The Garbage Collection Question in Logic Programming

Technical Report GMD 06/89

Raúl Rojas
GMD/FIRST
Hardenbergplatz 2
1000 Berlin 12

The Garbage Collection Question in Logic Programming

Raúl Rojas
GMD/FIRST
Hardenbergplatz 2
1000 Berlin 12

Abstract

We examine in this paper the garbage collection problem in Prolog. The peculiar semantics of Prolog, with its single assignment rule, imposes a heavy burden over the processor of a traditional von Neumann computer. The memory management has to be done by the processor and not by the programmer: this makes programming easier but the programs could be executed innefficiently if no appropriate memory management method is used.
We discuss the garbage collection strategies used in Lisp systems and in which way they could be adapted for a Prolog system. We give several alternatives for the implementation of a garbage collector in Prolog, which go from the very simple (garbage collection done by the programmer) to the very elaborated (a system of pointers to segments). The decision over which method should be used depends then of the type of computer or architecture which is to be employed.

1. The problem

The great majority of Prolog compilers and interpreters work with very inefficient memory management. This is not a problem of careless programming but of the peculiar semantics of Prolog, which is very different to the semantics of other languages. We can illustrate this point with an example.
Consider the following clause

```
a:- b(X),c(X),d(Y).
```

In a conventional procedural language we could collect the memory cell occupied by X just before the call to predicate d, because we know that in d the value of X will not be used. That is not always the case in Prolog. If the predicates b and c left choice points behind, we can not collect the memory cell occupied by X: we must wait and see if no backtracking occurs. The only case in which we can collect the memory cell occupied by X is when no choice point exist between "creation" and "annihilation" of the this variable.

2. Two new predicates.

It is possible to discuss the garbage collection problem more easily if we introduce to new predicates "create" and "destroy". The first predicate allocates a memory cell for a new variable in a clause. The second reclaims the storage used by a variable. For example, the clause:

```
a:- b(X),c(Y).
```

could be written so:

```
a:- create(X), b(X), create(Y), c(Y).
```

In this way we just make explicit that which is always implicit in every Prolog interpreter: before we can use the variables X or Y, we have to allocate some memory for them. We would like to write also:

```
a:- create(X), b(X), destroy(X), create(Y), c(Y), destroy(Y).
```

As already said this is not always possible.

Just note the following points:

- At the time of creation only a memory cell is allocated for every variable.
- At the time of annihilation more than a memory cell could be collected, because the variable could be instantiated to a structure or a list.

3. A pathological example.

Consider the quick-sort program:

```
qsort([H | T],S,X):- split(H,T,A,B),
                     qsort(A,S,[H | Y]),
                     qsort(B,Y,X).
qsort([],X,X).
```

In this program after the tail of the original list was splitted in the sublists A and B, we no longer need it. It would be better if we could write:

.... split(H,T,A,B), destroy(T) ....

All the information that we need for the sorting process is indeed already contained in the sublists A and B. Conventional Prolog interpreters will not annihilate T because of several possible reasons:

- In interpreters without a "switch"-capability (like that of a WAM) the qsort predicate leaves fast always a choice point behind (for the second option). Without an analysis of the second clause is not possible to know if T will be required again or not.
- The interpreter does not know if the lists T, A and B "share". Depending on the way that the split predicate works and depending on the structure sharing methods of the interpreter, it could be the case for A to be the same list as T (with B=[]). If we collect the memory cells of T too early we could destroy the information contained in A. The program would crash.

Because of this T will be preserved in every iteration, that is, a copy of the tail of every generated sublist will be stored with every recursive call. If we start with a list of length 2^{10} we are going to split the original list about ten times. In every step we are going to left behind a copy of the old information. That means that we need $10*2^{10}$ storage cells for qsort to work. If we want to sort a megabyte of integers, we should better be sure that we have 20 megabytes of storage available!.

In practice the storage requirements of even "respectable" Prolog interpreters are much worse than this. Our IF-Prolog interpreter uses 256Kb of global stack just to sort 200 integers!.

4. The garbage collection facilities of the WAM.

The Warren-Abstract-Machine does theoretically much better garbage collection than naive Prolog interpreters. As a matter of fact, David Warren built garbage collection as an inherent feature of the WAM, but the

associated problems are not trivial and it is not surprising that fast nobody implements garbage collection in practice.
Storage management optimization is achieved in the WAM with the use of argument registers. If a temporary memory cell is needed, then an argument register can be used to hold a new variable or a pointer to other data structures. The argument registers can be overwritten and in this way a simple form of garbage collection can be implemented.
Sometimes this overwriting capability of the WAM is not desirable, for example, when a specific value has to be kept for later use. In this case one has to define environment storage cells where it is possible to save "permanent" variables. Warren has provided a method to identify superfluous environment cells that are trimmed with every procedure call. He calls this method "last call optimization", a generalization of tail recursion optimization.
Take a look at this clause:

```
a(X,Y) :- b(X), c(Y), d(Y).
```

If we suppose that "global" argument registers (A1,A2, ...) are present and that all procedures become their arguments trough this registers we could rewrite this clause like:

```
a(A1,A2) :- Y1:=A2, b(A1), A1:=Y1, c(A1), d(A1).
```

Note:

- Predicate "b" gets its argument direct from "a" trough register A1.
- Argument register A2 must be saved in a local variable Y1, because argument registers are global and can be overwritten in procedure calls.
- After A2 was "refreshed", the local variable Y1 is no longer necessary. It can be destroyed safely but only in the case that "b" left no choice points behind.

If we suppose that the presence of argument registers is implicit in the procedure calls (call a/2 is equivalent to a(A1,A2), etc), we could rewrite the clause above as:

```
a/2 :-      create(Y1)
            move (Y1,A2)
            call b/1
            move (A1,Y1)
```

```
destroy (Y1)
call c/1
call d/1.
```

Here create and destroy have the already described meaning. "Move" is only a destructive assignment, possible with global registers.
Actually Warren has embedded the semantics of "create" in his "allocate" instruction. Allocate reserves memory cells for the needed local variables that are to hold "permanent" values and makes other housekeeping functions. "Destroy" is implicit contained in the instruction "call". This instruction trims the local environment before a procedure call is executed (only in the case that no choice points make this trimming impossible).
The only possible inefficiency with this scheme is that the storage for all required local variables is allocated at the beginning of the clause (with a single "allocate") and not only when it is needed. This is nonetheless more as compensated with the execution time savings of the simpler method.
As a matter of fact, the WAM is an optimal storage allocator in the case that no structures are used (we discuss the case of unsafe variables later on).

