, cons, tuple, variable, is designated by a `WORD`.

atom : the tag is `T_atom`, the information is a bit-coding of the value of the atomic constant.



cons : the tag is `T_cons`, the information is a pointer to a `CONS` structure.

```
typedef struct {
    WORD left ;
    WORD right ;
} CONS ;
```



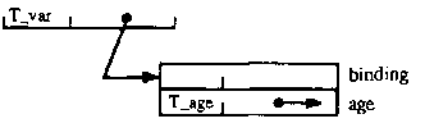
tuple : the tag is `T_tuple(i)`, the information is a pointer to an array of i words, one `WORD` per component.

variable : the tag is `T_var`, the information is a pointer to a `VAR` structure. The `binding` field holds the variable binding and a special tag `T_free` means that the variable is *unbound*. The `age` field is a

reference to the choice point which was at the top of backtrack stack when the variable was created. The tag of an age is `T_age`.

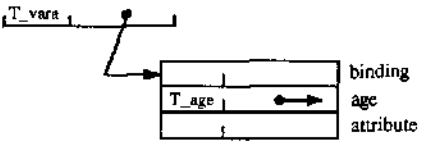
The `age` field is used by the garbage collector for implementing the variable shunting mechanism.

```
typedef struct {
    WORD binding ;
    WORD age ;
} VAR ;
```



attributed variable : the tag is `T_vara`, the information is a pointer to a VARA structure. This structure contains a supplementary field, the `attribute`, which is a term.

```
typedef struct {
    WORD binding ;
    WORD age ;
    WORD attribute ;
} VARA ;
```



2.3 Representing the active goal statements

The active goal statements are those in the choice point stack plus the current one.

Current goal statement : it is a list of goals held in a register G. Each goal is a term as previously described.

Backtrack stack : it is represented as a linked list of choice points. Each choice point is a structure called a LEVEL containing four fields :

```
typedef struct {
    WORD goal ;      /* the goal statement (a list of goals) */
    WORD clause ;    /* the clause (pointer to program, not relevant to the GC) */
    WORD trail ;     /* the trail (a list of bindings) */
    WORD next ;      /* the link to the next choice point */
} LEVEL ;
```

The top of that stack, is held in a register S.

The trail : each choice point contains a list of references to bound variables. It records such bindings that have to be undone on backtracking to recover the next choice point. In MALI it is a linked list of trail elements :

```
typedef struct {
    WORD var ;
    WORD next ;
} TRAIL ;
```

The current list of bindings is held in a register T.

Notice that the first element of the current trail is an *empty* element, the grey element in figure 1. It can be seen as a trail element created in advance. This element makes algorithm for updating the trail, figure 8, more uniform by putting the content of register T into memory location. [Barklund87a] and [Schimpf90] have met the same problem, their solution was to create a dummy choice point.

Registers G, S and T contain WORDs which give access to the useful data, these are the roots for the marking algorithm.

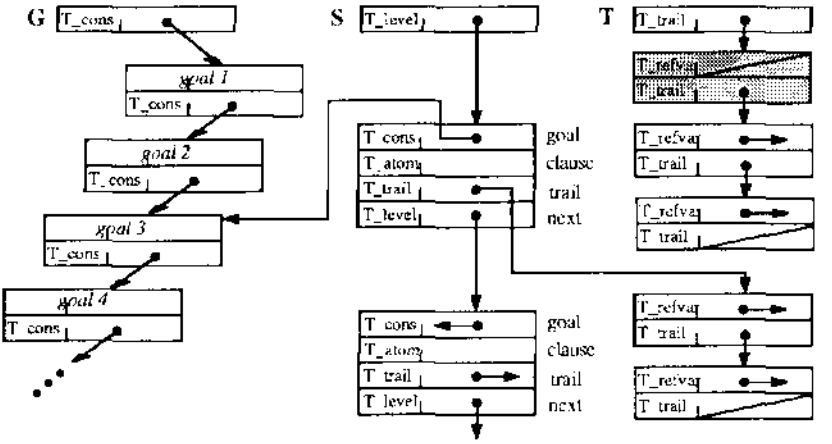


figure 1 : Goal statements, choice points and trail

3 Choosing the garbage collector

We took two constraints for the design of our garbage collector. First, it must work in real-time. Second, it must be well suited for Prolog, that is it must implement variable shunting and early reset of variables.

