UPPSALA THESES IN COMPUTING SCIENCE
25

# Data-parallel Implementation of Prolog

Johan Bevemyr

Computing Science Department
Uppsala University

Thesis for the Degree of
Doctor of Philosophy

UPPSALA 1996

# Data-parallel Implementation of Prolog

Johan Bevemyr

A Dissertation submitted in partial fulfilment of the requirements for the Degree of Doctor of Philosophy at Computing Science Department, Uppsala University.

Uppsala University
Computing Science Department
Box 311, S-751 05 Uppsala, Sweden

Uppsala Theses in Computing Science 25
ISSN 0283–359X
ISBN 91–506–1177–1

(Dissertation for the Degree of Doctor of Philosophy in Computing Science presented at Uppsala University in 1996)

## Abstract

Bevemyr, J. 1996: Data-parallel Implementation of Prolog, *Uppsala Theses in Computing Science 25.* 38 pp. Uppsala. ISSN 0283–359X, ISBN 91–506–1177–1.

Parallel computers are rarely used to make individual programs execute faster, they are mostly used to increase throughput by running separate programs in parallel. The reason is that parallel computers are hard to program, and automatic parallelisation have proven difficult. The commercially most successful technique is loop parallelisation in Fortran. Its success builds on an elegant model for parallel execution, data-parallelism, combined with some restrictions which guarantee high performance.

We have investigated how Prolog, a symbolic language based on predicate calculus, can be parallelised using the same principles: a simple model for parallel execution suitably restricted to guarantee efficient parallel execution.

Two models for expressing the parallelism have been investigated: parallel recursion, based on Millroth's work on Reform compilation, and bounded quantifications. We propose adding bounded quantifications to Prolog and show how both bounded quantifications and recursion parallelism can be implemented using the same implementation technique. The implementation is based on a conventional Prolog engine, Warren's Abstract Machine. The engine has been extended to handle parallel execution on a shared memory architecture.

The implementation shows promising speed-ups, in the range of 19–23.2 on 24 processors, on a set of benchmarks. The parallelisation overheads are between 2–15% and the parallel efficiency varies between 79–97% on the same set of benchmarks.

*Johan Bevemyr, Computing Science Department, Uppsala University, Box 311, S-751 05 Uppsala, Sweden.*

ISSN 0283–359X
ISBN 91–506–1177–1
Printed by Graphic Systems AB, Stockholm 1996

*To Katrin and Lisa*

# Acknowledgments

I am grateful to all who have supported me and my research during these years. In particular I would like to thank the following people:

Håkan Millroth, my supervisor. Not only did he initiate the Reform research group and provide us with insightful guidance (bringing us back on track at numerous occasions), he has also created a very enjoyable and inspiring research atmosphere.

My friend and colleague Thomas Lindgren for persuading Håkan to bring me onto the Reform project in the first place. We have since then been working together, closely at times. Many of the ideas in this thesis have arisen from discussions with Thomas, who is always full of novel ideas, and never late to understand what you are trying to say. Not to mention a portable Alta Vista and a never ending source of interesting papers.

Jonas Barklund for his collaboration on bounded quantifiers and his frequent comments on my writing. Jonas introduced me to the intriguing fields of parallel programming and abstract machine design and implementation.

There is also the daily coffee crowd where the most frequent participants have been Per Mildner, Sven-Olof Nyström, Björn Carlson, Magnus Nordin, and lately Greger Ottosson, Alexander Bottema, Joel Fredrikson, and Anders Lindgren. Their relentless probing and doubting questions (especially Per's) have often forced me to re-evaluate old ideas and frequently inspired to new.

Patric Hedlin for his help in removing a large number of programming errors and for writing all those libraries and built-in predicates you need to create a working Prolog.

Marianne Ahrne for correcting my numerous errors in handling the English language (after which I have readily introduced some more).

Mats Carlsson and Kish Shen for their comments on the draft of my thesis.

I would also like to thank Roland Karlsson for helping me master the shared memory routines on the Sequent Symmetry, and Kayri Ali for clarifying the trickier parts of his garbage collection algorithm.

Finally, Katrin Boberg Bevemyr for commenting on my papers but most of all for all her love and care and for being there as my friend.

This thesis consists of the following papers, which will be referred to in the text by their Roman numerals:

I. Johan Bevemyr, Thomas Lindgren and Håkan Millroth, Reform Prolog: The Language and its Implementation, In *Proc. 10th Int. Conf. Logic programming*, MIT Press, 1993.

II. Johan Bevemyr, Thomas Lindgren and Håkan Millroth, Exploiting Recursion-Parallelism in Prolog, In *Proc. PARLE'93*, Springer LNCS 694, 1993.

III. Thomas Lindgren, Johan Bevemyr and Håkan Millroth, Compiler optimizations in Reform Prolog: experiments on the KSR-1 multiprocessor, In *Proc. EURO-PAR'95 Parallel Processing*, Springer LNCS 966, 1995.

IV. Johan Bevemyr, A Scheme for Executing Nested Recursion Parallelism, In *Workshop on Parallelism and Implementation Technology for (Constraint) Logic Programming Languages*, 1996.

V. Jonas Barklund, Johan Bevemyr, Executing Bounded Quantifications on Shared Memory Multiprocessors, In *Proc. PLILP'93*, Springer LNCS 714, 1993.

VI. Jonas Barklund, Johan Bevemyr, Prolog with Arrays and Bounded Quantifications, *Journal of Programming Languages.* (In press)

VII. Johan Bevemyr, Thomas Lindgren, A Simple and Efficient Copying Garbage Collector for Prolog, In *Proc. PLILP'94*, Springer LNCS 844, 1994.

VIII. Johan Bevemyr, A Parallel Generational Copying Garbage Collector for Shared Memory Prolog, A shorter version appeared in *ILPS Workshop on Parallel Logic Programming*, 1995.

The papers have been reproduced with permission from the publishers.

# Contents

# Summary

This chapter summarises the papers in this thesis and discusses their scientific contributions, and my personal contributions.

The thesis deals with AND-parallel implementation of Prolog in the form of data-parallelism. My contributions range from language design to design of the runtime system.

Papers I, II, and III describe Reform Prolog, a system where data-parallelism is exploited in the form of structure-recursion. Paper IV shows how Reform Prolog can be extended for nested parallel execution.

Papers V and VI describe a different approach for expressing data-parallelism in Prolog. Prolog is extended with bounded quantifiers as a high level construct for expressing iteration over a set of values.

Both these approaches require a scalable runtime system. The most expensive activity in the runtime system is garbage collection. Paper VII lays the foundation for the parallel algorithm by describing a sequential generational copying collector for Prolog. The algorithm is extended for parallel operation in paper VIII.

## 1.1 INTRODUCTION

The number of parallel computers on the market has increased rapidly the last few years: all major computer manufacturers are selling parallel computers. Often the potential parallelism is used to increase the throughput by executing distinct programs in parallel. It is rarely used to make individual programs execute faster.

There are several reasons why these computers are used almost exclusively in this way.

1. Existing compilers are not able to utilise inherent parallelism in programs written for uniprocessors.
2. It is hard to write parallel programs when the parallelism has to be explicitly controlled by the programmer, e.g., by using *monitors*.

3. Using a single processor is sufficient for many problems.

The reason why existing compilers have problems parallelising existing programs is that most programming languages are hard to analyse for a compiler. Loops not containing procedure calls are the only constructs that have been automatically parallelised successfully in imperative languages. However, Singh and Hennessy [181] show that loop parallelisation is not enough to achieve significant speed-ups in real applications.

Since compilers cannot parallelise most real programs, programmers are forced to write their programs using operating system primitives, and in that way control the parallelism themselves. This requires extensive knowledge about the system, and the resulting program is usually complex and can only be efficiently executed on some given architecture, if at all.

One would like to have a programming language which can be executed efficiently on multiprocessor as well as uniprocessor computers, without modifying programs. It is also desirable that the programming language takes care of all underlying problems when executing a program in parallel, e.g., synchronisation between parallel processes, data distribution, etc.

We have investigated two solutions which are both inspired by the data-parallel work in languages such as Fortran: bounded quantifications and recursion-parallelism. These represent two different approaches to expressing data-parallelism in Prolog. Both aim at presenting an easily understood execution model to the programmer. Bounded quantifications do this through an extension to Prolog which makes it easier to express some algorithms, such as numerical algorithms. Reform Prolog transforms structure-recursion to data-parallelism.

Our scientific contributions are:

1. We show how recursion-parallelism, a restricted form of dependent AND-parallelism, can be efficiently implemented in Reform Prolog. Only small modifications to a sequential Prolog system are necessary.

2. We show how Prolog can be extended with bounded quantifications, as a convenient way of expressing iteration. We describe how they can be efficiently executed sequentially, and in parallel using the implementation techniques developed for Reform Prolog.

3. We present a generational copying garbage collector for Prolog, which correctly handles internal pointers. We show that instant reclaiming of data, which requires that the allocation order is preserved, can be sacrificed for data surviving garbage collection without significant efficiency penalties.

4. We describe a mark-copy parallel generational copying garbage collector for Prolog. We show how both the marking and the copying phase are parallelised and load-balanced.

From the programmer's point of view our achievements are:

1. Parallelism is transparent to the programmer. An efficient parallel program is also an efficient sequential program.
2. The programmer can reason about parallel efficiency at a high level without having to second guess neither the compiler nor the runtime system. Performance is predictable.
3. Synchronisation and data-distribution can be reasoned about at a high level for achieving optimal parallel performance. The parallelism does not change the sequential semantics of the program.

## 1.2 BACKGROUND

### Prolog and Prolog Machines

Prolog is commercially the most successful member of the family of logic programming languages. It is based on first order Horn clauses, a restricted subset of Predicate Calculus. We will not attempt to give an introduction to logic programming and Prolog. Instead we recommend the textbook by Sterling and Shapiro [186] for an introduction to Prolog, and Kowalski's book [117] for an introduction to programming in logic. Finally, the textbook by Lloyd [127] is recommended for an introduction to the theory behind logic programming.

Prolog has its origin in Robinson's [160] Resolution Principle for automated theorem proving from the early 1960's. Colmerauer, Kanoui, Pasero and Roussel [56] developed the first Prolog interpreter in 1972. Kowalski [116] showed how logic can be used for programming using declarative and procedural readings of clauses.

Warren [213, 217] developed the first Prolog compiler which became known as DEC-10 Prolog. This brought Prolog performance to a level comparable to LISP implementations. This implementation open-coded unifications, i.e. specialised tests and assignments replaced calls to a general unification algorithm.

Warren presented his "new Prolog Engine" [214] in 1983. This design included a register-based abstract machine which allowed several new space and time optimisations. It has proved to be very efficient. Most current Prolog implementations, both sequential and parallel, are based on that abstract machine which now is called Warren's Abstract Machine (WAM).

Recent high performance Prolog implementations are not directly based on WAM. Van Roy's [208, 210] Aquarius Prolog uses a WAM-inspired abstract machine (Berkley Abstract Machine). Taylor's [188] Parma uses an intermediate form close to 3-address code. Both systems make extensive use of abstract interpretation to perform optimisations.

The tutorial by Aït-Kaci [1] and Van Roy's [209] "1983–1993: The wonder years of sequential Prolog implementation" are recommended for an introduction to Prolog implementation and WAM.

### Prolog as a Parallel Language

Initiated in the early 1980's the Fifth Generation Computer Systems project in Japan was one of the largest coordinated research actions aimed at parallel (logic) programming. In the preliminary report Moto-oka [142] defines the requirements of the future computer system. The system should, among other things, be easily programmed in order to reduce the *software crisis*. They chose the high-level language Prolog as the programming language to parallelise. The reason for this was that Prolog contains many opportunities for parallelism: AND-parallelism, OR-parallelism, and unification parallelism. AND-parallelism can be described as resolving many atoms in the resolvent in parallel. If a variable appears in two different atoms in the resolvent at runtime, then these atoms are said to be *dependent*. Resolving dependent atoms, in parallel, is referred to as *dependent* AND-parallelism. Resolving atoms that are not dependent, in parallel, is called *independent* AND-parallelism. OR-parallelism, on the other hand, corresponds to trying to resolve a given atom in the resolvent with several clauses, in parallel. In addition to containing many opportunities for parallelism, Prolog programs are easy to analyse, especially when compared with C and Fortran programs.

It was perceived as difficult to parallelise Prolog. A number of modifications to Prolog were proposed in order to simplify the problem.

1. Explicit concurrency was introduced.
2. General nondeterminism with backtracking was abandoned for don't-care nondeterminism (or committed-choice).
3. Deep guards were abandoned for flat guards.
4. General unification was abandoned for dynamic synchronisation on read-only variables, which in its turn was abandoned for synchronisation on statically-declared arguments.
5. Instead of output unification, variables were allowed only one producer and, in some languages, only one consumer.

