A Generational Parallel Copying Garbage Collector
for Shared Memory Prolog

Johan Bevemyr
Computing Science Dept., Uppsala University
e-mail: Johan.Bevemyr@csd.uu.se
Box 311, S-751 05 Uppsala, Sweden
Phone: +48–18–182500
Fax: +46–18–51 19 25

Abstract
We show how to combine a sequential copying garbage collector for Prolog with a scheme for parallel shared memory copying collection, and how to extend these schemes for generational garbage collection. We also present a new approach to dealing with troublesome primitives. The new garbage collector shows reasonable speedup for a number of benchmarks.

1 Introduction
Our experience is that Prolog programs spend on average 15% of their execution time in garbage collection. Some programs garbage collect as much as 70% of their execution time.

To efficiently execute such programs in parallel it is essential that the garbage collector is parallelised as efficiently as the rest of the execution. Otherwise the garbage collector will become a sequential bottleneck. If the collection time is 15% of the sequential execution time, then that limits the speedup to \(1/0.15 = 6.67\).

It has been shown how a generational copying garbage collector can be used for sequential Prolog [6]. It has also been shown how to parallelise copying garbage collection for shared memory multiprocessors [1]. We present a collection scheme that combines these techniques resulting in a parallel generational copying garbage collector for Prolog.

Let us consider the architecture of a typical WAM [14]: most data are stored on a global stack (also called the heap), while choice points and environments are stored on a local stack (also referred to as the stack). A trail stack records bindings to be undone on backtracking. We will not consider garbage collection of code space in this paper, atom tables or other miscellaneous areas. There are no pointers from such tables into the garbage collected areas.

The WAM saves the state of the machine whenever a choice point is created. Using this information, stacks can be reset and storage reclaimed cheaply. We can
view the global stack as composed of several segments, delimited by the choice point stack. Creating a new choice point creates a new segment; backtracking removes segments, while performing a cut merges segments. Data are allocated in the topmost segment. Variable bindings are implemented as assigning a cell representing the variable. Bindings are recorded on the trail stack whenever the variable cell is not in the topmost segment. When two variables are unified, a pointer from one cell to the other cell is created. In general, pointer chains may arise which require dereferencing.

2 Sequential Copying Garbage Collection

Traditionally, Prolog garbage collectors are based on a mark-slide algorithm [2]. The reason is that Prolog implementations tend to rely on that live data maintains their relative positions throughout the execution. This is important for the following reasons:

1. The location of a variable is used for deciding if trailing is required when binding the variable.

2. Data allocated on the heap can be instantly reclaimed on backtracking.

3. Variables are ordered by their relative position in the heap, for example, when generic comparison operators such as \( @ \) are used.

Mark-slide collectors have the disadvantage that the collection time is proportional to the size of the memory area, where copying collectors are proportional to the size of the live data. It is desirable to use the faster copying collectors in many situations.

An improved method for dealing with the problem of comparison operators is presented below.

2.1 Instant Reclaiming

A compacting collector preserves the heap segments (see Figure 1) and entire segments can be deallocated on backtracking.

Bekkers, Ridioux and Ungaro[4] suggested that a reasonable approximation of the heap segments can be preserved across garbage collections. This is achieved by copying the data in a carefully chosen order, i.e., starting at the oldest choice point. This is not possible in a parallel setting without strict synchronisation, as explained below.

Bevemyr and Lindgren[6] indicated that instant reclaiming of data that have survived a garbage collection can be sacrificed without loss of efficiency, at least for a range of benchmarks. Using their scheme, instant reclaiming of data is still possible for data allocated between collections. All heap segments are merged into one during garbage collection (see Figure 2).

2.2 Trailing

The variables’ positions relative to the heap segments are used for deciding if a variable binding needs to be trailed or not. Bevemyr and Lindgren [6] solve this by trailing all bindings of variables that have survived a garbage collection, i.e., all variables residing in the collapsed segments.
Before garbage collection

After garbage collection

Figure 1: Saved heap top pointers (H) in choice points before and after compacting garbage collection. Heap segments are preserved.