3.1 An algorithm for real-time garbage collectors

We have experimented two implementations of real time GC which are discussed later in §5. For both implementations, we have chosen Cheney's [Cheney70] copying algorithm as it has been recognized [Baker78], [Lieberman83] as being very well suited for real-time garbage collection.

Recall that Cheney's algorithm copies useful cells from a *fromspace* to a *tospace*, figure 2. It uses two indexes, a *visit* index *M* which points to the queue of copied objects not yet visited and a *copy* index C which points to the beginning of the free area where objects are copied.

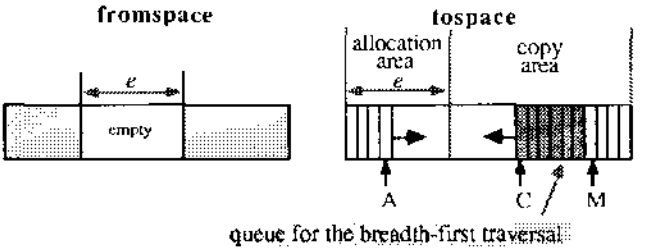


figure 2 : Visit and copy indexes for Cheney's algorithm

New objects are allocated in the allocation area in which index *A* gives the last allocated word. Allocation and copy areas grow in opposite directions. We have decided that C and M decrease towards lower addresses.

The size of the allocation area is bound by the size, *e*, of the previous empty area. Prolog system is suspended when reaching this boundary, until the current GC *batch* is finished. This is necessary to avoid deadlock on memory resource between Prolog system and its GC. At the end of a collecting batch, the allocation boundary is opened up to the C index until the next batch is started. A suspension of Prolog system corresponds to an abnormal functioning, in which the collector doesn't satisfy, in real time, Prolog system needs.

In the allocation area, objects are stacked as they are created. Hence, on backtracking this area may be instantly reclaimed. The instant reclaiming in the allocation area has proved to be sufficient for real-time GC, see discussion in §6.2.

3.2 Stratification of Prolog garbage collectors

Goal statements, the current one and those in the backtrack stack, can be marked independently. Doing so, the correct bindings is interpreted [Bekkers84a]. This allows the implementation of complete GCs which performs early reset of variables and variable shunting.

The garbage collection starts with the current goal statement, then it continues with active goal statements down through the stack, from the newest one to the oldest. Doing so, early reset and shunting of variables can be implemented very efficiently, requiring a time proportionnal to the amount of usefull objects.

4 Stratified Cheney's algorithm

Notation : in the following we use the notation (*F-L*) to refer line *L* of the program in figure *F*.

Two processes are involved, the mutator (Prolog) and the collector. At the beginning the collector is waiting for a signal (4-2) given by the mutator. A particular command, `Reduce`, figure 3, possibly starts the collector.

4.1 Starting the garbage collector

Between each resolution step, the state of Prolog is summarized within the three registers G, S and T ; Prolog executes the command `Reduce` to signal the collector that a batch can be started (3-7) with the three mentioned registers as roots (3-4). The command `Reduce` has no effect when the collector is working or when it is not enabled (3-2).

```
LEVEL * SM ; WORD * TM ;

1   Command Reduce() {
2       if (waiting(StartGC) && CollectorEnabled()){
3           "exchange from/to spaces" ;
4           G=NewVersion(G); S=NewVersion(S); T=Copy(T);
5           SM   S.info ;
6           TM = &(((TRAIL*) T.info)->next) ;
7           signal(StartGC)
8       }
9   }
```

figure 3 : The command Reduce

The collector is only *enabled* if a certain percentage of the memory is used, this is to avoid slowing down the mutator, see §6.1.

Memory overflow detection

One of the problems with realtime garbage collectors is to decide when memory overflow has occurred. If there is not enough memory the mutator must be suspended until the current collector batch is finished. Then if there is not yet enough memory, a new collector batch must be completed. If there is still not enough memory then the system is running out of memory.