5. Stacks are beautiful

The WAM uses three different stacks in order to manage the Prolog computation rule. The local stack contains all of the control information: environments and choice points. The global stack is used to hold structures and "unsafe" variables. The trail stack hold the addresses of memory cells which should be reset on backtracking.
This usage of multiple stacks makes backtracking very simple: in order to backtrack it is only necessary to reset the pointers of the three stacks to its original value (as stored in the choice point). In the case of the local and global stacks this can be made immediately. In the case of the trail, the stored addresses have to be popped and the corresponding memory cells reset.
In the case of frequent backtracking no storage allocation problem arises: in each backtracking step memory is being freed. The three stacks come back to their original starting point and everything is ready for a new forward computation.
If the objective is to optimize backward computation, then stacks are the optimal solution. It is difficult to realize a faster reset strategy than resetting two pointers (global and local stack) and popping off the rest of the pointers

to memory cells which additionally need to be reset (because they lie beyond the current choice point in the local or global stack).
It is because of some of this features that the WAM has become so popular: the registers perform some kind of garbage collection at no cost, they make the parameter passing faster, the deallocate feature maintains a minimal environment and the used stacks optimize backtracking.
There is only a big problem with the global stack: there is no simple way to implement garbage collection in it.

6. Stacks are difficult to compress

In a better world like this, a world with recursive architecture computers, there would be no need for a global stack to hold structures. If we represent a memory cell with the Prolog structure **cell(Tag,Value)** and if we suppose that every argument register is capable of holding one of this "cells", we could then have simple values like

A1 = cell(constant, [])

other

A1 = cell(integer, 13)

contained in the first register. A recursive structure, like the list [a,b], could be stored like

A1 = cell(list, '.'/2(cell(atom,a),cell(list,'.'/2(cell(atom,b),cell(const,[])))))

Such simple recursive storage of terms must be simulated in the memory of the computer . Every structure has to be stored in the global stack and only a pointer to this structure can be allocated in the argument register (with the tag "ref"). Many problems arise with this representation method.
Take the case of quicksort. At the beginning of the whole computation we have a list of n elements in the global stack and a pointer to it. The list is splitted in two new lists and the new lists are allocated below the old list in the global stack, before qsort is called again with the new sublists as arguments. At this point the WAM knows that it does not need the old list anymore but it can not deallocate it from the global stack, because other structures are just below it. If we deallocate the original list, then we have to compress the global stack and redirect all pointers in the local and global

stack that pointed to addresses below the deallocated structure. That is no easy matter, because we do not know which pointers are pointing there.
But it could be worse. Suppose that two lists A and B are stored in the global stack. Because of the way they were constructed, it could be that the first element of A is followed by the first element of B. After that comes the second element of A and then the second element of B and so on. We could know that A is no longer of any use and that B needs to be kept. We could deallocate all the elements of A but in order to get this storage back we have to compress the global stack and redirect all the pointers of the elements of B to new addresses. In this case the collection of the storage used by A is very expensive in computing time. Think for example of a list with 1000 elements: 1000 pointers should be redirected and 1000 elements should be copied back into upper positions in the global stack.
Stacks are fine for backtracking: they are terrible for storing loosely coupled structures, like those which arise in Prolog.

7. The garbage collection vision.

We would like to exploit the potential garbage collection capabilities of the WAM, but we have several constraints:

- Backtracking should be so time-efficient as possible, like in the classic stack oriented WAM.
- Backtracking must free storage cells fast immediately
- The execution overhead of garbage collection should stay below excessive values

There are several alternatives for implementing garbage collection in a WAM. We proceed to discuss several of them.

8. Naive Mark and Sweep.

Mark and sweep is the least attractive of all the garbage collection techniques. It is collection of storage by brute force.
Mark and sweep could be implemented in a WAM in the following way:

- stop processing
- traverse the local stack and mark all used structures in the global stack

- traverse all marked structures in the global stack and mark other used structures in the global stack ("scavenging")
- compress the global stack while redirecting all pointers to it

9. Clever Mark and Sweep

A variant to the brute force alternative is a dynamic-static variant of mark an sweep. Each time that a "call" instruction deallocates a pointer to a structure it could be marked as "unused". By stopping processing, we could delete all unused structures and compress the stack.
This technique has only a problem: we don't know exactly how many pointers are directed to a structure. If we deallocate one of this pointers, we have no guarantee that other pointers are not pointing to this very same structure ("aliasing"). Even worse: we don not know if a part of this structure is being pointed by some variables (the tail of a list or an argument of a structure, etc).
The only way to make this technique feasible is to store with each structure a counter of references to it. Only when the counter were zero, would it be safe to delete the structure from the global stack.
As it turns out, the biggest problem is that even when we know which structures should be deleted, the time needed for the reorientation of the pointers to the global stack could be prohibitive. We don't want the users to sit there and stare at the computer when it makes garbage collection. We want a responding, efficient system. The only way out seems to be letting the global stack drop and substitute it with a real heap.
In the original description of the WAM, Warren speaks of a "heap" to hold structures. Actually the heap is in fast all implementations no heap but a stack. A good idea could be to see what happens when the heap is a real heap.

10. Reference Counting

A heap is an area of memory divided in storage blocks of equal or different sizes. A block can be requested to store a part of a list or a structure and with the help of pointers is possible to link several blocks to hold the whole structure.
If blocks of different sizes are used in the heap, one word of storage has to be reserved to keep track of the size of the block. One problem with this scheme is that when later a block is freed and collected, this block can only

be used to store an element with the same storage requirements: for example, if the block contains 5 storage cells but 7 are needed for a new structure, then the old block can not be reused. If, on the contrary, only three cells are needed it must be decided if two cells could be wasted or if the original block of size 5 could be broken apart in two smaller blocks. Eventually the fragmentation of the heap will demand a compression and reordering of the used blocks in order to collect unused storage. This "mark and sweep" step could be delayed longer as in the case in which no real heap is used, but it will be needed sooner or later.
A better alternative is to allow only blocks of the same size to be used. When all blocks are equal, a collected block can be reused immediately after it has been collected. There is no fragmentation of the heap and the compression step is not needed. This strategy has only two problems:

- The locality of the data is not guaranteed. When the computation advances in time the allocated blocks will be more randomly distributed over the whole heap area. This is a potential problem in those applications in which virtual memory should be used (because of the size of the problem or because of the reduced storage available). The random distribution of the used structures could produce a lot of time-costly page faults.
- If all the blocks are to be of the same size, then in some cases part of the block will not be used. Some of the heap area will be wasted. It is important to know in advance how much of the heap memory could be wasted in this way.