2.3 Bevemyr-Lindgren’s Mark-Copy Algorithm

The copying collector is a straightforward adaption of Cheney’s algorithm [9] and works in three phases. The algorithm allows the standard optimisations of early reset. The global stack is divided into two areas. The old data reside in from-space and are evacuated into to-space.

A slight complication appears in Prolog-implementation since they tend to allow external references into structures, e.g., to variables inside structures. These externally referenced cells can be reached and copied before the surrounding structure, resulting in duplicated memory for a single variable [6] (see Figure 3 and 4). This is undesirable since the result of doing garbage collection might then be that more space is required! Furthermore, the length of reference chains are no longer predictable.

A solution to this problem is to mark all internal cells before copying. When a marked cell is encountered during copying all surrounding cells are copied. Marking the internal cells is done using the Deutch-Schorr-Waite[12] pointer reversal algorithm.

Algorithm:

1. Mark the live data. When a structure is encountered, mark the functor cell and all internal cells. When a simple object is found, mark that cell only.

2. Copy the data using Cheney’s breadth-first algorithm. When a marked cell is visited in from-space, do the following:

   (a) Scan backward (towards lower addresses) until an unmarked cell is found.
   (b) Scan forward and evacuate marked cells into to-space until an unmarked cell is found. Overwrite the old cells with forwarding pointers to the corresponding cells in the copy.

Thus, interior pointers are handled correctly. Several adjacent live objects may be evacuated at once. Continue until no cells remain to be evacuated.
Figure 2: Saved heap top pointers (H) in choice points before and after **copying** garbage collection. After garbage collection all segments have been merged into one single segment. Note that only the active heap is shown here—the live data have been copied from the old heap to the new heap.

Figure 3: An internal cell is referenced twice, both directly by a variable and indirectly through a structure.

3. Update the trail. If a trail entry does not refer to a copied cell (i.e., does not point at a forwarding pointer), it can be deleted. Implementing early reset is done by incorporating this step into the procedure that copies live data from the chain of choice points.

### 2.4 Generational Collection

Generational garbage collection [11, 3] relies on the observation that newly created objects tend to be short-lived. Thus, garbage collection should concentrate on recently created data. The heap is split into two or more generations, and the most recent generation is collected most frequently. When the youngest generation fills up, a collection spanning more generations is done, and the survivors move to the oldest of these generations. Frequently, implementations have two generations, and we will assume so from now on.

Roots include the pointers from the older to the younger generation. In languages...
Figure 4: As a result of copying without marking internal cells some cells might be duplicated.

Figure 5: The limit for trailing is set in such a way that all cross-generational references are recorded on the trail. In the presence of a choice point in the new generation the trail limit is set as usual.

such as SML, most objects are immutable, and assignments that cause cross generational pointers can be compiled into special code. In Prolog, there is a high incidence of assigning already created objects, so such a solution is likely to be expensive. Bevemyr-Lindgren show how it can be arrange that only trailed bindings may be cross-generational by setting the limit where trailing occurs appropriately, see Figure 5.

In other languages it is usually necessary to add a write barrier, code that detects cross generational bindings and record them on a stack. This results in a runtime cost for using generational garbage collection. In WAM based Prologs this overhead is already present in the form of trail tests and there is no extra runtime penalty for using generational collection.

3 Parallel Garbage Collection
Khayri Ali [1] has proposed an elegant scheme for parallelising copying garbage collection on shared memory multiprocessors. His method assumes that memory is divided into segments where all segments are readable by all processing elements
Figure 6: Memory is divided into segments and stored into a common pool. The processing elements allocate segments from the pool depending on their individual requirements.

(PEs). Initially all segments are stored in a common pool. The PEs allocate segments from the pool and link them into their private memory area (see Figure 6). References between the different memory areas are allowed.

Garbage collection is initiated when the number of free segments decreases below a given limit, say half the available segments. Each PE copies its live data into new segments allocated from the free pool. The old segments are returned to the free segment pool when all live data have been copied.