It is impossible to start a collector batch at any time because roots of access are not always summarized in the intended registers. We have solved this synchronization problem with the `Reduce` command which defines the correct instants for starting the GC. Prolog is suspended inside this command if *there is not enough memory* to continue. With this solution, one has to decide of a maximum amount of memory which can be allocated between two executions of the command `Reduce`. Prolog is suspended if the free memory is smaller than this amount. For Prolog, this amount can be bounded by the size of the largest clause.

4.2 The garbage collector

4.2.1 The main loop of the collector

For each choice point, the collector copies its goal statement using Cheney's algorithm (4-4). Then, the trail is updated before going on with the next choice point. This is done by the procedure `UnTrailOneElt` (4-5).

For synchronizing reasons explained in §4.2.4, only one element of the trail is processed at a time ; the procedure is also in charge to prime the copy of the next level when the end of a segment trail is reached and to signal the end of the backtrack stack.

```
1   while (true) {
2       wait(StartGC) ;
3       while (true) {
4           CopyLevel() ;
5           if (UnTrailOneElt() != EndOfLevels) exit
6       }
7   }
```

figure 4 : main loop of the collector

4.2.2 Copying a goal statement

Copying a level, figure 5, consists in a simple loop to update the references within the segment of memory situated between indexes C and M. The loop ends when C=M. Of course, this process might involve copying new information in the copy area, this is done by the procedure `NewVersion`, figure 6.

Notice that not all words are visited (5-4). The procedure `AllowedVisit`, not detailed here, successes only if M does not point to the attribute of a bound variable or to the goal statement of a LEVEL. According to the stratification principle :

- the attribute of a bound variable is copied while updating its trail element (9-9), see §4.2.4.2,
- the goal statement of a level is copied when the collector primes the copying of this level (8-6).

```
1   CopyLevel() {
2       while (M != C) {
3           M = M - 1 ;
4           if (AllowedVisit)
5               exclusion (CopyObj) *M = NewVersion(*M)
6       }
7   }
```

figure 5 : copying an entire goal statement

4.2.3 Getting the new version of an object

The `NewVersion` procedure, figure 6, gives the reference to a new version of an object, either by copying the object (6-16), or by using its forward reference (6-12).

This procedure is also in charge of shunting variables (6-14) when they have been marked "shunted" (9-12).

```
1   WORD NewVersion(W)
2   WORD W ;
3   {
4       switch W.tag {
5       case T_atom, T_free,
6            T_trail, T_refvar, T_refvara :
7           return(W) ;
8       case T_cons, T_var, T_vara, T_level, T_age :
9           if (InToSpace(W.info)
10              return(W)
11          elseif (Forwarded(W))
12              return((W.info)->info)
13          elseif (IsShuntedVar(W))
14              return(NewVersion(((VAR*) W.info)->binding))
15          else
16              return(Copy(W))
17              }
18      }
19  }
```

figure 6 : give the new version of an object

Critical section for copying objects : the mutator and the collector are both involved in copying objects. Hence, the presence of the critical section `CopyObj`, (5-5), (8-5) for the collector and (11-3), (12-15) for the mutator. The exclusion also includes the update of the `WORD` pointed to by index M.

```
1   WORD Copy(W)
2   WORD W ;
3   {   int Size = SizeOfObject(W.tag) ;
4       if (W.info == nil) return(W) ;
5       C = C-Size ;
6       CopyBlock(Size, W.info, C) ;
7       *(W.info).tag=T_forward; *(W.info).info=C ;
8       W.info = C ;
9       return(W)
10  }
```

figure 7 : Copy and forward an object

4.2.4 Updating the trail

Once a goal statement has been copied, the trail segment it holds is copied and updated. Early reset is applied and shunted variables are marked *shunted*. The procedure tackles only one element at a time, the iteration for a trail section is done by the loop of the collector (4-3).

```
1   EOL UnTrailOneElt()) {
2       exclusion (Trail) {
3           if (SM = nil) return EndOfLevels ;
4           if (TM->info = nil) {
5               exclusion (CopyObj)
6                   SM->goal = NewVersion(SM->goal) ;
7               SM = (SM->next).info ;
8               TM = (SM->trail).info
9           }
10          else UpdateTrailElt() ;
11          return NotEndOfLevels
12      }
13  }
```

figure 8 : visiting a trail section

Critical section for manipulating trailing information : the trail may be pushed down by a `Backtrack` command, therefore, updating a trail element may not be done in parallel with such a command. Hence, the presence of the critical section `Trail`, (8-2) for the collector and (12-2) for the mutator.