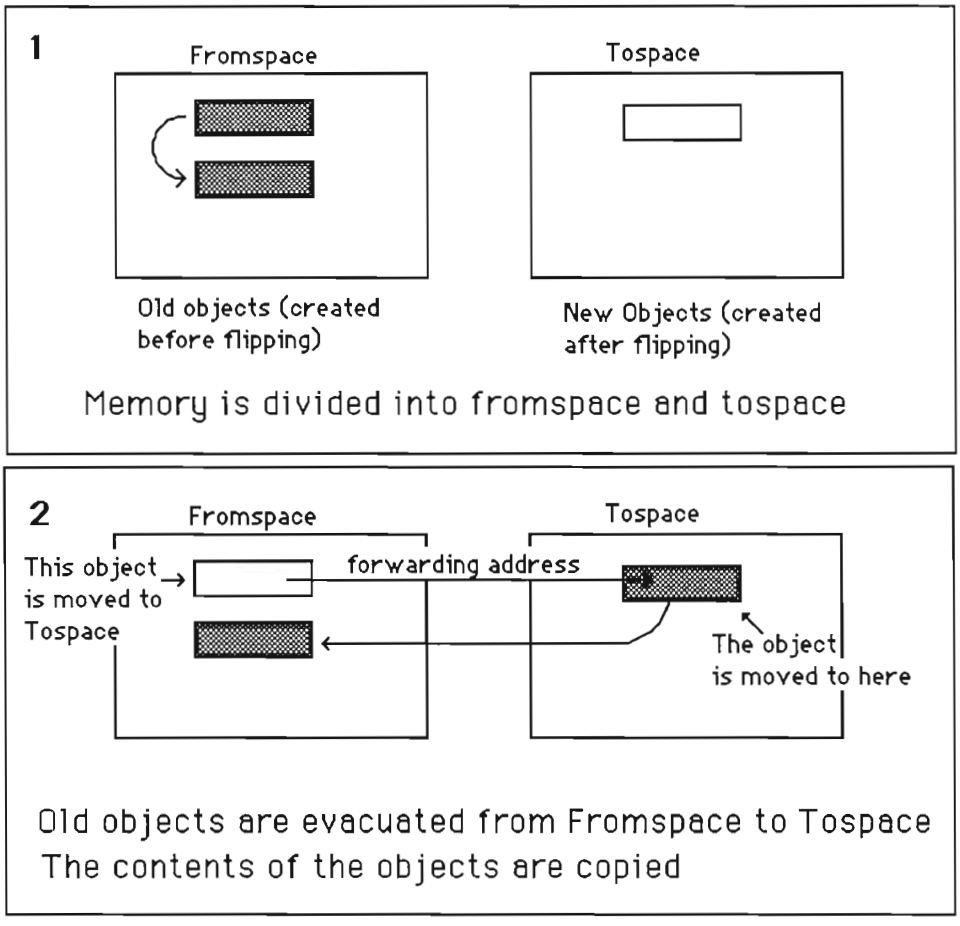
The biggest problem with every reference counting strategy of this type is to keep track of the references to objects. In the WAM, a pointer to an object in the global stack can be stored in a register and nowhere else. If the register is overwritten, we should then decrement the reference count of the pointed object. That implies that every time the contents of a register are changed, we must check to see if no reference to the global stack is being destroyed. The overhead incurred in this checking could be very sensible because the argument registers are being changed in every head matching and being restored on each fail. On backtracking it would be impossible to reset only the pointers to the local stack. Backtracking should instead pop each stored element in the local stack to see if no references to the global stack were being skipped. A popped element could be deleted only after decrementing the reference count of the pointed object. An additional problem is that collected memory cells should be managed in certain way. A straightforward technique could be to form a list of free memory blocks. Every time a structure is collected, the blocks that it occupies should be hanged to the free

list. But if the deallocated structure has a complicated form (a tree or something similar), it is necessary first of all to unravel the structure by traversing it, in order to reorder the used blocks in a serial list. Obviously a garbage collection which must always flatten a collected structure in order to hang the freed blocks to the free list can not be very efficient.
Another problem with reference counting is that a circular structure can fool the garbage collector in making it believe that an inaccessible structure continues to be pointed by an element in the local stack. This case should not arise in a Prolog interpreter with occur check in the unification procedure, but as a matter of fact the great majority of current Prolog implementations tolerate infinite recursive structures or they even encourage them (like in Colmerauer's Prolog II).
Because of all of this reasons (and another more) we drop reference counting as a viable strategy.

11. Baker´s algorithm

We describe now Baker´s garbage collection algorithm, because it is the basis for more complex algorithms used in computers like the Symbolics.
Baker's algorithm is a "copying" algorithm, that is, the elements of the heap that should be preserved are copied to a new memory area on each full cycle of the garbage collector and the objects that are garbage are not copied. In this way only accessible objects are copied and not accessible objects disappear of the new memory area. This method has two advantages:
• All what is needed is a list of accessible objects. Everything that is not accessible is considered to be garbage. The list of accessible objects is readily available in the WAM in the local stack (the contents of the argument registers should be considered too).
• The accessible objects can be compacted on copying, making the use of virtual memory techniques more efficient, because the probability that a list or structure can be allocated in a single memory page increases.
• The garbage collector can run incrementally and in parallel to the "mutator" (the program that creates new objects). This is specially important if garbage collection is to be performed in "real time" by another processor or by specialized hardware.
Baker´s algorithm works in the following manner: The heap manager allocates objects in memory without taking care of collecting any garbage until it is decided that garbage collection should start. The area of memory where the allocated objects reside is called "Fromspace". A big enough new

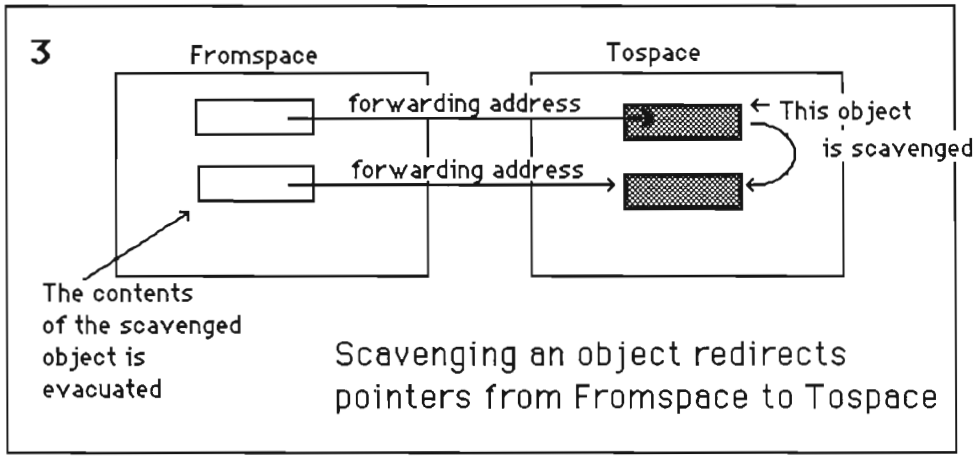
area of memory is provided and it is called "Tospace". The process in which this partitioning of the computer memory occurs is called "flipping".
The garbage collector must now evacuate all accessible objects in Fromspace taking them to Tospace and leaving the garbage behind. The address of the first accessible object in Fromspace is taken from the list of accessible objects and the first memory cell is evacuated:



After a memory cell has been copied to Tospace, a pointer to this new location is stored in the old location. This is called a "forwarding pointer" and serves to purposes:

- If locations that are to be evacuated contain a pointer to this removed object, than it is possible to reorient this pointer to the new location by copying the value of the forwarding pointer.
- The mutator (the real program) can work normally and doesn't perceive the garbage collection process. If a copied element is referenced by the mutator, the forwarding pointer is only another extra level of indirection for the dereferencing process.