Two things have to be considered:

1. How to copy the data in such a way that objects shared between PEs are only copied once.
2. How to facilitate load balancing between PEs. If a PE can reach only a small amount of live data, it should be able to help other PEs copy their data.

Ali solves these problems in the following way.

- By locking each substructure before copying it can be guaranteed that each object is only copied once.

- By traversing the data in such a way that idle PEs can steal work from working PEs. This is made possible by using a pointer reversal algorithm and allowing splitting of the pointer-reversal chain.

3.1 Ali's Algorithm

3.1.1 Copying The data is traversed depth first using pointer reversal. Two chains are constructed, one pointing backward into the pointer-reversal chain of partially copied data (P), and one pointing forward to yet uncopied data (L). One mark bit is used for two purposes. In from-space the mark bit is used for marking already copied data (forward pointers). In to-space the mark bit is used for marking the last cell to be scanned in an already copied structure.

Copying a structure proceed as follows:
The object immediately accessible from the root is copied into to-space (if not already copied, in which case the root pointer is simply updated). Forward pointers are installed in from-space and the first cell in the copied object is marked to indicate that it is the last cell to be scanned in the new object.

The first $L$ is set to point to the first cell to be scanned, i.e., the last cell in the copied object. Initially, $P$ points to a dummy cell. $L$ may:

1. Point to a marked cell indicating that it is the last to be scanned in the object.
   Here we have two possible situations. The cell may point to:
   
   (a) An uncopied object. In this case we proceed as in case 2 (a) below, except that the mark is preserved then the cell is linked into the pointer-reversed chain.

   (b) A copied object. The pointer is updated according to the forward pointer in from-space. Now the object is completely scanned.

   We have three possibilities. $P$ may point to
   
   i. The dummy cell. In this case the scanning is complete.
   ii. An unmarked cell. The chain is reversed and $P$ and $L$ are updated accordingly.
   iii. A marked cell. Both $P$ and $L$ point to completely scanned objects.

   The $P$ chain is followed until either the dummy cell is encountered, or an unmarked cell. In the former case scanning terminates, in the latter $P$ and $L$ are updated and the scanning proceeds.

2. Point to an unmarked cell.
   The cell pointed to by $L$ may point to

   (a) An uncopied object. In this case the object is copied to to-space and forward pointers written in from-space. The first cell in the copied object is marked to indicate that it is the last cell to be scanned in the new object.

   The cell in the scanned object (pointed to by $L$) is linked into the $P$ chain and $L$ is updated accordingly.

   (b) A copied object. The cell is updated and $L$ is advanced to the next cell to be scanned in the current object.

Copying an object from from-space to to-space is done as follows to ensure that only one PE copies the object.

1. The first cell in the from-space object is locked by atomically exchanging its contents for a special lock-value. It is possible that the object has already been copied by another PE, at this point. In that case it should be assumed that a copied object was found in the first place.

2. All cells in the object are copied to to-space and forward pointers are installed in from-space for all cells, except the one containing the lock value.

3. The lock is released by replacing the lock-value with a forward pointer.

Snapshots of the algorithm can be seen in Figures 7, 8, 9, and 10.
Figure 7: Snapshots of Ali’s collector in action. The first accessible object, from the root, is copied into to-space. The last cell in the copy is marked and forward pointers installed in from-space.

3.1.2 Load Distribution The idea is to divide the backward chain \((P)\) into shorter chains and make them available to other PEs. This is done by saving pointers to the backward chain on a work-stack at regular intervals. An idle PE selects a pointer from the work-stack of another PE and splits the chain in two (see Figure 11). The PEs then proceed to work on their respective chains.

This means that different parts of a tree can be copied in parallel by a set of PEs.

4 Parallel Prolog Garbage Collection

We assume that the parallel Prolog implementation consists of a number of workers. Each worker is a full WAM with all associated memory areas. All workers have shared access to each others heaps with the restriction that they can only create new objects on their own heap. Reform Prolog [7, 8] is an example of this kind of implementation.