Examples of languages which exhibit one or more of these simplifications are: Concurrent Prolog [173, 174], FCP [175, 226], Parlog [53, 54], (F)GHC [203, 204], Strand [77] and Janus [168].

These simplifications made it harder to write programs in these languages, the introduction of explicit concurrency being one of the main reasons. The following quote is from Tick's book on parallel logic programming [193, p. 410]. The book contains many programs written for different parallel logic programming systems. The quote is from the section in which he discusses how easy it was to write those programs.

> "In general, it was easy to write all of the programs in Prolog. However, it was difficult to rewrite problems without OR-parallelism to run efficiently under Aurora [an OR-parallel Prolog]. It was difficult to implement logic problems, such as Zebra and Salt & Mustard, in FGHC. This difficulty occurred because we lacked meta-logical builtins in Panda [a FGHC implementation], but moreover because backtracking over logical variables must be simulated. Such difficulty is also seen when comparing the coding effort to implement complex constraints—Prolog was significantly easier than FGHC, as shown in Turtles and layered-stream programs in general."

Tick's experience is representative for most of the logic programming community. It is evident that Prolog is to be preferred. It might be possible to execute de-evolutionised languages such as FGHC efficiently, but the programming effort involved in writing programs in these languages is in many cases much larger.

## 1.3 REFORM PROLOG

### Design and Implementation (paper I and II)

Most systems for AND-parallel logic programming define the procedural meaning of conjunction to be inherently parallel. These designs are based on an ambition to maximise the amount of parallelism in computations. In paper I and II we present and evaluate an approach to AND-parallelism aimed at maximising not parallelism but machine utilisation. The system supports selective, user-declared, parallelisation of Prolog.

Reform Prolog supports parallelism only across the different recursive invocations of a procedure. Each such invocation constitutes a process, which gives the programmer an easy way of estimating the control-flow and process granularity of a program. We refer to this variant of (dependent) AND-parallelism as *recursion-parallelism*.

We implement recursion-parallelism by *Reform compilation* [138, 136] (this can be viewed as an implementation technique for the Reform inference system [200]). This is a control-structure transformation that changes the control-flow of a recursive program quite dramatically. When invoking a recursive program with a call of size $n$ (corresponding to a recursion depth $n$) a four-phase computation is initiated:

1. A single head unification, corresponding to the $n$ small head unification with normal control-flow, is performed.
2. All $n$ instances of the calls to the *left* of the recursive call are computed in parallel.
3. The program is called recursively. This call is known to match the base clause. Hence, in practice, this call is often trivially cheap.
4. All $n$ instances of the calls to the *right* of the recursive call are computed in parallel.

This approach is somewhat akin to loop parallelisation in imperative languages such as Fortran. However, an important difference is that the granularity of top-level (or near top-level) recursive Prolog procedures typically far exceeds that of parallelisable Fortran loops.

One restriction is introduced on the recursive predicates subject for parallelisation: bindings of variables shared between recursive calls of the predicate must be unconditional. This is not a severe restriction in practice.

The execution model has two major advantages:

First, a static process structure can be employed. That is, all parallel processes are created when the parallel computation is initiated. In most other systems for parallel logic programming processes are dynamically created, rescheduled and destroyed during the parallel computation.

A consequence of the static process structure is that process management and scheduling can be implemented very efficiently. It is possible to use different scheduling techniques developed for imperative languages, ranging from completely static to fully dynamic scheduling, depending on the structure of the computation. This opens up for high parallel efficiency (91–99% on the programs tested). Another consequence is that it is easy for the programmer to see which parts of the program are going to execute in parallel. This facilitates the task of writing efficient parallel programs.

Second, it is possible by global data-flow analysis to optimise the code executed by each parallel worker very close to ordinary sequential WAM code. This results in very low overheads for parallelisation (2–12 % on the programs tested).

The apparent drawback of this approach is that not all available parallelism in programs are exploited. This is, however, a deliberate design decision: exploiting as much parallelism as possible is likely to lead to poor machine utilisation on conventional multiprocessors.

### Compilation and Optimisation (paper III)

In paper III we describe the compiler analyses of Reform Prolog and evaluate their effectiveness in eliminating suspension and locking on a range of benchmarks. Suspension is necessary for maintaining the sequential semantics when shared variables are used for communicating results between recursion levels. Shared variables must be locked before they are bound to to ensure that other workers (processing elements) do not get access to partially created structures. However, suspension code and locks need not be used for ground, non-shared (i.e. local) data, and some shared variables which are not subject to time critical operations.

We find that 90% of the predicate arguments are ground or local, and that 95% of the predicate arguments do not require suspension code. Hence, very few suspension operations need to be generated to maintain sequential semantics. The compiler can also remove unnecessary locking of local data by locking only updates to shared data; however, even though locking writes are reduced to 52% of the unoptimised number for our benchmark set, this has little effect on execution times. We find that the ineffectiveness of locking elimination is due to the relative rarity of locking writes, and the execution model of Reform Prolog, which results in few invalidations of shared cache lines when such writes occur.

The benchmarks are evaluated on KSR-1 [42], a cache-coherent multiprocessor with physically distributed memory, using up to 48 processors. The performance measurements on the KSR-1 can be summarised as follows.

- Low parallelisation overhead (0–17%, with the larger benchmarks in the range of 2–6%).
- Good absolute parallel efficiency on 48 processors (82–91%) provided that there is enough parallelism in the program.

Our data indicate that each process executes in a mostly sequential fashion: suspension and locking is rare. Hence, sequential compiler technology should be largely applicable to our system.

### Nested Execution (paper IV)

In paper IV we propose a scheme for executing nested recursion parallelism. The scheme requires only minimal extensions to the flat execution model of Reform Prolog [34].

It is possible to transform some nested recursions into a single recursive loop. However, it is not always feasible to flatten a nested recursion in Reform Prolog, e.g., when the size of the nested recursion cannot be statically determined. Suppose we have the following program:

```
p([],[]).
p([H|T],[NewH|NewT]) :-
        state_size(H,State,N),
        q(N,State,NewH),
        p(T,NewT).

q(0,_,Result) :- !, Result = [].
q(N,State,[H|T]) :-
        foo(N,State,H),
        N2 is N - 1,
        q(N2,State,T).
```

Then we would have to choose between parallelising the outer loop (p/2) or the inner loop (q/2). p/2 is the obvious choice provided that not both p/2 and q/3 are shallow recursions. It would then be desirable to parallelise both and distribute the work among the processing elements (workers).

This paper shows that there is no need for a complicated implementation technique to efficiently take advantage of nested data parallelism, and consequently a substantial part of nested *dependent* control-AND-parallelism using Lindgren's transformation [126].

Initiating parallel execution, and obtaining work when suspended, is only allowed in deterministic states. Nested work is only allowed to be obtained from the left of the suspension point in the resolvent.

These restrictions make it possible to efficiently exploit nested parallelism with exceptionally small overheads in term of execution time and implementation complexity.

### Scientific Contributions

The research on Reform Prolog has shown that data-parallelism, in the form of recursive predicates, can be implemented efficiently. We have shown that dependent AND-parallelism can be exploited with small modifications to a sequential system, and with small overheads for parallel execution.

Work is still needed to bring the performance closer to parallelised Fortran loops, but the potential has clearly been shown.

Reform Prolog is a fairly mature parallel system at this point. It is built to handle large programs; all data areas are expanded on demand and an efficient parallelised garbage collector is present. It has been ported to a number of large shared memory architectures.

### Author's Contributions

The execution model presented in paper I and II was designed together with Thomas Lindgren and Håkan Millroth. Thomas Lindgren designed and implemented the compiler and I designed and implemented the abstract machine and runtime system.

In paper III my contributions were to design the optimisations together with Thomas Lindgren, implement the emulator parts, and perform the experiments on the KSR-1.

I designed the scheme for nested parallel execution.

## 1.4 BOUNDED QUANTIFICATIONS

### Proposal and Sequential Execution (paper VI)

In Prolog, the only way to express iteration is by using recursion. This is theoretically sufficient but many algorithms are more elegantly expressed using definite iteration. That is, combining the results of letting a variable range over a finite set, and for each value repeating some computation. Most imperative languages provide such iteration over finite sets of integers, e.g., **for** loops in Pascal and **DO** loops in Fortran, combining the results through serialisation.

Previously Barklund & Millroth [17, 26] and Voronkov [211, 212] have presented *bounded quantifications* as a concise way of providing definite iteration in logic programs. In paper VI we propose an extension of Prolog with bounded quantifications and argue that it then naturally follows to introduce arrays. We present a sequential operational semantics for bounded quantifications and show how they can be implemented through modifications to Warren's Abstract Machine.

A *quantification* is an expression consisting of three parts: a *quantifier*, which is a symbol from a given alphabet of quantifiers, a locally scoped *iteration variable*, and a *body*, which is itself an expression. A *bounded quantification* is a quantification where the local variable ranges over a *finite* set of values, given by a *range expression*. We define the semantics of a quantification to be that obtained by evaluating the body for every value ranged over by the local variable, combining the results according to the quantifier. This combination is typically a reduction with an associative and commutative binary function, in which case the quantifiers can alternatively be seen as a reduction operator, but it is not always the case.

Some examples of bounded quantifications using ordinary mathematical notation are

$$(\forall\, 5 \leq i < 10)\, p(i) \leftarrow q(i)$$

$$\text{and} \quad \textstyle\sum_{0 \leq k < z} \frac{1}{(2k+1)(2k+3)}$$

where $i$ in the first and $k$ in the second quantification should be considered implicitly restricted to take on integer values only. The first quantification is truth-valued while the second expression is number-valued (approximating $\pi/8$ for large values of $z$).

Much of the beauty of bounded quantifications lies in the fact that they have a clear declarative semantics, while at the same time they behave well operationally on both sequential and parallel computers.

## Parallel Implementation (paper V)

In paper V we show that many Prolog programs expressed with bounded quantifications can be run efficiently on parallel computers using the same data-parallel implementation technique as Reform Prolog. In order to do so, we describe a quite straightforward transformation from bounded quantifications to recursive programs.

Note, however, that the parallel implementation cannot take full advantage of the simplicity of bounded quantifications when only the transformed programs are available. Some implementation methods needed for optimal execution of bounded quantifications may not be reasonable for arbitrary recursive programs. Hence, the concurrency in programs may not be detected and some programs may not be run in the most efficient way. In the future it therefore seems worthwhile to investigate methods that are applied directly to bounded quantifications. Until such methods are available, the methods described herein are appropriate.

Our results indicate almost linear speed-up on 24 processors for some quantified expressions. The parallelisation overhead is low, in the order of 10-15%. A best speed-up of 34.38 for a bounded quantification program compared with sequential execution of a corresponding recursive program, and 19.18 compared with sequential execution of the bounded quantification, was achieved using 24 processors.

From this we conclude that from a performance point of view, there is no significant difference between recursion and quantifications, and the programmer should be free to use the programming construct (bounded quantifications or recursion) that is suitable for a given problem—both can be efficiently executed on a shared memory multiprocessor.

### Scientific Contributions

It is proposed to add *bounded quantifications* to Prolog. The main reason is one of natural expression; many algorithms are expressed more elegantly in a declarative way using bounded quantifications than using existing means, i.e., recursion. In particular this is true for numerical algorithms, an area where Prolog has been virtually unsuccessful so far. Moreover, bounded quantification has been found to be at least as efficient as recursion, when applicable. We outline an implementation of some bounded quantifications through an extension of Warren's abstract Prolog machine and give performance figures relative to recursion. We have shown that bounded quantification has a high potential for parallel implementation and we conclude that one can often run the same program efficiently on a sequential computer as well as on several kinds of parallel computers.

### Author's Contributions

Both the language extensions and the operational semantics were done in collaboration with Jonas Barklund. I implemented most of the abstract machine extensions. The benchmarks were performed together with Jonas Barklund.

## 1.5 GARBAGE COLLECTION

Prolog programmers are relieved of the burden of managing memory. It is up to the runtime system to efficiently reclaim unused data through garbage collection. This procedure must be fast to achieve high system performance since Prolog programs usually create large amounts of data.

It has been observed in other languages, with similar allocation patters as Prolog, that most data tend to be short lived [69, 207]. This insight has led to the invention of generational garbage collection [121] where young objects are garbage collected more often than old.

### Sequential Copying Collection (paper VII)

In paper VII we describe a technique for adapting conventional copying garbage collection to Prolog and how to extend the new scheme for generational collection. Three problems have been solved, leading to efficient copying and generational copying collectors.

1. The first problem is interior pointers, which can lead to duplication of data if copied naively. *Interior pointers* are direct references to the contents of a structure. Such pointers are normally not present and previous algorithms do not take them into account. Our method correctly handles interior pointers by marking, then copying data.