After a memory cell is been copied to Tospace, it is "scavenged", i.e. it is analyzed and seen if it contains a pointer to Fromspace. If this is the case the pointed object is copied to Tospace. A forwarding pointer is left behind and the original pointer to Fromspace is changed to point to the new location in Tospace:



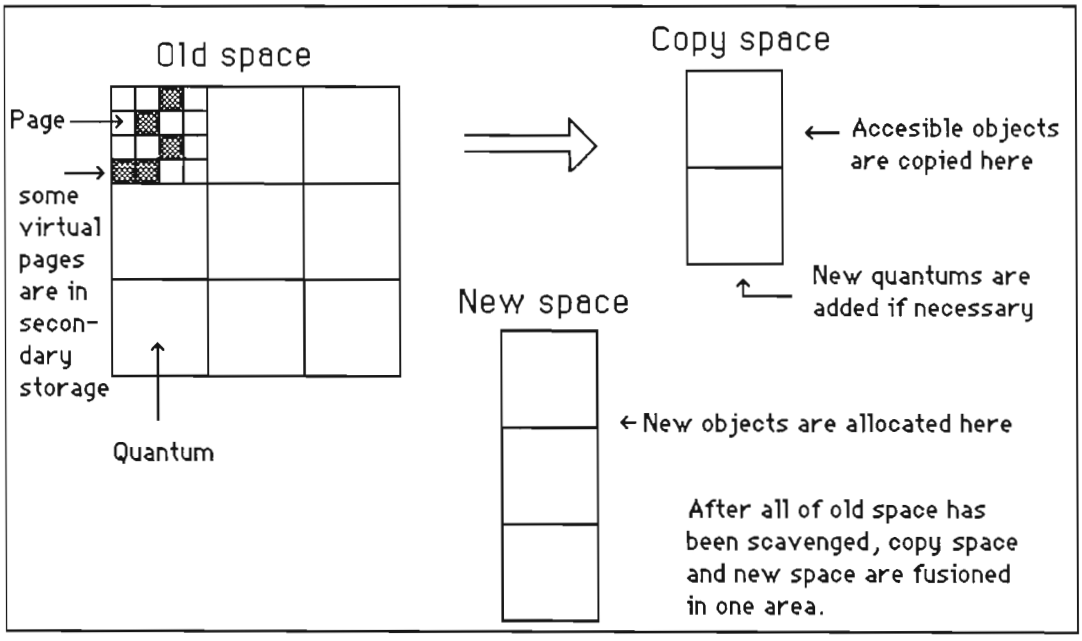
The scanvenging is repeated until an object is copied that doesn't contain a pointer to Fromspace. In this moment a new accessible object is taken from the list of accessible objects and is copied to Tospace. The scavenging process begins anew and so on. When the dust settles, all accessible objects have been copied to Tospace and the whole of Fromspace can now be safely declared free memory.

What happens if the mutator needs to allocate new objects when the garbage collection is going on? Baker proposes to allocate new objects in Tospace, possibly at the bottom of this area or interleaved with the accessible objects been collected. It should be guaranteed only that all new objects does not contain pointers to Fromspace (which we want to declare free memory). This condition could be fulfilled if every time a new object is allocated it is inspected to see if it contains a reference to Fromspace. The referenced object should be evacuated to Tospace and the pointer in the object should be redirected to point to Tospace.

12. Garbage Collection in the Symbolics

The Symbolics Lisp-Workstations work with a refined version of Baker's algorithm (that is not surprising: Baker is a researcher at Symbolics). It is well known how important it is for Lisp to count with an efficient garbage collector. Old Lisp implementations had the same problems with garbage collection as the modern Prolog interpreters. With the time a whole set of

techniques for garbage collection in Lisp machines has emerged and the algorithms used in the Symbolics should be considered as the state of the art in this field.
The Symbolics works with virtual memory. Typically the virtual address space is thirty times the physical address space and the virtual storage is divided in pages of 256 words (of 34 bits each). The virtual memory system administers a table of the pages that are located in the physical memory and those that are relegated to secondary storage.
For the garbage collection process the address space of the Symbolics is divided in three regions, instead of two. The first of them is the "Fromspace" region of Baker´s algorithm, which is called "old space". The second is the "Tospace" region of Baker, which is called "copy space". The third is called the "new region". In old-space are located all the objects created until the time that it was decided to flip spaces. Accessible objects located in old-space will be collected in copy-space by copying. All new created objects are not stored in copy-space but in new-space. This is a further optimization of the algorithm of Baker because old objects will not be interleaved with new objects. Another optimization is a further division of the memory space of the Symbolics in "quantums" of 16Kb each. When the time of collecting the accessible objects in old-space arrives, a new copy space is allocated, but this copy-space consists only of a quantum. If new memory is required additional quantums are appended to the copy-space. It is not necessary for the virtual address of consecutive quantums to be consecutive too. A table of the allocated quantums to one of the three described areas is maintained by the hardware and software of the machine.



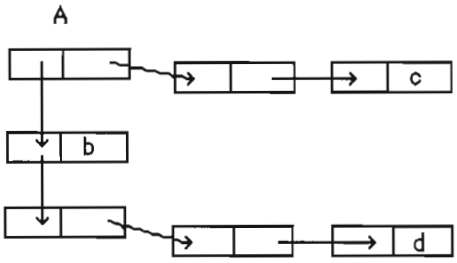
For the copying process the Symbolics-Algorithm works in a similar way to the original Baker method, but there exist three concurrent processes instead of two: the mutator is the program that creates new objects in new space, the scavenger scans the evacuated objects in copy-space looking for pointers to old-space and the transporter copies objects from old space to copy space leaving a forwarding pointer behind and giving as result the new address of the copied object in new space. The mutator and the scanvenger work in parallel. Each time the mutator creates a new object in new space, the scavenger takes control and performs several copy steps, after what it gives the control back to the mutator. In this way it is guaranteed that the storage reclamation process does´t fall behind the creation of new objects. This avoids an explosion of the memory requirements. The transporter is called by the scavenger each time that it wants to evacuate an object from old space to copy space. The transporter is called also by the mutator every time that it accesses an object in old memory. The accessed object is copied and the new address is stored in new space. In this way it is guaranteed that in new space no pointers to old space are stored and the scavenger does not need to scan this area of memory after it is done with the scanning of copy space. The scavenger is called also in every pause of the mutator, for example when it waits for input from the terminal or when it is waiting an I/O operation to the hard disk.