4.1 Segmented Memory

To use Ali’s scheme the heaps have to be divided into segments. This is desirable for other reasons as well. Memory management becomes more flexible: workers may use non-uniform amounts of memory and the heap can be extend simply by requesting new memory from the operating system and linking it into the pool of free segments.

Segmenting the heap requires some modification to a WAM implementation.

1. The stack cannot be guaranteed to be allocated below the heap. Heap and stack variables must be distinguished in some other way, e.g., by giving them separate tags.

2. Checking for memory overflow must be done differently. If the end of a segment is reached, a new segment is allocated from the free pool. If the
Figure 8: Foo/1 is copied into to-space and its last element is marked. A backward pointer chain is created. The cons cell in foo/1 is copied into to-space, marking its last element. The backward chain is extended with the list pointer in foo/1.

- Figure 8: Foo/1 is copied into to-space and its last element is marked. A backward pointer chain is created. The cons cell in foo/1 is copied into to-space, marking its last element. The backward chain is extended with the list pointer in foo/1.

free-segment limit has been reached garbage collection is to be initiated.
A special trail entry can be used for noting that a new segment has been allocated. The segment can then be deallocating on backtracking.

3. Heap segments are no longer ordered by their position in memory. Deciding if a variable binding must be trailed or not cannot be done by comparing the variables location on the heap to the current segment pointer. This problem occurs as soon as several heaps are used. One solution is to associate a choice point identifier with each variable. Different methods for this have been investigated by Bezemyr [5].

4.2 Modifications to the Sequential Algorithm
The sequential algorithm has to be modified in the following ways to run in parallel:

1. The algorithm for marking live data cannot use pointer reversal. The reason is that several workers may mark the same structures with the possibility of destroying each others’ pointer chains.

A simple solution is to use a recursive algorithm instead. The empty segments in to-space are used for keeping the stacks. This area is guaranteed to be sufficiently large if tail recursive optimisation, described below, is applied. Let us consider the worst case: a structure of nested lists occupying the entire memory. Each cons cell occupies two words. Each pushed entry also occupy two words. The stack would, at its worst, occupy the entire to-space.

Marking a live cell is done by atomically setting the mark bit.

2. A potential problem is that copying large chunks of surrounding live data may result in bad segment utilisation. The solution is to mark live data both as live-data and as internal-cells. The internal-cell mark is only set for structure
arguments, not the functor word or the first word in a cons-cell. It is now possible to copy only the structure surrounding an internal cell, i.e., the cells marked with the internal-cell mark and the first non-marked (see Figure 12). This will effectively limit the amount of data that can be copied as one block (apart from very large structures).

Cons cells present a problem in that the first cell can be referenced both as a variable and as a list (see Figure 12). The solution is to always check if the target cell is part of a cons and in that case copy the entire cons. This slows down copying of variable cells. However, the overhead is minimal when the mark bits are stored separately from the data.

3. Barrier synchronisation has to be used in three places:

   (a) Before garbage collection starts. This is to ensure that all workers have entered the garbage collector.

   (b) After the marking phase. All marking has to terminate before the data can be migrated.

   (c) After the copying phase to ensure that all data have been copied.

   (d) Before deallocating the segments in from-space to ensure that all workers have updated their trail pointers.

4. The early reset and variable shunting optimisations cannot be used. They rely on heap segments to be either marked or copied in order. This would require barrier synchronisation after marking and copying each choice point, which would almost sequentialise the collector.

For the same reason Bekkers et al.’s scheme for instant reclaiming cannot be used.
Two mark bits are used with different interpretations in from-space and to-space. In from-space the bits are used for marking forward-pointers (F [forward]) and for marking live arguments (A [arguments]). In to-space they are used for marking the last cell to be scanned (H [head]) in an object, and for marking the cell that the object was entered through if not entered from the top (I [internal]). The latter is required when surrounding objects are copied together with an internal cell. In that case the first cell is marked with both H and I and the internal cell with I.