2. The second problem is that copying collection does not preserve the heap ordering. In Prolog the heap can be viewed both as an ordinary heap and as a stack. The stack property is used to efficiently deallocate object from the heap when the computation backtracks. Since the heap order is not preserved, this stack behaviour cannot be used and memory cannot be reclaimed by backtracking. The heap order is also used for deciding if changes to the heap must be recorded (trailed) to be able to restore a previous state. The disrupted heap order means that all changes must be recorded (trailed).

   Our collector exploits that data allocated since the last collection still retain the desired heap ordering. Hence, memory allocated after the last collection can still be reclaimed by backtracking. Our measurements show that our copying algorithm recovers as much memory by backtracking as a conventional ("perfect") mark-sweep algorithm on a range of realistic benchmarks.

   We have also measured the amount of extra trailing due to losing the order of the heap. This was negligible: less than one-quarter of a percent of the total number of trailings at most. We conclude that copying collection is a viable alternative to the conventional mark-sweep algorithm for Prolog.

3. The third problem is how to extend the copying algorithm to generational collection. The crucial insight is that pointers from the old generation (in a two-generation system) can be found by scanning the change record (trail). By adapting the trailing mechanism, we get an almost-free write-barrier [207, 141]. The only extra cost is some unnecessary trailings in certain situations. This cost is again negligible for our benchmarks.

**Parallel Generational Collection (paper VIII)**

We present a parallel generational copying garbage collection scheme for Prolog in paper VIII. The algorithm uses a mark and copy technique for handling internal pointers.

The algorithm uses ideas from our sequential Prolog collector, and from a general parallel copying collection scheme described by Ali [3].

We show how the resulting collector can be made generational. An improved strategy for load balancing for Prolog is presented.

Our main contributions are:

- A reasonable way of dealing with internal pointers; load balanced parallel marking of live data.
- Parallel generational garbage collection.
- Prolog specific considerations, i.e. handling internal pointers, a modified load balancing scheme, and a scheme for ordering variables.
- An evaluation of how well the algorithm behaves in practice.

### Scientific Contributions

The sequential collection scheme showed how a generational copying collector could be applied to Prolog. Before this only mark-compact collectors and partially copying collectors were used. Our scheme showed how internal pointers were handled and how generational collection could be provided. Demoen, Engels and Tarau [67] have improved the scheme further using ideas from Bekkers, Ridoux and Ungaro [28].

This work also showed that instant reclaiming on backtracking can be sacrificed for data which survive garbage collection, without significant efficiency penalties. This is crucial for the parallel collector.

The main contribution for parallel collection is to show how Prolog specific problems are solved, i.e.

- load balanced parallel marking for handling internal pointers
- ordering of variables when their heap address cannot be used
- detection of cross generational pointers for generational collection
- improved load balancing for Prolog

It is possible to apply a number of different parallel copying schemes in this setting.

### Author's Contributions

The sequential collector and the benchmarking methodology was designed in collaboration with Thomas Lindgren. I implemented the ideas and performed the benchmarking.

I implemented and designed the parallel collector.

## 1.6 RELATED WORK

### Parallel Prolog

The earliest published work on parallel Prolog is Pollards PhD thesis [154] where execution of pure OR-parallelism is discussed. The possibility of AND-parallel execution was noted by Kowalski [116] in 1974.

Parallel Prolog can be divided into two categories: OR-parallel and AND-parallel. Our research falls into the AND-parallel category and we will not elaborate on the OR-parallel field.

Conery and Kibler [57] proposed the first scheme for OR- and AND-parallel execution, where dependencies for the AND-parallel execution were determined using an ordering algorithm. The dependencies were calculated when entering a clause and updated when each body goal succeeded. The scheme has not been considered practical due to substantial overheads.

Lin and Kumar [122, 123, 124] developed a more efficient version of Conery and Kibler's scheme where tokens are associated with each variable for determining the dependencies dynamically. Compile time information is also used to reduce the runtime overheads.

Somogyi, Ramamohanarao and Vaghani [183] developed a stream AND-parallel version of Conery and Kibler's scheme. In this system only one goal, the *producer*, is allowed to bind a variable. All other goals are *consumers*. A consumer is not allowed to bind a variable, it suspends until the variable becomes bound. This form of dependent parallelism has its origin in IC-Prolog [51].

DeGroot [66] proposed a scheme for *restricted AND-parallelism* (RAP), i.e. only independent goals were allowed to execute in parallel. Runtime tests were added for determining variable groundness and independence. The goals were executed in parallel when the tests succeeded. This scheme exploited less parallelism than Conery and Kibler's but was expected to have substantially smaller runtime overheads.

Hermenegildo [95] refined DeGroot's scheme and added a backtracking semantics. Hermenegildo [94, 99, 100, 145] defined and implemented the first independent AND-parallel system: &-Prolog. &-Prolog has played an important role for AND-parallel implementations. System such as ACE [84, 85] and DDAS [178, 176, 177] are largely based on &-Prolog.

Chang, Despain and DeGroot [47, 48] were first to propose that scheduling of parallel execution could be done through compile time analysis. Borgwardt [40, 41] proposed a stack-based implementation of the execution model developed by Chang et al. [48].

Hermenegildo and Rossi [100] classified independent and parallelism as strict independence and non-strict independence. No free variables were allowed to be shared in *strict* independence, while *non-strict* independence allowed unbound variables to be passed around as long as they were only used, i.e., bound to a non-variable value, by at most one goal.

Yang [224] observed that complicated backtracking schemes can be avoided if only deterministic computations are executed in parallel. When conflicting bindings occur the entire computation fail.

In PNU-Prolog, Naish [146, 151] requires that all bindings are deterministic during parallel execution. Non-deterministic goals are allowed as long as no non-determinate bindings are created. Conflicting bindings result in global failure. Naish's ideas on deterministic binding and computation have been one of the major influences on Reform Prolog.

Warren used ideas similar to PNU-Prolog when he formulated the Basic Andorra Model [89, 216] where deterministic goals are executed in parallel, and non-deterministic goals are suspended. The Basic Andorra Model has been implemented in Andorra-I [164, 165, 166, 225].

In the Extended Andorra Model [90, 163], a copy of the computation is created for each possible binding of a variable when all deterministic goals have been executed. The extended model is implemented in AKL [107, 108, 140].

Gupta, Santos Costa, Yang, and Hermenegildo [83] combine independent AND-parallelism, deterministic dependent AND-parallelism and OR-parallelism in IDIOM. This built on ideas from AO-WAM [86], which is an extension of RAP-WAM [95] with OR-parallelism.

Another approach to solving the problem with conflicting bindings is to execute different branches separately and join the bindings when the branches terminate. This is often combined with OR-parallelism. Examples of such systems are Wise's EPILOG [222] and Wrench's APPNet [223].

Shen has developed a scheme for general dependent AND-parallelism called DDAS [176, 177, 178, 179]. Subgoals that share a variable execute in parallel until the variable is accessed. The consumer suspends until the variable is bound or it becomes the producer. The producer and consumer of a variable is dynamically recognised, and sufficient information is saved for selectively undoing speculative work when a conditional binding of a shared variable is undone. Dependence variables must be annotated either by the programmer or the compiler. The current implementation of DDAS does not provide automated variable annotation, but Shen [180] mentions work in progress.

Another approach to parallelism is to introduce explicit concurrency. This was done in IC-Prolog by Clark and McCabe [51] and in Relational Language by Clark and Gregory [52]. These ideas were adopted by others and resulted in the concurrent logic languages: Concurrent Prolog [173], FCP [175], Parlog [53, 60], Strand [77], Guarded Horn Clauses (GHC) [201, 202], and Flat GHC [50, 205]. These languages are also referred to as de-evolutioned by Tick [194] since they have been significantly restricted to allow efficient implementation.

Most of the AND-parallel systems described above are implemented as extensions of WAM. Hermenegildo, Cabeza and Carro [97] recently proposed an elegant scheme where attributed variables [147, 120] are used for handling most of the support for parallelism. This technique makes it possible to model different forms of parallelism using the same runtime machinery, with some decrease in efficiency.

## Data OR-parallelism

Smith [182] described Multilog, a system utilising data OR-parallelism. Multiple binding environments are generated for an annotated goal and subsequent goals are executed in these environments, in parallel. The result is a partial breadth-first search which replaces backtracking. Typical applications are generate-and-test programs.

## Recursion Parallelism and Data AND-parallelism

Tärnlund [199] observed that parallelism could be increased by unfolding recursive calls to a predicate. He noted that clauses could be unfolded such that the entire recursive structure was captured by one head unification. Unfolding also resulted in one large conjunction which could be executed in parallel. The unfolding could be performed in $\log N$ steps, where $N$ is the size of the input. Tärnlund defined the Reform inference system [200], based on these ideas.

Inspired by Tärnlund's ideas Millroth developed *Reform Compilation* [135, 136, 137, 138] which is a technique for compiling linear structurally recursive predicates into parallel code.

Consider a recursive predicate that can be written on the following form.

$$\begin{array}{lcl} p(\bar{x}) & \leftarrow & \Delta \\ p(\bar{x}) & \leftarrow & \Phi, p(\bar{x}'), \Psi \end{array}$$

If a goal $p(\bar{y})$ is determined to recurse at least $n$ times, then the second clause of p/2 can be unfolded $n$ times resulting in the following clause.

$$p(\bar{x}') \leftarrow \Phi_1, \cdots, \Phi_n, p(\bar{y}'), \Psi_n, \cdots, \Psi_1.$$

This clause is then executed by first running the $\Phi_1 \cdots \Phi_n$ goals in parallel, then executing $p(\bar{y}')$ (usually the base case), and finally running the $\Psi_n \cdots \Psi_1$ goals in parallel.

Warren and Hermenegildo [219] performed some early experiments with what they called MAP-parallelism. This was a limited form of independent data AND-parallelism where a procedure was applied over a set of data, similarly to mapcar in Lisp and DOALL in Fortran.

Harrison [92] has developed a scheme for exploiting recursion parallelism in Scheme which is similar to Reform Prolog. The main difference is that Harrison only handles DOALL loops and recurrences which do not require synchronisation between different recursion levels. Reform Prolog handles general DO-ACROSS loops. Also, Prolog uses side-effects for variable binding to a much higher degree than Scheme, resulting in a different system design.

Sehr, Kale, and Padua [172, 171] independently discovered recursion parallelism in Prolog. However, their ideas are much less developed. Only inner loops are parallelised, similarly to Fortran. The consequence is that less work can be parallelised and the exploited parallelism is fine-grained. Also, no compiler nor implementation exists. A compilation technique is outlined but never implemented.

Hermenegildo and Carro [98] and Debray and Jain [64] discuss how goals can be spawned more efficiently using program transformation techniques. These have some similarities to the transformations described by Tärnlund [199] in his thesis. Hermenegildo and Carro also discuss how the &-Prolog implementation can be extended with low level primitives to support data-parallelism.

Pontelli and Gupta [87, 156, 157, 158, 159] present a number of similar techniques for minimising the overheads for creating processes in &ACE. They then argue that data-parallelism can be efficiently exploited. However, there are still some significant differences compared to Reform Prolog. They can only exploit independent AND-parallelism. Their implementation technique is also considerably more complicated, resulting in higher overheads and complications when applying high performance sequential implementation techniques.

A different approach to data-parallel logic programming is to construct a data-parallel interpreter for the language. The data represents the program combined with its state. Reductions and inferences may be performed in parallel. This technique results in significant overheads compared with compiled implementations. It has therefore mostly been aimed at massively parallel machines.

Kacsuk designed a parallel interpreter for Prolog based on his generalised data flow model [111, 112, 113]. This in turn was inspired by other data flow models for parallel execution of Prolog proposed by Moto-oka and Fuchi [143] and Umeyama and Tamura [206]. These execution models were motivated from the research on data flow parallelism which emerged in the early 1980's [12, 68].

Nilsson and Tanaka [148, 149] designed a scheme for compiling Flat GHC into Fleng. Fleng is a primitive process-oriented language which has been implemented using a data-parallel interpreter. Barklund, Hagner and Wafin [21, 22] translated a flat committed choice language into condition graphs, and proposed a data-parallel inference mechanism. Barklund [16] also proposed a data-parallel unification algorithm suitable for data-parallel logic programming implementations, e.g., Reform Prolog.

Yet another approach to data-parallelism is to add parallel data structures on which certain operations can be performed in parallel. This is the approach taken by Kacsuk in DAP Prolog [113], Barklund and Millroth in Nova Prolog [24], and by Fagin [75]. The idea of having explicitly parallel data structures have previously been used in other languages such as APL [106], CM Lisp [102, 185], ⋆Lisp [191], NESL [38] among others. These languages are often designed to exploit the architecture of a specific machine, i.e. CM Lisp was designed for the Connection Machine and DAP Prolog for the Distributed Array Processor.