The synchronizing problem happens when the mutator backtracks to a choice point which has not yet been visited by the collector (12-14). In that case the collector must stop its untrailing process, because trail elements are no more significant. When such a *deep backtrack* occurs (12-14), the backtrack stack seen by the collector is forced to shrink (TM and SM registers are updated (12-19)) and the `CopyLevel` process is primed with the roots of the recovered level (12-16). After the `Backtrack` command, the next iteration of the collector loop (4-3) will see a nonempty queue between C and M and will pursue the copying of this level even if it was currently tackling a trail section.

4.2.4.1 Implementation of early reset and variable shunting

When updating a trail element three cases must be considered :

(1) useful binding (the variable has already been copied) :
  (1.1) useful free variable (the variable is older than the next choice point) :
  the trail element is copied to be kept in the trail; for an attributed variable a new version of its attribute is obtained, figure 10;
  (1.2) useless free variable (variable is younger than the next choice point) :
  the trail element is discarded and the variable is marked *shunted*;
(2) useless binding (the variable has not been copied):
the trail element is discarded and the variable is reset *free*.

```
1   UpdateTrailElt()
2   {   WORD RefVar ; LEVEL * AgeOfVar ;
3       RefVar = (TM->info)->var ;
4       if (Forwarded(RefVar)) { /*(1)*/
5           RefVar.info = (RefVar.info)->info ;
6           AgeOfVar = (((VAR*) RefVar.info)->age).info
7           if (AgeOfVar != SM) {    /*(1.1)*/
8               exclusion (Copy) TM* = Copy(TM*) ;
9               UpdateAttribute(RefVar)
10          }
11          else {                   /*(1.2)*/
12              SetShunted(RefVar);
13              TM* = (TM->info)->next
14          }
15      }
16      else {                       /*(2)*/
17          ((VAR* RefVar.info)->binding).tag = free;
18          TM* = (TM->info)->next
19      }
20  }
```

figure 9 : visiting a trail element

Attributes of shunted variables (9-11) does not need to be updated, this is because such atributes will never be accessed.

4.2.4.2 Copying attributes with the correct binding environment

The attributes of bound variables are not updated in a standard way. This is to avoid copying an attribute with an incorrect binding environment. Atributes are copied while visiting the trail (9-9).

```
1   UpdateAttribute(RefVar)
2   WORD RefVar ;
3   {WORD *RefAttrib ;
4       if (RefVar.tag == T_refvara)
5       exclusion (CopyObj) {
6           RefAttrib=&(((VARA*)RefVar.info)->attribute);
7           *RefAttrib = NewVersion(*RefAttrib)
8       }
9   }
```

figure 10 : updating the attribute of a bound VARA

4.3 Examples of commands

To illustrate the interactions between the collector and the mutator, some examples of commands are given, an access command (`Left`), the backtrack command (`Backtrack`) and a binding command (`BindVar`).

4.3.1 How the mutator is involved in copying objects

The command `Left` returns the left component of a binary "cons" term. In case this component is a reference to an object still in the fromspace, the command applies the `NewVersion` procedure. Therefore, Prolog system never gets any reference into the fromspace and the allocation area will never contain any reference to the fromspace. Of course, the NewVersion procedure may enforce the copy of an object.

```
1   Command WORD Left(Cons)
2   WORD Cons ;
3   {   exclusion (CopyObj)
4           return(NewVersion((CONS*) Cons.info)->left)) ;
5   }
```