High level view of the algorithm:

1. Mark the live data in parallel using a recursive algorithm. The segments in to-space are used for keeping the stacks. All structure cells, except the first in each structure, are marked with the internal-cell bit.

2. Copy the live data in parallel using a modified version of Ali’s pointer reversal scheme.

3. Update the trail, removing segment deallocation entries.

4. Deallocate from-space.

4.3.1 Algorithm for marking live data  

C points to the current cell and R contains the number of unmarked cells in the current object. Initially C points to the root.

Iterate the following until marking terminates with R equal to zero and an empty stack.

C may:

1. Point to a marked cell (D). We have two possibilities:

   (a) R is zero. We have two possible situations:
Figure 11: Dividing a chain into two small chains to be scanned in parallel.

i. The stack is empty and marking terminates.
ii. The stack is non-empty. $R$ and $C$ are popped from the stack. If the popped $R$ is zero, continue to pop $R$ and $C$ until $R$ is non-zero or the stack is empty.

(b) $R$ is greater than zero. Decrement $R$ and increment $C$ to the next object argument.

2. Point to an unmarked cell. Mark the cell (D).

We have three possible situations. The cell is:

(a) A pointer to a structure. Push the current $C$ and $R$ on the stack. Set $C$ to point to the new object and $R$ to the arity of the new object. Mark all structure cells, except the first, as internal (A).

(b) A pointer to a variable. Push the current $C$ and $R$ on the stack. Set $C$ to point to the new variable and $R$ to zero.
(c) An immediate value, e.g., an integer or atom. Proceed as 1.

The algorithm can be tail recursively optimised by not pushing \( R \) and \( C \) if \( R \) is zero. The algorithm can be parallelised by letting idle workers pop entries from the stack, or simply start marking data from the pointers found in the stack. Already marked data is ignored by the algorithm.

### 4.3.2 Algorithm for copying data

The object immediately accessible from the root is copied into to-space (if not already copied, in which case the root pointer is simply updated). Forward pointers are installed in from-space and the first cell in the copied object is marked to indicate that it is the last cell to be scanned in the new object.

Initially, \( L \) is set to point to the first cell to be scanned, i.e., the last cell in the copied object. \( P \) points to a dummy cell. All live arguments in from-space have initially been marked with \( A \). \( L \) may:

1. Point to a \( H \)-marked cell indicating that it is the last to be scanned in the object.

   Here we have two possible situations. The cell may point to:

   (a) An uncopied object. In this case we proceed as in case 2 (a) below, except that the mark is preserved when the cell is linked into the pointer-reversed chain.

   (b) A copied object. The pointer is updated according to the forward pointer in from-space. Now the object is completely scanned.

   We have two possibilities:

   i. The cell is also marked with \( I \). In this case the object was entered through an internal cell. The current cell is \( I \)- and \( H \)-unmarked and \( L \) is incremented through the object until an \( I \)-marked cell is found. The \( I \)-mark is removed.

   ii. The cell is not marked with \( I \). In this case the object was entered from the top and \( L \) does not have to be modified.

We now have three possibilities. \( P \) may point to
i. The dummy cell. In this case the scanning is complete.

ii. A H-unmarked cell. The chain is reversed and \( P \) and \( L \) are updated accordingly.

iii. A H-marked cell. Both \( P \) and \( L \) point to completely scanned objects. 
The \( P \) chain is followed until either the dummy cell is encountered, 
or a H-unmarked cell. In the former case scanning terminates, in the 
latter \( P \) and \( L \) are updated and the scanning proceeds.

2. Point to a non H-marked cell.

   The cell pointed to by \( L \) may point to

   (a) An uncopied object. In this case the object is copied to to-space and 
   forward pointers written in from-space. The first cell in the copied object 
is H-marked to indicate that it is the last cell to be scanned in the new 
object.
   The cell in the scanned object (pointed to by \( L \)) is linked into the \( P \) 
chain and \( L \) is updated accordingly.
   (b) An uncopied internal cell. In this case the surrounding object is copied 
to to-space and forward pointers written in from-space. The first cell 
in the copied object is H-marked to indicate that it is the last cell to be 
scanned in the new object. It is also I-marked to indicate that the 
object was entered through an internal cell.
   The copy of the internal cell is I-marked to mark the entry point.
   The cell in the scanned object (pointed to by \( L \)) is linked into the \( P \) 
chain and \( L \) is updated accordingly.
   (c) A copied object. The cell is updated and \( L \) is advanced to the next cell 
to be scanned in the current object.