**Loop Parallelisation in Imperative Languages**

Parallelising iteration in imperative languages have received much attention since it potentially may increase the speed of large sets of existing applications. Most of the research have focussed on parallelising DO-loops in Fortran. Kuck, Kuhn, Leasure, Wolfe [119], Kennedy [114], Burke and Cytron [43] among others have looked into this.

The problem has proven to be difficult to solve for general loops. The best results have been obtained for loops performing simple operations on arrays. Singh and Hennessy [181] observe that loop-parallelisation is insufficient for extracting enough parallelism from the programs they have examined. One of the reasons is that it is difficult to analyse loops containing procedure calls. Languages containing arbitrary pointers, such as C, are particularly difficult to parallelise. In Prolog, a disciplined form of pointers are present in the form of logical variables.

**Bounded Quantifiers**

Voronkov [211, 212] has independently studied bounded quantifiers in logic programming. He introduces a class of generalised logic programs, where certain list relations can be elegantly expressed using bounded quantifications with head/tail iterators. He defines both a declarative semantics and a procedural interpretation (a complete variant of resolution called SLDB-resolution), and presents a translation from generalised logic programs to Horn clause programs. Voronkov notices the potential for AND-parallel execution in bounded universal quantifications, and the potential for OR-parallel execution in existential quantifications. He also mentions previous work in Russia.

Our approach is slightly different. We view bounded quantifications as an elegant way of expressing iteration and data-(AND-)parallelism. We present Prolog extensions for certain bounded quantifications, and describe WAM based compilation schemes for quantifications ranging over integers and lists. We evaluate their implementations for sequential and parallel execution.

Kluźniak has (also independently) proposed SPILL [115], a specification language which includes certain bounded quantifications with integer ranges. Specifications written in SPILL are executable by translating them to Prolog, according to a set of translation rules. However, the quantifications are translated essentially as by Lloyd & Topor (cf. below).

Some authors have studied the use of bounded quantifiers with sets, for example in SETL [170] and {log} [70].

Barklund & Hill [23] have studied how to incorporate restricted quantifications and arrays in Gödel [101], while Apt [10] has studied how bounded quantifications and arrays could be used also in constraint based languages.

Lloyd & Topor [128] have studied transformation methods for running more general quantifier expressions, although their method will 'flounder' for some examples that can be run using our method (Lloyd, personal communication). Sato & Tamaki [167] have an interpreter that will run more general quantifier expressions although the method is currently not so efficient (Sato, personal communication). In comparison, our approach is only applicable to range-restricted formulas, but is quite efficient for that case.

The methods by Meier [133] for compiling recursive programs to iterative code are also relevant. For tail recursive programs it seems to us that Meier gets code which is quite similar to ours for the corresponding bounded quantification (except that we have defined a small collection of new abstract machine instructions while his code is a mix of WAM, C, etc.). Meier also considers "backtracking" iteration, a subject we have not discussed here (compilation of bounded existential quantifications). It seems to us that the instruction set we use would be appropriate as a base also for Meier's methods.

Array comprehensions in Haskell [5] can be used for expressing array operations, similar to our array extended bounded quantifications.

The Common LISP language [184] (and some earlier LISP dialects), as well as Standard ML [91], contain iteration, mapping and reduction constructs that in some cases resemble ours.

The idea for using bounded quantification in logic programming was inspired by Tennent's proposed use of them in ALGOL-like languages [189, 190], and also by an exposition by Gries [82].

Hermenegildo and Carro [98] discuss how parallel execution of bounded quantifications relates to more traditional AND-parallel execution of logic programs.

Arro and Barklund [11] have investigated how bounded quantifiers can be executed on the Connection Machine, a SIMD multiprocessor that directly supports data parallel computation.

Finally, we should mention that transforming recursive programs to iterative programs is an activity that has been studied extensively in computing science. This often involves tabulation techniques [30, 36, 80] and has also been applied in logic programming [18, 218].

### Garbage Collection of Prolog

*Sequential Collection* It has been observed in other languages that most data tend to be short lived [69, 207]. This insight led to the invention of generational garbage collection [7, 121] where young objects are garbage collected more often than old. Copying collectors [76, 139] and generational copying collectors have been considered unsuitable for Prolog until recently. Prolog implementations such as SICStus Prolog [45] use a mark-sweep algorithm [131] that first marks the live data, then compacts the heap. We take the implementation of Appleby et al. [9] as typical. It is based on the Deutsch-Schorr-Waite [55, 169] algorithm for marking and on Morris' algorithm [55, 144] for compacting.

Touati and Hama [196] developed a generational copying garbage collector for Prolog. The heap is split into an old and a new generation. Their algorithm uses copying when the new generation consists of the top most heap segment, i.e., no choice point is present in the new generation, and no troublesome primitives have been used (primitives that rely on a fixed heap ordering of variables). For the older generation they use a mark-sweep algorithm. The technique is similar to that described by Barklund and Millroth [27] and later by Older and Rummell [150].

Bekkers, Ridoux and Ungaro [28] describe an algorithm for copying garbage collection of Prolog. They observe that it is possible to reclaim garbage collected data on backtracking if copying starts at the oldest choice point (bottom-to-top). However, their method has several differences to ours.

- Their algorithm does not preserve the heap order, which means that primitives such as `@</2` will work incorrectly. They do not indicate how this problem should be solved.
- Their algorithm (the version that incorporates early reset) copies data twice, while our algorithm visits data once and then copies the visited data. We think our approach leads to better locality of reference. However, we have not found any published measurements of the efficiency of the Bekkers-Ridoux-Ungaro algorithm.
- Variable shunting [46, 120], i.e. collapsing variable-variable chains, is used to avoid duplication of variables inside structures. However, this technique may introduce variable chains in new places. We want to avoid this situation.

Their algorithm does preserve the segment-structure of the heap (but not the ordering within a segment). Hence, they can reclaim all memory by backtracking. In contrast, our algorithm only supports partial reclamation of memory by backtracking.

Appel [6, 7] describes a simple generational garbage collector for Standard ML. The collector uses Cheney's [49] garbage collection algorithm, which is the basis of our algorithm as well. However, Appel's collector relies on assignments being infrequent. In Prolog, variable binding is assignment in this sense. Our algorithm handles frequent assignments efficiently.

Sahlin [162] has developed a method that makes the execution time of the Appleby et al. [9] algorithm proportional to the size of the live data. The main drawback of Sahlin's algorithm is that implementing the mark-sweep algorithm becomes more difficult, not to mention guaranteeing that there are no programming errors in its implementation. To our knowledge it has never been implemented.

The collector described by Demoen et al. [67] maintains heap segments across garbage collections, and even increases the amount of memory that can be deallocated on backtracking. It was designed after our collector and use our ideas for handling internal pointers. It is not clear how their algorithm can be made generational.

Cohen [55], Appel [8], Jones and Lins [110], and Wilson [221] have written comprehensive surveys on general-purpose garbage collection algorithms. There is also a survey of collection schemes for sequential logic programming languages by Bekkers, Ridoux and Ungaro [28].

*Parallel Collection* Most parallel Prolog implementations do not include a garbage collector. OR-parallel systems such as Muse [4] and Aurora [44, 129] use more or less sequential mark-sweep collectors [2, 71, 72]. The Aurora collector [220] designed by Weemeeuw and Demoen is slightly more complicated since Aurora uses the SRI-model [215] for OR-parallel execution with its more complicated data structures.

The closest we get to a parallel collector for an AND-parallel Prolog is Crammond's [59] mark-sweep collector for Parlog. It is parallelised by pushing all external references to each heap on a heap specific *import stack*. Each heap can then be collected using an almost sequential mark-sweep collector. The algorithm is not load balanced, and behaves reasonably only when each worker allocates an equal amount of memory. Its execution time is proportional to the size of the largest heap instead of the live data. The space requirements are, at worst, as large as for a copying collector since the import stacks may be as large as the live data.

Ali [3] describes a general copying collector for shared memory. It is a two space collector (to- and from-space) where each subspace consists of a number of segments. Processing elements (PEs) allocate data in the segments in from-space. Collection takes place when from-space fills up. During collection each PE copies its reachable data into to-space. Load balancing is provided for the coping phase. Our parallel algorithm use ideas from Ali's collector and adds support for handling internal pointers and improves on the load balancing machinery.

Imai and Tick [105] describe a parallel stop-and-copy collector based on Cheney's [49] algorithm. It provides dynamic load balancing through a global work stack. Object of equal size are allocated in the same memory block. The later makes it unsuitable for Prolog.

Baker [15] describes a concurrent collection scheme divided into two processes, a *mutator* which creates data and a *collector* which performs garbage collection. Execution of the two processes are interleaved.

Halstead [88] describe a parallelisation of Baker's algorithm. The heap is statically divides into separate areas collected by distinct PEs. No support for load balancing is provided. Herlihy and Moss [93] developed a concurrent lock-free version of Halstead's algorithm.

Lieberman and Hewitt [121] describe a real-time generational collector in which all pointers from older to newer generations pass through an indirection table. Our implementation instead uses the trail for pointing out references from the old to the new generation.

Huelsbergen and Larus' [103] developed a concurrent copying garbage collector for shared memory with two processes; collector and mutator. Ellis, Li and Appel [73] propose a similar design with several mutators and one collector. Röjemo [161] extended the collector by Ellis et al. for the $\langle v, \mathrm{G}\rangle$-machine [13] (a parallel version of the G-machine [14]).

# Reform Prolog: The Language and its Implementation

Johan Bevemyr, Thomas Lindgren, Håkan Millroth

In *Proc. 10th Int. Conf. Logic programming*, MIT Press, 1993.

Reform Prolog is an (dependent) AND-parallel system based on recursion-parallelism and Reform compilation. The system supports selective, user-declared, parallelization of binding-deterministic Prolog programs (non-determinism local to each parallel process is allowed). The implementation extends a conventional Prolog machine with support for data sharing and process management. Extensive global dataflow analysis is employed to facilitate parallelization. Promising performance figures, showing high parallel efficiency and low overhead for parallelization, have been obtained on a 24 processor shared-memory multiprocessor. The high performance is due to efficient process management and scheduling, made possible by the execution model.

## 2.1 INTRODUCTION

Most systems for AND-parallel logic programming defines the procedural meaning of conjunction to be inherently parallel. These designs are based on an ambition to maximize the amount of parallelism in computations. We present and evaluate an approach to AND-parallelism aimed at maximizing not parallelism but machine utilization. The system supports selective, user-declared, parallelization of Prolog.

Reform Prolog supports parallelism only across the different recursive invocations of a procedure. Each such invocation constitutes a process, which gives the programmer an easy way of estimating the control-flow and process granularity of a program. We refer to this variant of (dependent) AND-parallelism as *recursion-parallelism*.

We implement recursion-parallelism by *Reform compilation* [138] (this can be viewed as an implementation technique for the Reform inference system [200]). This is a control-structure transformation that changes the control-flow of a recursive program quite dramatically. When invoking a recursive program with a call of size $n$ (corresponding to a recursion depth $n$) a four-phase computation is initiated:

1. A big head unification, corresponding to the $n$ small head unification with normal control-flow, is performed.
2. All $n$ instances of the calls to the *left* of the recursive call are computed in parallel.
3. The program is called recursively. This call is known to match the base clause. Hence, in practice, this call is often trivially cheap.
4. All $n$ instances of the calls to the *right* of the recursive call are computed in parallel.

This approach is somewhat akin to loop parallelization in imperative languages such as Fortran. However, an important difference is that the granularity of top-level (or near top-level) recursive Prolog procedures typically far exceeds that of parallelizable Fortran loops. A major feature of the approach is that it allows a static process structure: all parallel processes are initialized when the parallel computation starts. This has profound impact on performance.

This paper is organized as follows. The Reform Prolog execution model is defined in Section 2. In Sections 3 to 5, the design of the parallel abstract machine is discussed. The global dataflow analysis employed by the compiler is outlined in Section 6. The extension of the instruction set for parallel execution is described in Section 7. Section 8 presents our experimental results.

## 2.2 REFORM PROLOG

Reform Prolog parallelizes a deterministic subset of Prolog. Below we define the condition for when a recursive Prolog predicate can be parallelized. We then consider how this condition can be enforced by the implementation. We need to define two auxiliary concepts:

- A variable is *shared* if it is accessible from more than one recursion level. Note that a variable can be shared at one point of time and unshared (local) at another.
- A variable binding is *unconditional* if it cannot be undone by backtracking.

A call in a parallel computation is *safe* if all bindings made to its shared variables are unconditional when the call is executed. The condition for when a predicate can be parallelized is then:

> **A recursive predicate can be parallelized only if all calls made in the parallel computation are safe.**

Safeness of a call is defined wrt. to the *instantiation* of the call (i.e., what parts of the arguments are instantiated). We can distinguish between the parallel instantiation and the sequential instantiation of a call. These might differ as a parallel call can 'run ahead' of the sequential instantiation: recursion levels that would execute after the current one sequentially, may already have bound shared variables.