This division of labor between three different processes opens the possibility that they be mastered by different hardware components. In the Symbolics the main processor distributes its time between the mutator and the scavenger whereas the transporter is emulated in hardware. Every time that a storage cell in old-space is accessed, a hardware trap occurs. The mutator copies the accessed object and gives his new address to the mutator or to the scavenger. In this way a lot of the garbage collection process is made transparent for the systems programmer.

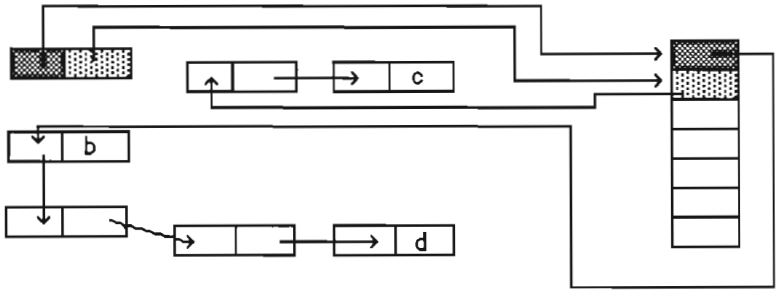
It could be possible to employ a special processor only for the scavenging of copy-space. It could work in parallel with the main processor and perform other housekeeping functions. The only complication would be the contention of the two used processors for access to the bus. A multiple bus architecture or other methods could solve this problem.

This is the essence of the garbage collection method used in the Symbolics, but there are other details that should be considered. The fact that the Symbolics is a virtual memory machine makes very important to compact the storage allocated to different objects, so that they pass into a single page of virtual memory if that is possible. Lists that contains lists should be copied in a depth-first manner, because they are accessed in this way. With

no changes the used algorithm tends instead to copy lists in a breadth first manner.
Consider the following list:



When this list is copied from old space to copy space, the first cons-cell is copied first. The new cons-cell contains two pointers to old space: one to the continuation of the list and the other to the car, itself a list:



When the copied object (the cons cell) is scavenged, the first pointer points to the continuation of the list and the second to the CAR. The scavenger would then copy first the continuation of the list and then the CAR. The CAR-sublist would not be compacted by the copy process, on the contrary it would be interleaved with the rest of the list, a possible source of inefficiency for the virtual memory system (if no virtual memory is used, then it does't matter how the objects are copied).
There is no simple solution to this problem when we don't want to use a recursive routine (a recursive routine needs a stack and this produces new storage allocation problems) and when we want to collect accessible objects in real time, but the Symbolics employs what is called approximate depth first copying . In this scheme, the scavenger scans through copyspace looking for references to oldspace, it always scans the partially filled page at the end of copyspace first. Any object in oldspace pointed by objects in this page is evacuated to copyspace and then the scavenger continues with the objects at the top of copyspace. If a new object is evacuated, there is a new partially filled page and attention is redirected to it. If no pointer to oldspace exists in this page, the scavenger resumes where it interrupted, until all of copyspace has been scanned. This method has only the drawback that some

locations will be scanned twice, but that is more than compensated with the increased locality of the structures in copy space and the incurred overhead is almost always negligible.
Actually garbage collection in the Symbolics is more complex as here explained, because the Symbolics differentiates between ephemeral and static objects. The ephemeral objects are more likely to become garbage and so they are handled in a special manner. The garbage collection process is tuned so that ephemeral objects are copied more often than other objects, guaranteeing that storage will be freed at a higher pace.

13. A Garbage Collection Strategy for the WAM

There are several possible garbage collection methods that could be applied in a WAM.
The first possibility is not to do garbage collection at all. If virtual memory is used, we could provide for a big enough virtual address space, so that a program which does a long forward computation will not run out of address space. If the locality of the data is guaranteed (that is, if the program creates constantly new structures, that are nevertheless tight packed in memory) this will not impose a very big penalty on the execution time of the program. Eventually the user could run out of space and in this case a "mark and sweep" garbage collection step could be performed.
The second possibility is to do garbage collection using virtual address space and a Baker-type copying algorithm. Like the classical Baker-Algorithm, we divide the address space in Fromspace, Tospace and Newspace (only in the case of the global stack). The copying process starts in the local stack. All the pointers to structures in the global stack are examined and pointed structures are copied to Tospace and scavenged to translate the rest of the structure to Tospace. When all of the global stack has been copied Fromspace can be declared free memory. There are several problems with this scheme:

- When the global stack is copied, the stack structure is lost. If a list, for example, has been incrementally created, it could well be the case that it spans several areas that belong to different choice points. The simple strategy of resetting GPOS (the global stack pointer) don't work any more for fast backtracking. All the GPOS-values stored in the local stack with every choice point lose any meaning.

• New structures are created in Newspace in a global stack. Here it is valid to do backtracking in the normal way.

• Several pointers in the trail could be pointers to deleted structures in the copying process. It is necessary then to collect also the garbage in the trail that arises when collecting garbage in the global stack.

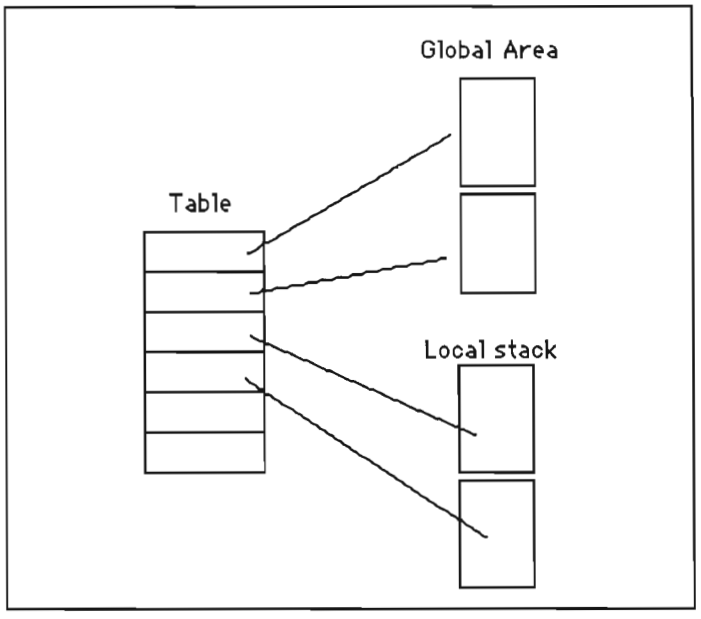
This problems are lost in the following manner:

• After flipping, the global stack loses its structure but the local stack keeps it. It is safe to backtrack resetting the pointer to the local stack. The old GPOS values are not considered valid and are not taken into account. This implies that after flipping, backtracking does not recover automatically the garbage in the global stack. This garbage will be recovered only in the next copying process.
• In Newspace a global stack is constructed. If backtracking does not attempt to reset a global stack pointer to Fromspace, then backtracking can go on as normally.
• It is necessary to compact the trail after having copied all of Fromspace in order to delete dangling pointers and in order to redirect the trail pointers. The stack structure of the trail can be preserved in this operation.