A snapshot of the collector is seen in Figure 13. Objects are copied and locked as 
before.

4.4 Optimisations

   The forward-pointer bit and the live-data bit can be merged into one bit. The bit 
is set by the marking algorithm and unset when a cell is forwarded, i.e., a forward 
pointer is recognised by not having the live-data bit set.

5 Generational Parallel Garbage Collection

   In a generational memory hierarchy objects are created in one generation; the new 
generation. When a generation is garbage collected all live data are moved to an 
older generation than the present one. Older generations are garbage collected less 
often. This scheme is motivated by an observation that newly created objects tend 
to be short-lived.

   We propose a scheme using two generations; new and old. Data are created in the 
new generation. Filling the oldest generation triggers a normal garbage collection.

5.1 Finding the Live Data

   The roots to the data in new generation are found in registers, choice points and trail 
entries created after the last gc, environments, and in the old generation (we call 
pointers from old to new cross generational). References to the different generations
may appear in all these places. It is essential that the collector can differentiate between objects in different generations, to avoid marking and copying data in other generations.

The segmented memory is divided by a *generation limit* into a new and an old generation. A simple pointer comparison with the generation limit can be used to distinguish elements in the new and the old generation (see Figure 14).

The size of generations can be changed by moving segments between them, and adjusting the generation limit accordingly. A new segment requested from the operating system can be sorted into the proper generation depending on its location in memory.

### 5.1.1 Limiting the Search

It is too expensive to exhaustively search all areas for pointers into the new generation. The benefit of not having to copy long lived data can be severely reduced by repeatedly searching for roots in areas where none exist.

The *old generation*. The traditional solution is to use a *write barrier* to detect cross generational pointers as they are created, and record them for use in the gc. The write barrier is implemented by adding detection code for each potential cross generational assignment. Bevenyr and Lindgren [6] observe that this code is already present in Prolog in the form of trail tests. All cross generational bindings can be automatically recorded by carefully setting the trail condition. This results in some extra trailing.
The trail. Only trail entries added since the last gc have to be examined. A pointer to the top of the trail is saved after garbage collection. This pointer is updated when the trail is unwound to a point below the saved point, adding a slight overhead in the code for backtracking.

Choice points. Only new choice points, i.e., choice points created after the last gc, have to be searched. Old choice points can be detected by marking them during garbage collection. A spare bit in the trail top pointer can be used for this purpose (this is what the mark-sweep collector in SICStus Prolog [2] does).

Environments. Environments can be divided into three categories.

1. New environments, i.e., environments created after the last collection.
2. Old unprotected environments, i.e., environments created before the last collection which are not protected by a surviving choice point.
3. Old protected environments, i.e., old environments protected by an old choice point.

Environments in the first and second category have to be searched. References from the third category are found when examining the trail. Having to search all old unprotected environments is potentially a problem since efficient programs (largely deterministic) have few choice points.
A remedy is to force trailing of all bindings in all old environments, even those not protected by choice points. This is done by introducing a trail limit register, similar to the one used for the heap. This register is set to the maximum of the youngest old environment and the current choice point. A separate register is used to keep track of the youngest old environment. This register must be updated when environments are deallocated.

We have implemented the first solution, i.e., “search all old unprotected environments”.