We say that a call is *par-safe* when it is safe w.r.t. the parallel instantiation, and that it is *seq-safe* when it is safe w.r.t. the sequential instantiation.

The compiler is responsible for checking that programs declared parallel by the programmer are safe. For calls that can be proven par-safe at compile-time, there is no need for extra safeness checking at runtime. For calls that can be proven seq-safe at compile-time, but not par-safe, it is necessary to check safeness at runtime. If the call is not safe, then it is delayed until it becomes safe. This is done by suspending until:

1. The unsafe argument has become sufficiently instantiated by another recursion level; or
2. The current call becomes leftmost.

If neither par-safeness nor seq-safeness can be proven at compile-time, parallelization fails.

The execution model described above has some similarities to the approach taken in Parallel NU-Prolog [146] in that both approaches parallelize a binding-deterministic subset of Prolog. However, Reform Prolog exploits recursion-parallelism when parallelizing this subset, whereas Parallel NU-Prolog exploits AND-parallelism.

With Reform Prolog, as with Parallel NU-Prolog, it is straight-forward to call parallel subprograms from a nondeterministic program. Thus, there is a natural embedding of parallel code in ordinary sequential Prolog programs.

Safeness allows local nondeterminism in each recursion level as long as no unsafe bindings are made. In Parallel NU-Prolog this is not done, since the entire proof tree is subject to parallelization. The consequence is that any variable may be accessible to other processes and so any binding in a nondeterminate state may be unsafe.

## 2.3 THE PARALLEL ABSTRACT MACHINE

The parallel machinery consists of a set of workers numbered 0,1,. . . ,$n-1$, one per processor. Each worker is implemented as a separate process running a WAM-based Prolog engine with extensions to support parallel execution.

### Execution phases

The execution of a program alternates between two modes: sequential execution and parallel execution. A phase of sequential execution is referred to as a *sequential phase* and a phase of parallel execution as a *parallel phase*. One worker is responsible for sequential execution (the *sequential* worker). During sequential execution all other workers (the *parallel* workers) are idle, during parallel execution the sequential worker is idle.

### Memory Layout

A standard WAM implementation has three main data areas: heap, stack and trail. The heap contains variables and data structures, the stack contains choicepoints and environments, and the trail contains references to bound variables. Environments contain only variables which are either unbound, referring to other variables in the stack, or referring to variables or structures on the heap.

In the Reform engine each worker has its own distinct data areas (heap, stack and trail). The stack and the trail are local to each worker and cannot be accessed by other workers. All heaps are shared and all workers have restricted access to other workers' heaps. The motivation for the design is given below.

**Heap.** Each parallel worker might require access to data built on the heap of the sequential worker. Moreover, if there are data dependences in the Prolog program, then parts of each workers data might have been created by other workers during the current parallel phase. The easiest way to support this is to let all workers have access to each others heaps. This access is restricted so that no worker may create new objects on another workers heap, but only bind existing variables. Such an arrangement ensures simple memory management of each heap.

**Stack.** The stacks can remain unshared if it can be ensured that no references to stack objects will occur in shared objects. Since heap variables cannot be bound to stack objects, the only restriction that has to to be imposed is to disallow stack objects in the arguments to the parallel call.

**Trail.** If each worker is responsible for any unconditional bindings it has created, then the trail can remain local. For this to work, the sequential worker must notify all workers when it backtrack across a parallel phase and let each worker undo the conditional bindings it created during that parallel phase.

### Heap-allocated vectors

Some variables, that would be allocated on the stack in a sequential WAM, are allocated in vectors stored on the heap. Consider, for example, the 'naive reverse' program:

```
nrev([],[]).
nrev([A|X],Y) :- nrev(X,Z), append(Z,[A],Y).
```

Here the variable Y on each recursion level is bound to the variable Z on the next recursion level. In a sequential WAM both Y and Z would be stored on the stack. This is not possible in our machine for two reasons. First, stacks are not shared between workers. Second, all instances of the variables are created *before* the parallel execution of the append calls is initiated, as a consequence of our execution model.

Hence all instances of the variables Y and Z are allocated in vectors on the heap rather than in binding environments on the stack. This is an example on the trade-off between efficient memory usage (stack allocation) and increased potential for parallelism (heap allocation). For tail-recursive programs this effect is even more dramatic.

The Reform engine also exploits heap-allocated vectors for another purpose. Consider the list in the first argument position of nrev/2 above. The sequential engine traverses the list and builds a vector of its elements before the parallel phase is initiated. This is necessary since we (*i*) need to know the final recursion depth (i.e. the length of the list) *before* the parallel phase is initiated, in order to determine how many parallel processes (recursion levels) to spawn, and (*ii*) need to index into the list during the parallel phase.

A problem is that after the parallel phase, the vector should be viewed as a linked list again. Our solution to this problem is to build a 'vector-list' by allocating the cons cells densely one after the the other. In this way we can index into the list during the parallel phase, and still view it as a linked structure in the following sequential phase.

## 2.4 DATA SHARING

There are two aspects of data sharing between workers. First, terms created by the sequential worker (variables, numbers, structures and lists) must be accessible to parallel workers. Second, terms created by parallel workers must likewise be accessible to other parallel workers, and to the sequential worker.

### Binding variables

The memory layout described above gives each worker the ability to refer to objects shared with other workers. This ability might lead to a situation where two workers try to simultaneously bind the same variable, potentially with the result of one binding destroying the other.

We therefore use an atomic exchange operation when binding a variable. If another worker has managed to bind the variable ahead of this worker, then the exchange operation will return that binding. The other workers binding must then be unified with this workers, to ensure consistency. A similar method is used in the implementation of Parallel NU-Prolog [146].

We have found that in our system this method is significantly faster than spin locking [132].

### Creating shared structures

When building a structure on the heap other workers should not have access to the structure until it has been fully initialized. In WAM a structure is built using either put instructions or get instructions. When put instructions are used there is no problem, since no variable is bound to the structure until it has been fully initialized. A get instruction, on the other hand, might bind a variable to an incomplete structure and then proceed to fill in the missing parts using unify instructions.

We avoid references to uninitialized structures by modifying the get instructions so that they do not bind the variable, but save it for later binding. A new instruction has to be introduced after the last unify instruction (when the structure is complete) to bind the variable. This method is also discussed in Naish's paper on Parallel NU-Prolog [146].

### Trailing

Variables must be trailed in the parallel phase, even though only safe programs are parallelized. There are two reasons for this.

1. There might be local nondeterminism within each recursion level. If that is the case, the worker must be able to backtrack locally. This forces the worker to, at least, trail bindings of local heap and stack variables ('local' variables reside in a data area managed by the worker).
2. A parallel worker might bind variables created before the parallel execution started. Since sequential backtracking is allowed across parallel calls, the machine must be able to undo bindings created during parallel phases.

The problem with trailing is that each worker has its own heap to work on. It is no longer possible to simply compare a variable's position on the heap with a heap pointer to determine whether it should be trailed or not. The situation gets even worse if we consider what might happen when the sequential worker continues executing after a parallel execution. In this case there might be variables distributed over all workers' heaps. A simple comparison between pointers cannot be used to determine whether a variable should be trailed or not, no matter how the heaps are arranged.

Our solution to this problem is to extend heap variables with a *timestamp*. This implies that the WAM has to be extended with a counter that is incremented whenever a choice point is created. This timestamp has to be saved in the choice point and restored on backtracking. A timestamp comparison is then used for determining whether to trail a variable or not.

## 2.5 PROCESS MANAGEMENT

Process management and scheduling are critical points in many of the parallel Prolog systems existing today.

In recursion-parallel systems much of the scheduling is done at compile time. It is then possible to determine which code is going to be executed in parallel. The number of processes executing the parallel code is determined immediately before parallel execution begins. These properties greatly simplify the process management problem; in fact, the scheduling overhead becomes negligible.

### Suspension

The set of programs that can be parallelized in Reform Prolog has been restricted to those which are binding-deterministic with respect to shared variables. This condition is verified at compile time.

Recursion levels suspend only to preserve sequential semantics and safeness. To ensure sequential semantics, a recursion level suspends before binding variables subject to time-sensitive tests; the level resumes when the variable is instantiated or it becomes leftmost. In both cases, preceding recursion levels cannot be affected by the binding.

To ensure safeness, a recursion level must not conditionally bind shared variables. In this situation, the level suspends until the variable is instantiated. If the level becomes leftmost, instantiation would occur and a safeness violation is signalled.

Some operations, notably general unification, unpredictably instantiate terms. Recursion levels suspend until leftmost before general unifications with terms containing time-sensitive variables (in the sense of above); if a general unification might create conditional bindings to shared variables, parallelization currently fails.

We implement these tests by busy-waiting. For instance, a worker suspending to preserve sequential semantics repeatedly checks if the variable has become instantiated or if the goal has become leftmost.

This method has the drawback that a suspended worker can tie up a processor for a long time. Whether this is a problem or not depends on the application program. If the average waiting time is short, then the overhead of this method is negligible.

The advantage of this method is that it does not slow down processes that do not wait for data—only the waiting worker is slowed down. No extra overhead is imposed on the Reform engine as compared to a sequential implementation.

### Scheduling

One of the advantages of the Reform execution model is that much of the scheduling can be done at compile time. It is possible to determine at compile time which code is going to be executed in parallel, and the number of recursion levels to run can be determined prior to parallel execution. In the Reform engine this is done by examining the recursion argument.

The scheduling process is thus reduced to dividing recursion levels among workers. There are two different approaches to scheduling: static scheduling and dynamic scheduling.

**Static scheduling.** Static scheduling minimizes the need for synchronization. Most parallel Prolog systems cannot use static scheduling since too little information is available about the structure of the parallel execution before it is initiated. In recursion-parallel systems, information about which code is going to be executed, as well as the number of parallel processes, is available. This makes it possible to distribute recursion levels to workers statically.

Of course, some programs are not well-suited for static scheduling, e.g., programs with poor load balancing due to significantly varying execution times of recursion levels.

**Dynamic scheduling.** The goal of dynamic scheduling methods is to optimize the trade-off between large process granularity and good load balance.

The simplest dynamic algorithm for scheduling is *self-scheduling* [187]. In this algorithm each processor executes one recursion level at a time until all levels have been executed. This method achieves almost perfect load balancing.

The problem with self-scheduling is, not surprisingly, that it tends to create too many processes with too fine granularity. However, this is less a problem in a recursion-parallel Prolog system than in loop-parallel Fortran systems, since the granularity of a single recursion level typically is greater than the granularity of a single iteration of a parallel loop.

More sophisticated dynamic algorithms has been proposed [118, 155, 130]. In these algorithms each processor is allocated a chunk of iterations at a time, instead of a single iteration. The chunk size may be fixed or variable. These algorithms have not yet been tested in Reform Prolog.

**Task switching.** Each worker is responsible for calculating which recursion levels it is going to execute. With static scheduling no synchronization is necessary to schedule work. With dynamic scheduling, on the other hand, it is necessary to synchronize accesses to a global variable holding the remaining number of unscheduled processes.

Task switching (i.e., starting a new process) amounts to a simple jump operation. With 'chunking' dynamic scheduling methods, the jump operation is preceded by an arithmetic calculation of how many processes to schedule in one chunk.

**Mapping processes to processors.** There are two simple ways to map recursion levels to processors regardless of whether we use static or dynamic scheduling (assuming chunks of more than one level in the latter case). Either consecutive recursion levels are mapped to consecutive processors (*horizontal* mapping), or consecutive recursion levels are mapped to the same processor (*vertical* mapping). If there are data dependencies in the program, then horizontal mapping is to be preferred since that enables process pipelining. If there are no data dependencies, on the other hand, then it might be better to use vertical mapping since it improves data locality.

## 2.6 COMPILING RECURSION PARALLEL PROGRAMS

Compilation has two main components: ensuring safeness and introduce suspension and locking unification instructions where required. This is managed by combining three analyses: type inference, safeness analysis and locality analysis.

The type inference domain is an extension of the Debray-Warren domain [62], with the addition of support for lists and difference lists. The compiler uses both parallel and sequential types in code generation; parallel types hold at all times in the program, while sequential types hold when leftmost.

Safeness analysis investigates when the computation is in a nondeterminate, parallel state. Determinacy changes on procedure entry and cuts. Note that there is no attempt to prove determinacy of a goal; instead the analyzer merely records when determinacy may change. To get reasonable results, the analyzer simulates indexing on procedure entry.