14. A closer look.

There are several features that a high performance Prolog-Workstation should have in common with Lisp-Machines like the Symbolics. One of them is virtual memory and the possibility to discharge part of the garbage collection process in the hardware. The partitioning of the address space that is typical of virtual memory machines simplifies also the task of doing garbage collection.
We could adopt a three-level partitioning of the address space:
• pages are 1K memory chunks, that when accessed reside complete in the main storage. If structures are compacted in a minimal quantity of pages, the access to them can be made more efficient.
• quantums could be big enough sections of memory (64K for example), that are reallocated in memory according to a quantums-table.
• the whole address space is divided in three principal areas: the local-stack memory area, the trail-memory area and the heap. Each area is allocated a memory-quantum at the beginning of the computation and additional quantums are allocated on request. If the virtual address space is big enough,

no conflict should arise. The main storage will be distributed dynamically between the three memory areas, not statically at the beginning of the computation, like in a traditional WAM and so it is not necessary to displace the stacks when one of the memory areas is exhausted.
The transformation between physical and logical addressing of the pages could be handled by a memory management unit.



15. A simpler method

One of the simplest garbage collection methods in Prolog consists in using some kind of anotation to let the programmer specify which structures should be preserved in the forward computation. A fail can be forced then and in this way space in the global stack is recovered.
Consider the following piece of code:

```
goal:- construct(U), qsort(U,S), manipulate(S).
```

In this example a list U is ordered and the result S is used in the rest of the computation. If U is not used anymore, it has become garbage and it should be possible to recover the memory cells occupied by this list. One possible technique to do this would be to transform the goal above into this form:

```
goal:- (create(U), qsort(U,S), asserta(s(S)),fail;
        retract(s(S)), manipulate(S)).
```

We assume in this example that create/1 and qsort/2 are two deterministic predicates. Note that after S has been computed, it is copied from the global stack to the space reserved for code (through the use of assert/1). Thereafter a fail is forced. In this way all the global stack space occupied by parcial results of create/1 and qsort/2 is cleared. S is recovered from the code space and the computation continues with the predicate manipulate/1.
As can be seen this method of garbage collection in Prolog is straightforward and relatively efficient. There are although some inconveniences:

- Care must be taken to ensure that the forced fail does not take us back to a choice point left behind by the predicates whose partial results we want to clear.
- The forced fail should not change the result of the computation. Care must be taken to "encapsulate" it in a suitable manner.
- When a structure is copied to the code space, then aliasing of embedded variables is not necesarily preserved. Consider the code below:

```
a:- X=Y, asserta(eins(X)),asserta(zwei(Y)),
         retract(eins(U)),retract(zwei(V)),... etc.
```

In most Prolog systems after this manipulation, the variables U and V will not represent necesarily the same variable. They will be independent.
- The burden of garbage collection reverts to the programmer. This can be very efficient (because the programmer knows better that anyone else which results should be preserved), but in this case errors can be made and one of the advantages of declarative programming (compared to imperative languages) is lost.

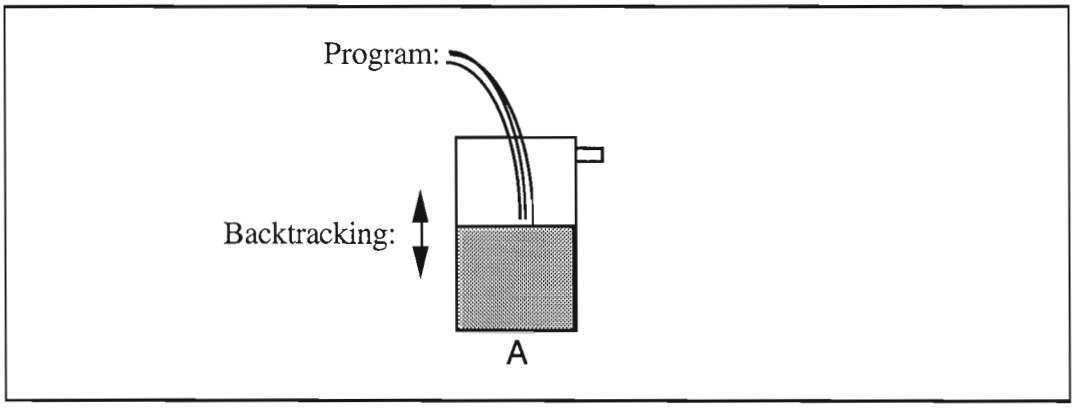
One alternative solution to this last problem is to investigate the possibility of letting the compiler do the program transformation by himself. The compiler could determine (by static analysis or partial interpretation) which structures are no longer needed and set automatically the necessary fails.
A more efficient way to do the copying of structures is also needed. A possible solution would be to define two new built-ins: "preserve/1" and "copyback/1". The argument of the first predicate is a list of terms which should be copied from the global stack to a temporary area. The argument of the second is a list of variables which are bounded to the terms stored in the temporary area. This terms are copied also back to the global stack (of course after this has been pruned).
The code above could be rewritten:

```
goal:- (create(U), qsort(U,S), preserve([S]),fail;
        copyback([S]), manipulate(S)).
```

The advantages of this method are that in the argument list more than one term can be specified, the copy process can be speeded up when the terms are copied to a temporary area (and not to the code area, which must be always manipulated with great care) and after copying back the temporary area must not be compacted again. The next time something is stored, this area can be overwritten. The aliasing of variables could be preserved also.
Note that this method of garbage collection could be considered a very simplified variant of the copying algorithms.