6 Ordering Variables

In most Prolog implementations variables are ordered by their relative position on the heap. This is not possible in our scheme since copying collectors do not preserve the relative position of the data. Also, using segmented memory makes this problematic. We suggest that ordering numbers are unconditionally associated with variables when they are compared. The ordering numbers are stored in a new area, the static heap, which is not reclaimed on backtracking. The following scheme can be used:

Comparing two variables:

1. If both are already ordered, compare their ordering number. A variable may be ordered if it has been compared previously.
2. If one is ordered, denote that one as older.
3. If none is ordered, pick one and associate an ordering number with that one. The now-ordered variable is older than the unordered.

The unification algorithm has to be modified accordingly. An unordered variable should be bound to an ordered and of two ordered variables the younger should be bound to the older.

Ordering a variable \( X \) is done in the following way:

1. Create an ordered-variable structure on the static heap.

\[
\begin{array}{c|c}
\text{ATM} & \text{ordering number} \\
\hline
\text{HVA} & \text{unbound variable} \\
\end{array}
\]

The structure is formed by an ordering number (a special atom) followed by an unbound variable.

2. \( X \) is unconditionally (without trailing) bound to the new variable. This is possible since the ordering structures are created in an area which is not recovered on backtracking.

The garbage collector must detect when an ordered variable is copied and in that case also copy the ordering number. This incurs a slight overhead when copying variables. However, if the data that survive a garbage collection is not recoverable on backtracking it is safe to copy both unordered and ordered variables into the same memory area. This is the case in Beemyr-Lindgren’s algorithm as well as in the parallel algorithm proposed in this paper. If one wishes to use a scheme where
data can be reclaimed on backtracking after GC, then all ordered variables have to be copied to a new static heap. This would not impose any extra overhead.

This scheme has the advantage that variables that have been compared will stay ordered in different OR-branches. Programs that, for example, use unbound variables as keys will works as expected, i.e.

\[
\text{insert}(\text{Key}, t(\text{Left}, \text{Element}, \text{Right})) :\]
\[
\text{Key} @< \text{Element},
\]
\[
\text{...}
\]

\[
\text{insert}(\text{Key}, t(\text{Left}, \text{Element}, \text{Right})) :\]
\[
\text{Element} @< \text{Key},
\]
\[
\text{...}
\]

This would not be the case if variables were not guaranteed to be consistently ordered in different OR-branches.

The disadvantage is that a slight overhead is imposed when unifying unbound variables and when garbage collecting heap variables. However, our measurements indicate that the run-time overheads added are to small to be measured, at least in an emulator based WAM implementation. We measured both programs containing a moderate amount of ordered variables and programs without.

7 Performance

Preliminary results indicate speedup in the order of 5-6 on 10 processors, compared to garbage collection in our sequential Prolog. This measured on a Sparc-Center 2000 (bus based shared memory multiprocessor). For these programs the speedup limit, due to garbage collection, is increased from 6.7 to 37 (total execution time/garbage collection time).

Preliminary results also indicate that the generational version may practical as it is. The overheads for synchronising more often, and the added search for roots to new generation, appear to be to large. However, streamlining the implementation may change this.

The load balancing algorithm does not work satisfactory for long lists of ground elements. The reason is that the main work lies in the depth first traversal, and that part is not parallelised.

In Reform Prolog\cite{7, 8} long list are used for storing arguments to the parallel processes. The cons-cells of these lists are stored in consecutive memory cells which make it possible to treat these lists as vectors. It should be possible to exploit this structure for more efficient parallelisation of the garbage collector. CDR-coding of lists could be used to achieve the same result in a more general setting.

The performance of the collector will be further investigated.

8 Conclusion

We have shown how a copying garbage collector for Prolog can be combined with a scheme for parallel copying collection. We have also shown how the resulting collector can be made generational. However, it is not yet clear if the generational collector is more efficient than the non-generational. The overheads from synchronising more often, and from having to repeatedly search for roots, may outweigh what is gained not having to copy long lived data.

Preliminary results indicate that speedup in the range of 5-6 can be achieved on 10 processors.
Acknowledgements. I would like to thank Khayri Ali for helping through the trickier parts of his garbage collection scheme. Thanks also to Thomas Lindgren and Per Mikkelsen for valuable discussions on the presentation and the content.

References