figure 11 : Accessing a cons, the command `Left`

4.3.2 Synchronizing problems while backtracking

The command Backtrack retrieves the top of the backtrack stack. Due to concurrent use of the trail, this command is executed under the Trail exclusion. Variables belonging to the current trail section are reset and registers G, S and T are updated. If the collector has not yet started the copy of the recovered choice point, (12-14) this is what we have called a *deep backtrack* condition, §4.2.4, the backtrack stack seen by the collector is forced to shrink (12-19).

```
1   Command Backtrack() {
2       exclusion (Trail) {
3           TRAIL * TT = (((TRAIL*) T.info)->next).info ;
4           while (TT != nil) {
5               VAR * Refvar = (TT->var).info ;
6               (RefVar->binding).tag = free ;
7               TT := TT->next ;
8           } ;
9           LEVEL * ChoicePoint = S.info ;
10          G = ChoicePoint->goal ;
11          Clause = ChoicePoint->clause ;
12          S = ChoicePoint-> next ;
13          ((TRAIL*) T.info)->next = ChoicePoint->trail
14          if (ChoicePoint == SM) {
15              exclusion (CopyObj) {
16                  G = NewVersion(G) ;
17                  S = NewVersion(S) ;
18              }
19              SM = S.info ;
20              TM = &(((TRAIL*) T.info)->next)
21          }
22      }
23  }
```

figure 12 : Backtracking operation, the command Backtrack

4.3.3 An example of allocation : binding a variable

This command BindVar creates an empty trail element requiring two words in the allocation area. At the beginning, there is a check (13-4) to see if there is enough space in the allocation area. The test is meant to always succeed because the system is supposed to be suspended inside the command Reduce if space is insufficient (see §4.1).

```
1   Command BindVar(Var, Term)
2   WORD Var, Term ;
3   { WORD * NewT ;
4       CheckAllocation(2);
5       ((VAR *)Var.info)->binding = Term ;
6       ((T.info)->var).info = Var.info
7       (++A)->tag = T_refvar ; /* creating a new empty */
8       A->info = nil ;         /* trail element  */
9       NewT = A ;
10      (++A)->tag = T_trail ;
11      A->info = T.info ;
12      T.info = NewT.info ;
13  }
```

figure 13 : Binding a variable, the command BindVar

5 Implementations of Realtime garbage collectors

We have implemented two kinds of realtime garbage collectors, a pseudo-parallel and a parallel one.

5.1 pseudo-parallel garbage collectors

The pseudo-parallel GC emulates two processes on a single processor. The first process is the Prolog system and the second is MALI garbage collector. The code for both processes is written in C. A corroutining mechanism provides sequencing between the two processes. Garbage collection is done incrementally : the smallest grain of work is either an iteration of the `CopyLevel` loop (5-2) or an execution of procedure `UnTrailOneElt` (figure 8). These steps of garbage collection may be done each time a mutator command requires memory allocation. Therefore the garbage collecting time is spread along Prolog interpretation time, but the two processes never work really in parallel.

In this implementation, the `Copy` and `Trail` exclusions are naturally realized by the chosen step of garbage collection.

5.2 parallel garbage collectors

To implement our parallel GC, we have chosen a private memory architecture, figure 14 : one processor, MALI, supports the garbage collector together with the commands and has exclusive access to Prolog memory state. The other processor, the host, supports Prolog interpretation. The two processors communicate via a small shared memory. Processor MALI is a specially microprogrammed processor board to be installed in an IBM PC compatible computer.

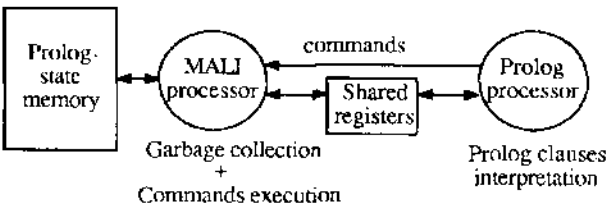


figure 14 : a MALI processor architecture

The GC process runs as a background task. Commands sent by the host are processed by the mutator as interruptions.

The `Copy` exclusion is realized in the GC code by disabling interruptions of the micromachine. Commands are delayed during the copy of objects, but this latency is reasonably short, see §6.1. The `Trail` exclusion is implemented with boolean variables plus a hardware mechanism to suspend and restart a command. The time spent by the collector to process a trail element is long enough to justify such a selective mechanism for implementing the `Trail` exclusion.

6 Discussion and results

The parallel garbage collector has been implemented on a PC AT with a 5MHz 80286 interpreting Prolog and a 6MHz microprogrammed processor built around an AMD 2916 supporting MALI. Compared with a software version of MALI implemented in C on the same PC having a serial garbage collector, this parallel version showed a speedup factor of 2. This result is essentially due to the fact that the commands, which roughtly represents 50% of the total execution time with the software version of MALI, are executed on the specialized hardware which is about 10 time faster for that job.