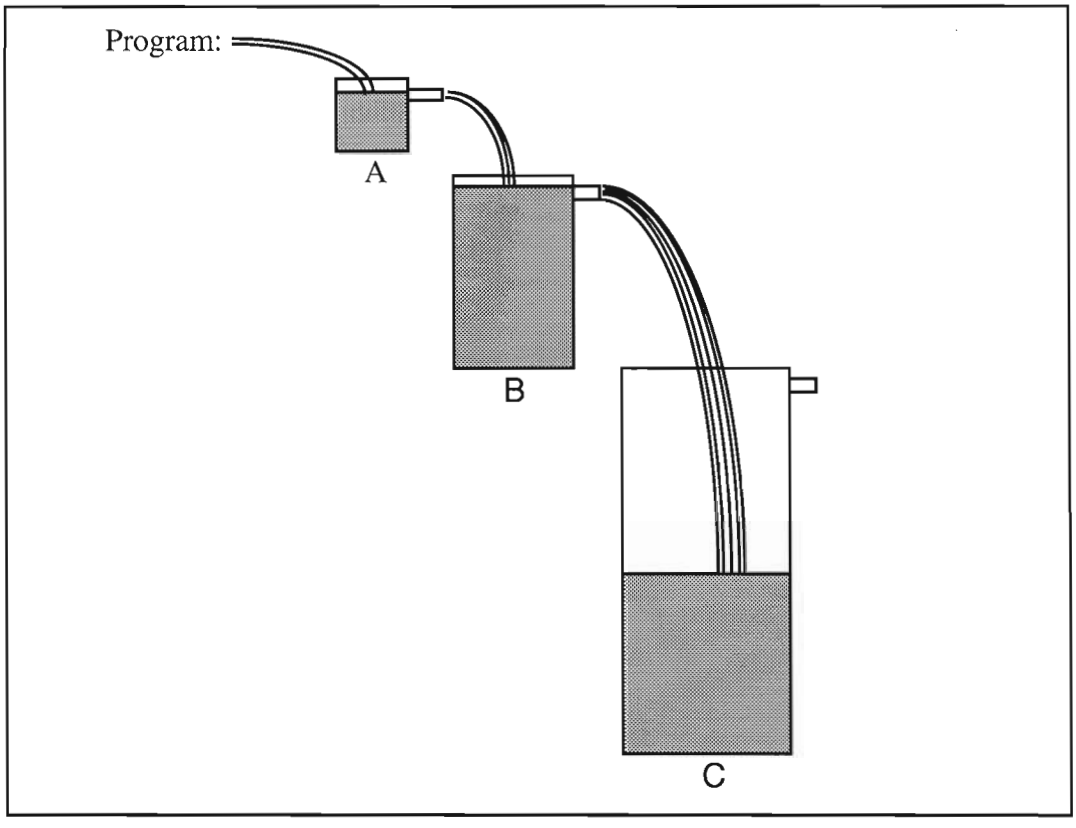
16. A copying method with garbage collection in "stages"

We describe now a method that is a generalization of the Baker algorithmus and that can be used to recover space in the global stack in a very fast way.
The method can be understood more easily by looking at the Figure below. We suppose that a program is pouring structures in the global stack and this will be filled slowly. Backtracking helps to retard the moment in which the global stack will be full. Each time that a fail occurs space in the global space is recovered (i.e. the level of our container descends). Unfortunately backtracking alone can not prevent that the global stack becomes full of used and unused terms. In this moment this section of the global stack "spills over".



Look at the next Figure. When the first section of the global stack spills over, then the following section of the global stack begins to be filled. We would not gain nothing if the spill process transfers only garbage from one section of the stack to the other. But we do not simply transfer terms from the upper to the lower stack section. We copy only those terms that are still alive. Terms are alive if they are pointed by some memory cell in the local

stack or by an alive cell in the global stack. As can be seen, we only apply the Baker algorithm to the stack section A, which is considered as "old space" and we copy the alive terms to stack section B, which is considered "new space". Note that stack section B is bigger than stack section A. That means that we can repeat this copying process several times before stack section B becomes filled. When this happens, we can repeat the copying process, but now we copy terms from B to C.



Ideally we could continue using this method recursively. In practice we have to define a maximal depth of the stack sections. A depth of two or three containers should be efficient for most Prolog programs.
Note the following characteristics of the proposed algorithm:

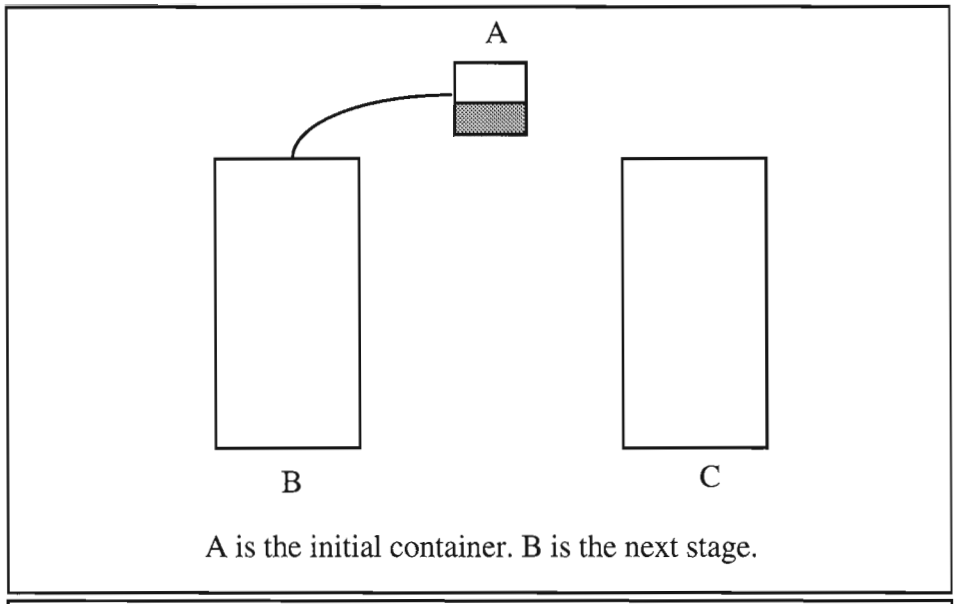
- Any time that terms are copied from an upper to a lower "container" only the last segment of the global stack is being copied. That means that we only have to look at the local stack to find the pointers to the global stack that are of interest as well as to the trail stack to find the pointers in sections of the local stack above the relevant part (more about this later).
- The garbage collection characteristics of backtracking are exploited as in the traditional WAM. All the space that can be recovered through backtracking is recovered and the moment of spill over is delayed.

• This method can be combined with the "preserve/1" and "copyback/1" built-ins proposed above. In this way the spill overs can be further minimized.