Locality analysis has two tasks: to find terms that are local to one recursion level, and to mark shared terms subject to time-sensitive operations (such terms are called *fragile* since they must be handled with care). Local operations do not require suspension or locking unification; shared terms require locking unification but no suspension, while fragile terms may not be instantiated out of the sequential order. If the compiler were not to respect fragility, the system might stray from simulating sequential behavior [33].

The goal of locality analysis is to generate precisely WAM code for parallel operations on unshared data. When this is possible, there is no parallelization overhead once the parallel execution has started. That is, the compiler attempts to localize the overheads of parallel execution to the points where the full machinery is actually needed.

## 2.7 INSTRUCTION SET

The sequential WAM instruction set is extended with new instructions for supporting recursion-parallelism.

## New instructions

The new instructions can be divided into five groups as follows.

**Creating shared structures.** These instructions create structures that might be accessed by other workers while they are created.

**Creating vectors.** These instructions build vector-lists that are used in the parallel phase. They are executed by the sequential worker. There are instructions that convert lists into vector-lists, as well as instructions that create vector-lists corresponding to binding environments.

**Accessing global arguments.** These instructions are used by the parallel workers to fetch data needed at their recursion levels from the sequential working space.

**Process control.** These instructions are used by the sequential worker to spawn parallel workers, and by parallel workers to switch from one recursion level to the next.

**Runtime safeness tests.** These instructions perform runtime tests to enforce safeness.

## Examples

The following programs illustrate the use of some of the new instructions.

In the description of each instruction we use the term *step*, by which we understand the number of elements the recursive argument is reduced by in each recursive call. Some registers are denoted Gn, this refer to the sequential workers register Xn, which is globally accessible to all workers during a parallel phase.

### Example 1:

```
map([],[]).
map([X|Xs],[Y|Ys]) :- foo(X,Y), map(Xs,Ys).
```

The program is compiled into the following extended WAM code.

```
map/2:  switch_on_term Lv La L1 fail

Lv:     try La
        trust L1

La:     % Sequential code for first clause of map/2

Lb:     % Sequential code for second clause of map/2

L1:     build_rec_poslist X0 X2 X3 X0  % first list
        build_poslist X1 X2 X4 X1      % second list
        start_left_body L2             % execute L2 in parallel
        execute map/2                  % call base case
L2:     initialize_left 1              % I := initial recursion level
L3:     spawn_left 1 X2 G2             % while (I++ < N) do X2 := I;
        put_nth_head G3 X2 0 X0        %  X0 := G3[I]
        put_nth_head G4 X2 0 X1        %  X1 := G4[I]
        call foo/2 0                   %  foo(G3[I],G4[I])
        jump L3                        % od
```

Let us describe the effects of the code for the recursive clause (label L1).

- `build_rec_poslist X0 X2 X3 X0` traverses the list in X0 and builds a vector-list of its elements, storing a pointer to it in X3, and storing the list length of X0 in X2. The last tail of the list is stored in X0.

- `build_poslist X1 X2 X4 X1` traverses the list X1 to its X2'th element, and builds a vector-list of its elements in X4. If the list has fewer than X2 elements and ends with an unbound variable, then it is filled out to length X2. The X2'th tail of the vector-list is unified with the X2'th tail of the list in X1, and finally X1 is set to the X2'th tail of the vector-list.

- `start_left_body L2` starts parallel execution of the foo/2 calls. The code at label L2 is run in parallel by all active workers. The sequential execution continues with the next instruction (execute map/2) when the parallel phase is finished.

- `initialize_left 1` initializes a worker for parallel execution. The step 1, given as argument, and the worker number is used for calculating the initial recursion level in static scheduling mode. In dynamic scheduling mode this instruction is ignored.

- `spawn_left 1 X2 G2` calculates the next recursion level for this worker and stores it in X2. The new level is calculated from the step 1 and the internal level count. If the new level is greater than the value stored in the global register G2 (i.e., the register X2 in the sequential worker), then then parallel computation is finished, otherwise the execution continues with the next instruction.

- `put_nth_head G3 X2 0 X0` performs the assignment X0 := G3[X2+0]. G3 points to a vector-list and X2+0 is the offset into the vector-list.
- `put_nth_head G4 X2 0 X1` similarly assigns X1 := G4[X2+0].

**Example 2:**

```
nrev([],[]).
nrev([X|Xs], Y) :- nrev(Xs,Z), append(Z,[X],Y).

append([],X,X).
append([X|Xs],Ys,[X|Zs]) :- append(Xs,Ys,Zs).
```

The program is compiled into the following extended WAM code.

```
nrev/2: switch_on_term fail La Ll fail  % fail cases never occur

 La:  get_nil X0
      get_nil X1
      proceed

 Ll:  allocate
      build_rec_poslist X0 X3 X6 X0  % first list
      build_variables X1 X3 X5 X1    % second list
      /* code for saving X3, X5 and X6 in environment */
      call nrev/2, 0                 % call base case
      /* code for restoring X3, X5 and X6 from environment */
      start_right_body X3 L1         % execute L1 in parallel
      deallocate
      proceed                        % done
 L1:  initialize_right 1 G3          % I := initial recursion level
 L2:  spawn_right 1 X7               % while(I-- > 0) do X7 := I;
      allocate
      put_nth_head G5 X7 1 X0        %  X0 := G5[I+1]
      put_list X1                    %  X1 := [
      unify_nth_head G6 X7 0         %         G6[I]
      unify_nil                      %              ]
      put_nth_head G5 X7 0 X2        %  X2 := G5[I]
      call append/3, 0               %  append(G5[I+1],[G6[I]],G6[I])
      deallocate
      jump L2                        % od

append/3:
      await_nonvar X0                % wait until first arg nonvar
```

```
     switch_on_term fail La L1 fail % fail cases never occur

La:  get_nil X0
     get_value X1 X2
     proceed

L1:  get_list X0                    % X0 = [
     unify_variable X3              %       X|
     unify_variable X0              %         Xs]
     lock_and_get_list X2 X4        % lock X2; X4 = [
     unify_x_value X3               %                X|
     unify_x_variable X5            %                  Zs]
     unlock X2 X4                   % unlock X2; X2 = [X|Zs]
     put_value X5 X2                % X2 := Zs
     execute append/3               % append(X0,X1,X2)
```

We describe below the effects of the new instructions in the above code.

- `build_variables X1 X3 X5 X1` builds a vector-list containing X3+1 distinct unbound variables, storing a pointer to it in X5. A reference to the last variable in the vector-list is stored in X1.
- `start_right_body X3 L1` initiates parallel execution of the append/3 calls in the body of nrev/2. The code at label L1 is run in parallel by all workers. The sequential execution continues with the following instruction (deallocate) when the parallel phase is finished. The length of the recursion list is given in X3.
- `initialize_right 1 G3` initializes a worker for parallel execution. The step 1, given in the first argument, the length of the recursion list, given in G3, and the worker number is used for calculating the initial recursion level in static scheduling. In dynamic scheduling mode this instruction is ignored.
- `spawn_right 1 X7` calculates the next recursion level for this worker, using the step given in the first argument, and stores it in X7. If all recursion levels have been executed, the worker suspends and awaits the next parallel phase.
- `unify_nth_head G6 X7 0` writes the element G6[X7+0] onto the heap. G6 contains a pointer to a vector-list and X7+0 is the offset into the vector-list. This instruction never occurs in read mode.
- `await_nonvar X0` suspends until X0 contains a nonvariable or the recursion level has become leftmost in the resolvent.
- `lock_and_get_list X2 X4` checks the value in X2. If X2 contains a variable, it creates a list on the heap, stores a pointer to it in X4, and

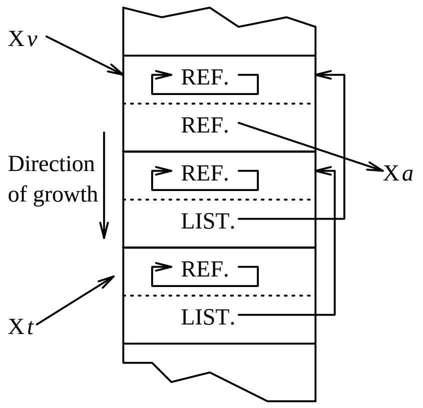


Figure 2.1: List (viewed as a vector) created by `build_neglist` .

enters write mode. If X2 contains a list, the S register is set. Otherwise failure occurs.

- `unlock X2 X4` is ignored in read mode. In write mode, X2 and X4 are unified.

**Other Instructions.** The extended instruction set contains some instructions not used in the two examples above. These instruction are described below.

We use the following notation. If $x$ is a cons cell of a vector-list, then $x.tl$ denotes the tail of the cons cell.

- `build_poslist_value Xa Xn Xv Xt` This instruction is to `build_poslist` as `unify_ value` is to `unify_variable`.
- `build_neglist Xa Xn Xv Xt` A 'reverse list' vector-list of length Xn is created and stored in Xv. Xt is set to the head and Xa to the last tail of the vector-list, respectively. The last tail of the vector-list is in this case also the tail of the first element of the vector-list. See figure 2.1.
- `build_neglist_value Xa Xn Xv Xw Xt` is to `build_neglist` as `unify_value` is to `unify_variable`.
- `put_global_arg Gn Xi` stores the value of the sequential workers register Xn in Xi.
- `put_nth_tail Gv Xl 0 Xi` Similar to `put_nth_ head` but performs the assignment Xi := Gv[Xl+0].tl.

- `unify_nth_tail Gv Xl O` Similar to `unify_nth_head` but writes Gv[Xl+0]. tl to the heap.
- `unify_global_arg Gg` writes the value of the sequential workers register Xg on the heap. This instruction never occurs in read mode.
- `await_leftmost` forces the current recursion level to suspend until it is leftmost in the resolvent, i.e., until all preceding recursion levels have terminated.
- `await_strictly_nonvar Xi` suspends the current recursion level until Xi contains a non-variable. If the recursion level becomes leftmost in the resolvent and Xi still contains an unbound variable, then a run-time error is signaled and execution fails.
- `await_variable Xi` suspends the current recursion level until it is leftmost in the resolvent. If the variable Xi becomes bound during suspension then a run-time error is signaled and the execution fails.
- `lock_and_get_structure F Xi Xn` Similar to `lock_and_get_list` but for structures with functor F.

## 2.8 EXPERIMENTAL RESULTS

In this section we present the results obtained when running some benchmark programs in Reform Prolog on a parallel machine.

### Experimental methodology

Reform Prolog has been implemented on the Sequent Symmetry multiprocessor. This is a bus-based, cache-coherent shared-memory machine using Intel 80386 processors. The experiments described here were conducted on a machine with 26 processors, where we used 24 processors (leaving two processors for operating systems activities).

The metric we use for evaluating parallelization is the speedup it yields. We present *relative* and *normalized* speedups.

Relative speedup expresses the ratio of execution time of the program (compiled with parallelization) on a single processor to the time using $p$ processors.

Normalized speedup expresses the ratio of execution time of the original sequential program (compiled without parallelization) on a single processor to the time using $p$ processors on the program compiled with parallelization.

**Benchmarks.**

**Programs and input data.** We have studied the performance of four benchmark programs. One of the programs exploits independent AND-parallelism and the others dependent AND-parallelism. Two programs are considerably larger than the others.

*Map.* This program applies a function to each element of a list producing a new list. In the measured program, the function simply decrements a counter 100 times. A list of 10,000 elements was used.

*Nrev.* This is the classic 'naive reverse' program run on a list of 900 elements.

*Match.* A dynamic programming algorithm for comparing, e.g., DNA-sequences. A sequence of length 32 was compared with 24 other sequences and the resulting similarity-values collected in a sorted binary tree.

*Tsp.* This program implements an approximation algorithm for the Traveling Salesman Problem. A tour of 45 cities was computed.

**Load balance.** One way of estimating the load balance of a computation is to measure the finishing time of the workers. We measured the execution time for each worker when executing our benchmarks. Static scheduling was used in all experiments.

*Map.* This program displayed a very uniform load balance (less than 0.3% difference between workers). This is hardly surprising since the number of recursion levels executed by each worker is large, and there is no difference in execution time between recursion levels.

*Nrev.* The execution time of each worker only varied about 3% when executing this program. There is a slight difference in the execution time of each recursion level but the large number of recursion levels executed by each worker evens out the differences.

*Match.* When 16 workers were used, 8 workers executed 2 recursion levels each, while 8 workers executed a single recursion level. This explains the relatively poor speedup on 16 workers. When 24 workers were used the execution time varied less than 0.3% between workers. This is explained by the fact that each worker executed one recursion level, and that all recursion levels executed in the same time.

*Tsp.* This program displayed an uniform load balance on all but three workers. This is explained by the fact that 45 recursion levels were executed in all; 21 workers executed 2 recursion levels each while 3 workers executed 1 recursion level each. Despite this the program displays good speedup (21.85). Using dynamic scheduling would not have improved the results in this case.