The pseuso-parallel collector has been written in C and is portable with Prolog on any computer. The relative speed of the collector with respect to the mutator is tuned by giving the number of collector steps executed for each allocated word. The best execution time which has been observed with this implementation is 30% longer than that of the serial collector implementation, and it was obtained for a speed ratio less than one step per allocated word. Generally, for programs requiring 70% or less of the memory, a real-time behavior is obtained for a collector speed between 1 and 2 steps per allocated word. As expected, speed degradation increases with collector speed, because useless collecting work is done.

6.1 Interferences between a parallel collector and the mutator

When the parallel collector is running the speed of the mutator is somewhat reduced by two agents.

- The latency of execution of the commands is increased because the collector must complete the atomic action it is currently doing. In our implementation, this latency has been measured to be about 10% of the command execution time.
- As we have said earlier, the mutator may be forced to copy objects. The time spent for these copies has been measured indirectly and represents between 10% to 15% of the command execution time.

Therefore, the command execution time is around 20% slower when the collector is running. In our system, due to the specialized implementation of commands, MALI commands represents only 10% to 20% of the total execution time of Prolog. So the overall degradation induced by the parallel collector is only 3% to 4% of the total system performances.

6.2 Instant reclaiming on backtracking and realtime garbage collector

In a conventional Prolog system, space is immediately reclaimed on backtracking. With our GC, this reclaiming is not entirely applicable because the order of objects in memory is modified by Cheney's copying technique. On backtracking, we can only reclaim memory in the allocation area, where objects are still located in creation order.

The expected normal functioning, also called stable functioning, of such a collector is when the collector is fast enough to avoid suspensions of the mutator for allocation reasons. In our system, the immediate reclaiming in the allocation area has proved to be amply sufficient to slow down apparent consumption of memory so that the garbage collector follows in real time the mutator needs.

7 References

Appleby88. Appleby, K., Carlsson, M., Haridi, S., and Sahlin, D. Garbage collection for Prolog Based on the WAM. *Communications of the ACM 31*, 6 (June 1988), pp. 719-741.

Baker78. Baker, H.G. List processing in real time on a serial computer. *Communications of the ACM 21*, 4 (April 1978), pp. 280-294.

Barklund87a. Barklund, J. A Garbage collection Algorithm for Tricia. Tech. Rept. 37B, UPMAIL, Uppsala University, Sweeden, December, 1987.

Barklund87b. Barklund, J. and Millroth, H. Hash Tables in Logic Programming. In *Proceedings of the International conference on Logic Programming, ICLP87*, Lassez, J.L., Melbourne, Austria, MIT Press, May 1987, pp. 411-427.

Bekkers84a. Bekkers, Y., Canet, B., Ridoux, O., and Ungaro, L. A short note on garbage collection in Prolog interpreters. *Logic Programming News letters* , 5 (1984).

Bekkers84b. Bekkers, Y., Canet, B., Ridoux, O., and Ungaro, L. A memory management machine for Prolog Interpreters. In *Proceedings of the second international Logic Programming conference*, Uppsala, Sweden, July 1984, pp. 343-353.

Bekkers86. Bekkers, Y., Canet, B., Ridoux, O., and Ungaro, L. A memory with a real-time garbage collector for implementing logic programming languages. In *Proceedings of the second International symposium on Logic programming*, IEEE, Salt-Lake City, Utah, 1986, pp. 258-265.

Bekkers88. Bekkers, Y., Canet, B., Ridoux, O., and Ungaro, L. MALI, A memory for implementing logic programming languages. In *Programming of future generation computers*, K. Fuchi, M.N., Tokyo, Japan, Elsevier Sciences Publishers BV North Holland, 1988, pp. 25-34.

Brisset91. Brisset, P. and Ridoux, O. Naive reverse can be linear. In *Proceedings of the international conference on Logic programming*, Paris, France, June 1991.

Bruynooghe84. Bruynooghe, M. Garbage collection in prolog interpreters. In *Implementations of Prolog*, J. Campbell, E.H., 1984, pp. 259-267.