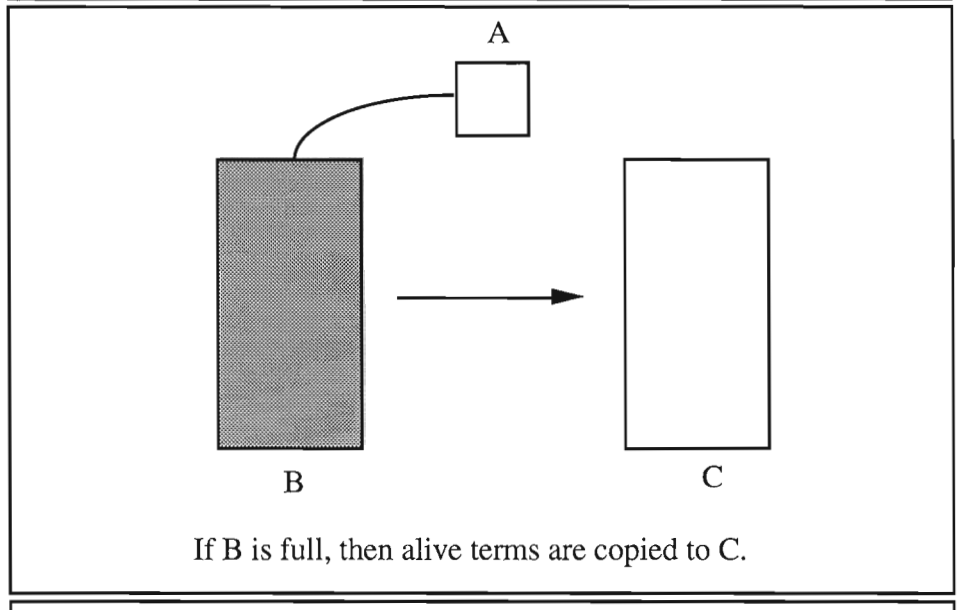
This garbage collection method in "stages" could be considered a method which makes use of generations, like the algorithms used in some Lisp systems [Lieberman/Hewitt 1983]. It is a more general method that the one proposed by Touati [1987]. Each time a term is copied from one stage to the next, the probability that this term could become garbage is further reduced. Each "container" is bigger as the last one, but it contains terms of "higher quality" in the sense, that they represent final or provisional results of the computation being performed.

17. Garbage collection in two stages with a security space

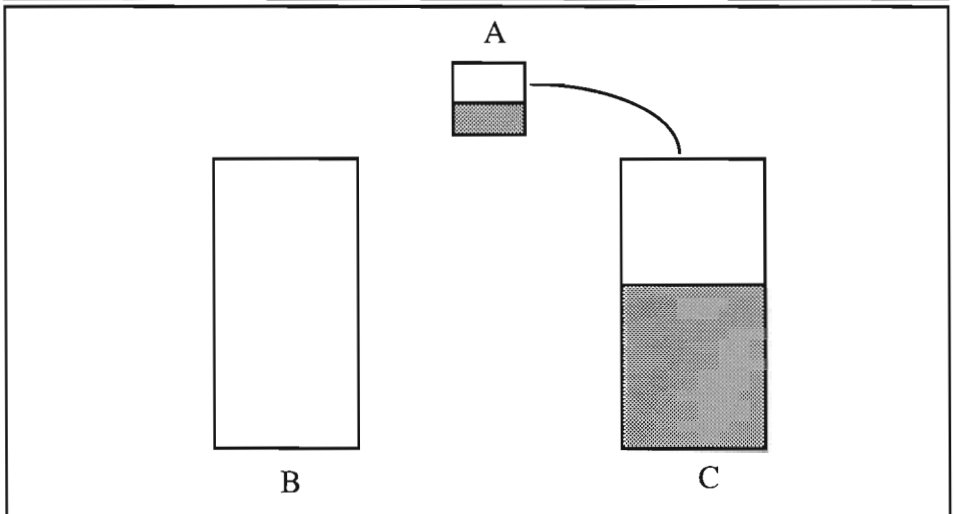
It would be very interesting to investigate empirically how many stages in the above algorithm maximize the memory reclamation characteristics of the copying process. This depends clearly on the characteristics of the programs used. Small benchmarks can not be used for this experiments. Big and complicated programs (like a Prolog compiler) should be examined in order to determine the optimal depth of the chain of stack segments.
We propose for POPE [Beer 1987] a method based in a chain of only two segments. The first segment is used as the initial container, and the second as the "new space" of each copying process. If the second and first segment become full, then the whole global stack is copied using the Baker method to a new area, which is so big as the second segment. In the process some memory will be regained and the process is repeated again with the third section of the stack being defined now as "new space". The Figures below shows this.



A is the initial container. B is the next stage.



If B is full, then alive terms are copied to C.



Some space is recovered, A now works with C

One possible objection to this method is that only one of the big containers B or C is used anytime. The other remains passive waiting for a space flip. To this objection we could answer that this copying method is very fast and it trades memory space for speed in the garbage collection process. If we are interested in getting the most speed from our processor and in reducing the interrupts of the copying process to a minimum, then the extra space is worth the benefits that we get. On the other side, if we do not want to reserve this extra space for the copying method, we could always compress the two used stages in the conventional manner using the Morris algorithm (a description of the Morris technique is given in the Apendix).

18. Conclusions

We have discussed in this paper several problems associated with garbage collection in Prolog systems. We reviewed the techniques used in Lisp systems and concluded that a modified form of the Baker algorithm could be used in a Prolog system. For POPE we propose a copying technique supplemented by two new built-in predicates, which trades space for speed in garbage collection. Future work should assess the possibility of doing garbage colection in real time in the processor pipeline provided by POPE.

References:

Baker, H.G.:List Processing in Real Time on a Serial Computer, Comm. ACM 21, 4 (April 1978) 280-294.

Beer, J.: Concepts, Design, and Performance Analysis of A Parallel Prolog Machine, Dissertation, Berlin 1987.

Bruynooghe, M.: Garbage Collection in Prolog Interpreters, in J.A. Campbell (Ed.), Implementations of Prolog, John Wiley & Sons, New York 1984.

Cheney, C.J.: A nonrecursive list compacting algorithm, Comm. ACM 13, 11 (Nov 1970).

Lieberman, H. and Hewitt, C.: A Real-Time Garbage Collector Based on the Lifetimes of Objects. Comm. ACM 26,6 (June 1983) 419-429.

Moon, David A.: Garbage Collection in a Large Lisp System, ?, ACM.

Morris, F.L.: A Time and Space-Efficient Garbage Compaction Algorithm, Comm. of the ACM, August 1978, Vol. 21, N.8.

Pittomvils, E., Bruynooghe, M., Willems, Y.D.: Towards a Real-Time Garbage Collector for Prolog, in: Proceedings of the 1985 Symposium on Logic Programming, Boston Massachusetts, 1985.

Sterling, L. and Shapiro, E.: The Art of Prolog, MIT Press, Cambridge Mass. 1986.

Symbolics Inc.: Internals, Processes and Storage Management (Symbolics 36xx), USA 1986.

Touati, Herve: A Prolog Garbage Collector for Aquarius, Report No. UCB/CSD 88/443, August 1988, Berkeley, California.

Warren, D.H.: An Abstract Prolog Instruction Set, Technical Note 309, October 1983, SRI International.

Warren, D.H.D.: Optimizing Tail Recursion in Prolog, in Michel van Caneghem, D. Warren (Eds), Logic Programming and its Applications, Ablex Publishing Corporation, New Jersey 1986.