**Sequential fraction of runtime.** Parallelization occurs on a single level in Reform Prolog, and there are necessarily sequential startup portions of the programs. The startup portions of our benchmark programs include setting up the arguments for the parallel call, large head unifications, and spawning parallel processes.

According to Amdahl's law, the time spent in the sequential part of the program will ultimately limit the speedup from parallelization.

The following table shows for each benchmark program how large fraction of the execution time on a sequential machine is not subject to parallelization.

| Map | Nrev | String | Tsp |
|---|---|---|---|
| 0.3% | 0.04% | 0.003% | 0.005% |

The unparallelized parts of the programs are negligible. As a consequence, we do not currently see a need for parallelizing head unifications of parallel predicates.

## Results

The results of the experiments are summarized in the tables below. In the tables $P$ stands for number of workers, $T$ for runtime (in seconds), $S_R$ for relative speedup, and $S_N$ for normalized speedup. The sequential runtime for each program is given below each table.

| $P$ | $T$ | $S_R$ | $S_N$ | $T$ | $S_R$ | $S_N$ |
|---|---|---|---|---|---|---|
| 1 | 40.40 | 1.00 | 0.98 | 30.80 | 1.00 | 0.88 |
| 4 | 10.12 | 3.99 | 3.89 | 8.08 | 3.81 | 3.43 |
| 8 | 5.07 | 7.96 | 7.76 | 3.96 | 7.77 | 6.99 |
| 16 | 2.54 | 15.91 | 15.50 | 2.01 | 15.32 | 13.78 |
| 24 | 1.70 | 23.76 | 23.15 | 1.36 | 22.65 | 20.36 |

**Map**. (39.59 sec.) **Nrev**. (27.70 sec.)

| $P$ | $T$ | $S_R$ | $S_N$ | $T$ | $S_R$ | $S_N$ |
|---|---|---|---|---|---|---|
| 1 | 68.88 | 1.00 | 0.95 | 258.22 | 1.00 | 0.90 |
| 4 | 17.22 | 3.99 | 3.80 | 68.85 | 3.75 | 3.37 |
| 8 | 8.61 | 7.99 | 7.60 | 34.55 | 7.47 | 6.73 |
| 16 | 5.76 | 11.95 | 11.35 | 17.25 | 14.96 | 13.47 |
| 24 | 2.91 | 23.70 | 22.52 | 11.82 | 21.85 | 19.67 |

**Match**. (65.44 sec.) **Tsp**. (232.40 sec.)

### Discussion

We briefly compare our system with Andorra-I, another compiler-based implementation supporting deterministic dependent AND-parallelism [164]. Note, however, that Andorra-I parallelizes a wider class of computations than does Reform Prolog. In particular, Andorra-I also supports OR-parallelism. The results reported for Andorra-I were obtained on a 10 processor Sequent Symmetry.

As before, take the parallel efficiency of a program to be the speedup on $N$ processors divided by $N$. We have computed the efficiency relative to parallel Andorra-I, sequential Andorra-I and SICStus 2.1.

- Relative speedups on a set of 12 benchmarks range from 3.32 to 9.66, with a median of approximately 6.5 and a mean of approximately 6.4. The relative efficiency is taken to be 64%.
- Normalized speedups w.r.t. sequential Andorra-I on the same 12 benchmarks range from 2.04 to 7.26, with a median of 4.1 and a mean of 4.6. Normalized efficiency is then 46%.
- Compared with SICStus 2.1, Andorra-I was apparently 344 times faster in running the constraint solving fly_pan benchmark. Apart from this data point, on a range of 9 benchmarks, Andorra-I exhibited normalized speedups of 0.76 to 3.80 . The median was 1.47 and the mean 1.98. Normalized efficiency is thus 20%.

Only one benchmark can be directly compared to Reform Prolog, the nrv400 benchmark of naive reverse on 400 elements. The reported relative speedup is 8.25, the normalized speedup w.r.t. sequential Andorra-I is 6.55 and the normalized speedup w.r.t. SICStus 2.1 is 3.80. The efficiency is then 82%, 65% and 38%, respectively, on 10 processors. Reform Prolog has a relative efficiency of 95% and a normalized efficiency of 83% on 24 processors when running naive reverse on 900 elements.

We conclude that while Andorra-I can parallelize quite a wider range of programs, exploiting recursion-parallelism where available seems to have considerable benefits.

## 2.9 CONCLUSIONS

Reform Prolog implements parallelism across recursion levels by Reform compilation. One restriction is introduced on the recursive predicates subject for parallelization: bindings of variables shared between recursive calls of the predicate must be unconditional. This is not a severe restriction in practice.

The execution model has two major advantages:

First, a static process structure can be employed. That is, all parallel processes are created when the parallel computation is initiated. In most other systems for parallel logic programming, processes can be dynamically created, rescheduled and destroyed during the parallel computation.

A consequence of the static process structure is that process management and scheduling can be implemented very efficiently. This opens up for high parallel efficiency (91–99% on the programs tested). Another consequence is that it is easy for the programmer to see which parts of the program are going to execute in parallel. This facilitates the task of writing efficient parallel programs.

Second, it is possible by global dataflow analysis to optimize the code executed by each parallel worker very close to ordinary sequential WAM code. This results in very low overheads for parallelization (2–12 % on the programs tested).

The apparent drawback of this approach is that not all available parallelism in programs are exploited. This is, however, a deliberate design decision: exploiting as much parallelism as possible is likely to lead to poor machine utilization on conventional multiprocessors.

*Acknowledgments* We thank the Swedish Institute of Computer Science (SICS) for making their 26 processor Sequent Symmetry available to us, and the Andorra-I team of Bristol University for providing us with their latest benchmarks results.

# EXPLOITING RECURSION-PARALLELISM IN PROLOG

Johan Bevemyr, Thomas Lindgren, Håkan Millroth

In *Proc. PARLE'93*, Springer LNCS 694, 1993.

This paper received the Best Paper Award at PARLE'96
in competition with over 200 papers from
more than 20 different countries.

We exploit parallelism across recursion levels in a deterministic subset of Prolog. The implementation extends a conventional Prolog machine with support for data sharing and process management. Extensive global dataflow analysis is employed to facilitate parallelization. Promising performance figures, showing high parallel efficiency and low overhead for parallelization, have been obtained on a 24 processor shared-memory multiprocessor.

## 3.1 INTRODUCTION

The Single Program Multiple Data (SPMD) model of parallel computation has recently received a lot of attention (see e.g. the article by Bell [29]). The model is characterized by the feature of each parallel process running the same program but with different data.[1] The attraction of this model is that it does not require a dynamic network of parallel processes: this facilitates efficient implementation and makes the parallel control-flow comprehensible for the programmer.

[1]This should not be confused with the Single Instruction Multiple Data or SIMD model, where processes are synchronized instruction by instruction.

We are concerned here with the SPMD model in the context of logic programming. For recursive programs, the different recursive invocations of a predicate are all executed in parallel, while all other calls are executed sequentially. We refer to this variant of (dependent) AND-parallelism as *recursion-parallelism.* A recursive invocation minus the head unification and the (next) recursive call is referred to as a *recursion level.* Each recursion level constitutes a process, which gives the programmer an easy way of estimating the process granularity of a given program or call.

We implement recursion-parallelism by *Reform compilation* [138] (this can be viewed as an implementation technique for the Reform inference system [200]). This is a control-structure transformation that changes the control-flow of a recursive program quite dramatically. When invoking a recursive program with a call of size $n$ (corresponding to a recursion depth $n$) a four-phase computation is initiated:

1. A big head unification, corresponding to the $n$ small head unifications with normal control-flow, is performed.
2. All $n$ instances of the calls to the *left* of the recursive call are computed in parallel.
3. The program is called recursively. This call is known to match the base clause. Hence, in practice, this call is often trivially cheap.
4. All $n$ instances of the calls to the *right* of the recursive call are computed in parallel.

The difference between standard AND-parallelism and recursion-parallelism is illustrated in Figure 3.1. The figure shows execution of a recursive clause $H \leftarrow L, H', R$, where $H$ is the head, $H'$ is the recursive call and $L$, $R$ are (possibly empty) conjunctions. Note that the figure shows a situation where there are no data dependencies between recursion levels and no unification parallelism.

This paper is organized as follows. In Section 2 we define Reform Prolog. We discuss some programming techniques and concepts in Section 3. An overview of the parallel abstract machine is presented in Section 4. Section 5 provides an overview of the compile-time analyses employed for parallelization. Extensions to the sequential instruction set are presented by means of an example in Section 6. Experimental results obtained when running benchmark programs on a parallel machine are presented and discussed in Section 7.

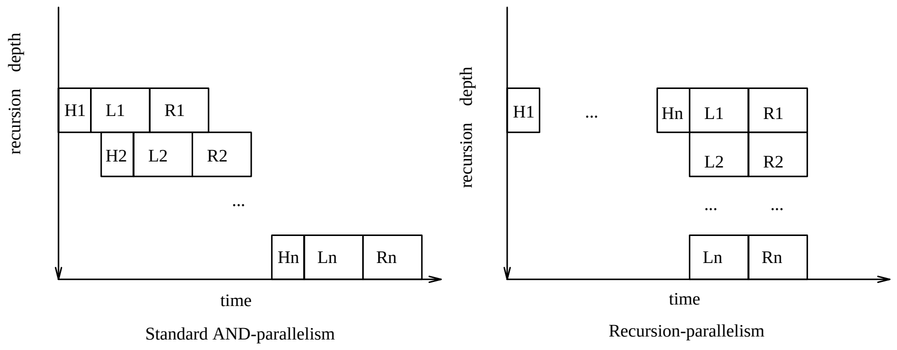


Figure 3.1: Executing a clause $H \leftarrow L, H', R$ with standard AND-parallelism and recursion-parallelism.

## 3.2 REFORM PROLOG

Reform Prolog parallelizes a deterministic subset of Prolog. This is similar to the approach taken in Parallel NU-Prolog [146]. However, Reform Prolog exploits recursion-parallelism when parallelizing this subset, whereas Parallel NU-Prolog exploits AND-parallelism.

With Reform Prolog, as with Parallel NU-Prolog, it is straightforward to call parallel subprograms from a nondeterministic program. Thus, there is a natural embedding of parallel code in ordinary sequential Prolog programs. Since the entire call tree below a parallel call is not parallelized, some nondeterministic computations can be supported in a parallel context as shown below. This is not done in Parallel NU-Prolog.

Below we define the condition for when a recursive Prolog predicate can be parallelized, and how this condition can be enforced by the implementation. We need to define two auxiliary concepts:

- A variable is *shared* if it is visible from more than one recursion level. Note that a variable can be shared at one point of time and unshared (local) at another.
- A variable binding is *unconditional* if it cannot be undone by backtracking.

We say that a call in a parallel computation is *safe* if all bindings made to its shared variables are unconditional when the call is executed. The condition for when a predicate can be parallelized is then:

> **A recursive predicate can be parallelized only if all calls made in the parallel computation are safe**.

Hence, limited nondeterminism is allowed: when conditional bindings are made only to variables local to a recursion level, the computation is safe.

Safeness of a call is defined w.r.t. to the *instantiation* of the call (i.e., what parts of the arguments are instantiated). We can distinguish between the parallel instantiation and the sequential instantiation of a call. These might differ as a parallel call can 'run ahead' of the sequential instantiation: recursion levels that would execute after the current one sequentially, may already have bound shared variables.

We say that a call is *par-safe* when it is safe w.r.t. the parallel instantiation, and that it is *seq-safe* when it is safe w.r.t. the sequential instantiation.

The compiler is responsible for checking that programs declared parallel by the programmer are safe. For calls that can be proven par-safe at compile-time, there is no need for extra safeness checking at runtime. For calls that can be proven seq-safe at compile-time, but not par-safe, it is necessary to check safeness at runtime. If the call is not safe, then it is delayed until it becomes safe. This is done by suspending the call until:

1. The unsafe argument has become sufficiently instantiated by another recursion level; or
2. The current call becomes leftmost.

If neither par-safeness nor seq-safeness can be proven at compile-time, then parallelization fails.

## 3.3 RECURSION PARALLEL PROGRAMMING

Before describing the execution machinery, we briefly consider some programming techniques and concepts.

### Machine utilization

The parallel programmer is concerned with utilizing the available parallel machine as efficiently as possible. In Reform Prolog, this means creating sufficient work and avoiding synchronization due to data dependences. A parallel machine where most of the workers are inactive due to lack of work is underutilized; a parallel machine where most workers are waiting for data is inefficiently used.

The number of processes available to the workers is precisely the number of recursion levels of the parallel call. To keep a large machine busy, there should consequently be many recursion levels – far more than the number of workers, ideally.

### Safeness

To illustrate the concept of safeness, consider the following two programs:

```
split([],_,[],[]).
split([X|Xs],N,[X|Ys],Zs) :- X =< N, split(Xs,N,Ys,Zs).
split([X|Xs],N,Ys,[X|Zs]) :- X > N, split(Xs,N,Ys,Zs).

split*([],_,[],[]).
split*([X|Xs],N,Ys,Zs) :- X =< N, !, Ys = [X|Ys0],
                          split*(Xs,N,Ys0,Zs).
split*([X|Xs],N,Ys,[X|Zs]) :- split*(Xs,N,Ys,Zs).
```

Assume that the third arguments of both predicates are shared, as might be the case if they were called from a parallel predicate. The predicate split/4 is then unsafe, since the third argument might be conditionally bound in the second clause, and then unbound again if the comparison $X \leq N$ fails. In contrast, the third argument of split∗/4 is only bound in a determinate state and so split∗/4 is safe for parallel execution (the binding of the fourth argument in the last clause does not affect safeness, since clauses are tried in textual order).

### Suspension

The programmer would like to avoid suspension as far as possible. However, in the implementation described in this paper, cheap suspension and simple, efficient scheduling combine to lessen synchronization penalties considerably. Consider the following program:

```
nrev([],[]).
nrev([X|Xs],Zs) :- nrev(Xs,Ys), append(Ys,[X],Zs).

append([],Ys,Ys).
append([X|Xs],Ys,[X|Zs]) :- append(Xs,Ys,Zs).
```

The compiler detects that the first argument of append/3 is shared. Hence, indexing must suspend until the first argument Ys is instantiated by the previous recursion level. The third argument Zs is also found to be shared, but the situation is reversed: the next recursion level will wait for Zs to be bound.

The inner loop of append/3 is then: wait for the input to be instantiated; when this occurs, write an element on the output list and go back to the beginning again. Thus, there is considerable scope for overlapping computations. As can be seen in the benchmark section, speedups are almost linear on 24 processors.

We also ran a second version of nrev/2, where the data dependence of append/3 was removed by a simple transformation: the length of the first argument is known at call-time by an extra parameter. Every call to append/3 can then construct the list Ys asynchronously (the elements of the list will be filled in later) and there is no suspension under execution.

```
nrev*(0,[],[]).
nrev*(N+1,[X|Xs],Zs) :- nrev*(N,Xs,Ys), append*(N,Ys,[X],Zs).

append*(0,[],Ys,Ys).
append*(N+1,[X|Xs],Ys,[X|Zs]) :- append*(N,Xs,Ys,Zs).
```

Surprisingly, the nrev∗/3 program is slower than the suspending nrev/2 program. Measurements show that this is because recursion levels of nrev/2 usually suspend very briefly, due to simple, fast suspension and straightforward scheduling of processes. For 16 processors, no processor was suspended more than 0.6% of the total execution time on the nrev/2 program; when run on 8 processors, the program suspended each worker less than 0.1% of the execution time.

On the other hand, the asynchronous nature of constructing the answer lists in nrev∗/3 led to an increase in the number of general unifications, due to later recursion levels overtaking earlier ones. The cost is time and memory. (Note that nrev∗/3 still exhibited a speedup of approximately 13 on 16 processors; nrev/2, however, was clocked at a speedup of over 15 on 16 processors.)

## 3.4 THE PARALLEL ABSTRACT MACHINE: OVERVIEW

The Reform engine consists of a set of *workers*, at least one per processor. Each worker is a separate process running a WAM-based [214] Prolog engine with extensions to support parallel execution. The Prolog engine is comparable in speed with SICStus Prolog (slightly faster on some programs, slightly slower on others).

The Reform engine alternates between two modes: sequential execution and parallel execution. One dedicated worker (the sequential worker) is responsible for the sequential execution phase. During this phase all other workers (the parallel workers) are idle. The sequential worker initiates parallel execution and resumes sequential execution when all parallel workers have terminated.

A common code area is used and all workers have restricted access to each others' heaps. All other data areas are private to each worker. The shared heaps are used to communicate data created during sequential and parallel execution (an atomic exchange operation is employed when binding possibly shared heap variables). When there are several shared heaps it is no longer possible to use a simple pointer comparison for determining whether a binding should be trailed or not. We solve this problem by extending the representation of each variable with a timestamp.

The sequential worker's temporary registers can be read by the parallel workers. These registers are employed for passing arguments to the parallel computation (one such register contains the precomputed recursion depth, i.e., the total number of parallel processes).

Synchronization is implemented by busy-waiting, i.e., suspended processes actively check if they can be resumed. The drawback of this method is that a suspended process ties up a processor. The advantage is that nonsuspended processes are not slowed down. In particular, very simplisitic and efficient approaches to process scheduling are possible. The Reform Prolog implementation currently supports *static scheduling* and *dynamic self-scheduling* [187]. With both approaches, the actual task switching amounts to a simple jump operation.

## 3.5 COMPILING RECURSION PARALLEL PROGRAMS

The compiler ensures the safeness of the computation, guarantees that time-dependent operations are performed correctly and employs suspending and locking unification when necessary.

These tasks depend on compile-time program analyses that uncover type, locality and determinacy information. The compiler then emits instructions based on this information, possibly falling back to more conservative code generation schemes when high-precision analysis results cannot be obtained.

### Type analysis

The type inference phase employs an abstract domain based on the standard mode-analysis domain [62], augmented with support for lists and difference lists as well as handling of certain aliases.

The compiler distinguishes the parallel and sequential types of a predicate. The sequential type is the type that must hold when the current recursion level is leftmost, while the parallel type holds for any recursion level. The parallel type is the most frequently used for compilation.

### Locality analysis

Locality analysis tries to find what terms are local to a process (recursion level), what terms are shared between processes and what terms contain variables subject to time-dependent tests. Consider the following program:

```
rp([],[]).
rp([X|Xs],[Y|Ys]) :- p(X,Y), rp(Xs).

p(a,X) :- q(X).                    p(b,c).

q(X) :- var(X),!,X = b.            q(c).
```

Given the call rp([a,b],[Y,Y]), we get two processes, p(a,Y) and p(b,Y). Both are safe and can thus proceed in parallel. Now assume p(b,Y) precedes p(a,Y), and binds Y to c. Then p(a,c) will reduce to q(c) which succeeds.

Sequential Prolog would have a quite different behaviour: first, p(a,Y) reduces to q(Y) and in turn to Y = b. Then p(b,b) fails. Hence, for this example, the parallel execution model is unsound w.r.t. sequential Prolog execution. This is due to the time-dependent behaviour of the primitive var/1.

Our solution to this problem is to mark, at compile-time, certain variables as being time-sensitive, or *fragile*. In the example, the argument of q/1 is fragile and the compiler must take this into account.

Furthermore, knowledge that a variable is *not* fragile, or *not* shared is extremely useful to the code generator. Using such information, operations with a very complex general case, such as unification, can in some cases be reduced to the same code as would be executed by a sequential WAM.

### Safeness analysis

Safeness analysis aims at ensuring that no conditional bindings are made to shared variables. In this respect, it is quite different from determinacy analysis: while determinacy analysis attempts to prove that a given call yields a single solution at most, safeness analysis instead proves no unifications with shared terms are made in a nondeterminate, parallel state. Safeness analysis thus employs the results of type inference (to see whether the call is determinate or not) and locality analysis (only shared terms can be unsafe).

## 3.6 INSTRUCTION SET

The sequential WAM instruction set is extended with new instructions for supporting recursion-parallelism. Due to space limitations, we can only describe these by means of a simple example and refer to other sources for a full discussion [34, 32, 125]. Consider the program:

```
map([],[]).
map([A|As],[B|Bs]) :- foo(A,B), map(As,Bs).
```

The program is compiled into the following extended WAM code.

```
map/2:  switch_on_term Lv La L1 fail

Lv:   try La
      trust L1

La:   get_nil X0
      get_nil X1
      proceed

L1:   build_rec_poslist X0 X2 X3 X0   % first list
      build_poslist X1 X2 X4 X1       % second list
      start_left_body L2              % execute L2 in parallel
      execute map/2                   % call base case

L2:   initialize_left 1               % I := initial recursion level
L3:   spawn_left 1 X2 G2              % X2 := I++; while(I < N) do
      put_nth_head G3 X2 0 X0         %  X0 := G3[I]
      put_nth_head G4 X2 0 X1         %  X1 := G4[I]
      call foo/2 0                    %  call foo(G3[I],G4[I])
      jump L3                         % next iteration
```

The instructions `build_rec_poslist` and `build_poslist` employ a data structure called a *vector-list*. This is a list where the cells are allocated in consecutive memory locations, to allow constant-time indexing into the list. The instructions work as follows:

- `build_rec_poslist X0 X2 X3 X0` traverses the list in X0 and builds a vector-list of its elements, storing a pointer to it in X3, and storing the vector length in X2. The last tail of the list is stored in X0.
- `build_poslist X1 X2 X4 X1` traverses the list X1 to its X2'th element, and builds a vector-list of its elements in X4. If the list has fewer than X2 elements and ends with an unbound variable, then it is filled out to length X2. The X2'th tail of the vector-list is unified with the X2'th tail of the list in X1, and finally X1 is set to the X2'th tail of the vector-list.

The sequential worker's registers X2, X3 and X4 are referred to as G2, G3 and G4 in the parallel code.

The instruction `initialize_left` calculates the initial recursion level in static scheduling mode. In dynamic scheduling mode, this instruction is ignored.

## 3.7 EXPERIMENTAL RESULTS

In this section we present the results obtained when running some benchmark programs in Reform Prolog on a parallel machine.

### Experimental methodology

**Parallel machine.** Reform Prolog has been implemented on the Sequent Symmetry multiprocessor. This is a bus-based, cache-coherent shared-memory machine using Intel 80386 processors. The experiments described here were conducted on a machine with 26 processors, where we used 24 processors (leaving two processors for operating systems activitites).

**Evaluation metrics.** The metric we use for evaluating parallelization is the speedup it yields. We present *relative* and *normalized* speedups.

Relative speedup expresses the ratio of execution time of the program (compiled with parallelization) on a single processor to the time using $p$ processors.

Normalized speedup expresses the ratio of execution time of the original sequential program (compiled without parallelization) on a single processor to the time using $p$ processors on the program compiled with parallelization.

### Benchmarks.

**Programs and input data.** We have parallelized four benchmark programs. Two programs (Match and Tsp) are considerably larger than the other two. One program (Map) exploits independent AND-parallelism, whereas the other three exploit dependent AND-parallelism.

*Map.* This program applies a function to each element of a list producing a new list. The function merely decrements a counter 100 times. A list of 10,000 elements was used.

*Nrev.* This program reverses a list using list concatenation ('naive reverse'). A list of 900 elements was used.

*Match.* This program employs a dynamic programming algorithm for comparing, e.g., DNA-sequences. One sequence of length 32 was compared with 24 other sequences. The resulting similarity-values are collected in a sorted binary tree.

*Tsp.* This program implements an approximation algorithm for the Travelling Salesman Problem. A tour of 45 cities was computed.

**Load balance.** One way of estimating the load balance of a computation is to measure the finishing time of the workers. We measured the execution time for each worker when executing our benchmarks. Static scheduling was used in all experiments.

*Map.* This program displayed a very uniform load balance (less than 0.3% difference between workers). This is hardly surprising since the number of recursion levels executed by each worker is large, and there is no difference in execution time between recursion levels.

*Nrev.* The execution time of each worker only varied about 3% when executing this program. There is a slight difference in the execution time of each recursion level but the large number of recursion levels executed by each worker evens out the differences.

*Match.* When 16 workers were used, 8 workers executed 2 recursion levels each, while 8 workers executed a single recursion level. This explains the relatively poor speedup on 16 workers. When 24 workers were used the execution time varied less than 0.3% between workers. This is explained by the fact that each worker executed one recursion level, and that all recursion levels executed in the same time.

*Match and Tsp.* These program displayed a uniform load balance on all but three workers. This is explained by the fact that 45 recursion levels were executed in all; 21 workers executed 2 recursion levels each while 3 workers executed 1 recursion level each. Despite this the program displays good speedup (21.85). Using dynamic scheduling would not have improved the results in this case.

**Sequential fraction of runtime.** Each parallelized benchmark program has an initial part which is not parallelized. This includes setting up the arguments for the parallel call. It also includes head unification and spawning of parallel processes.

According to Amdahl's law, the ratio of time spent in this sequential part of the program to that spent in the part which is parallelized (measured on a sequential machine) determines the theoretically best possible speedup from parallelization.

The following table shows for each benchmark program how large a fraction of the execution time on a sequential machine is not subject to parallelization.

| Map | Nrev | String | Tsp |
|---|---|---|---|
| 0.3% | 0.04% | 0.003% | 0.005% |

We conclude from this data that the unparallelized parts represent negligible fractions of the total execution times. Another conclusion is that there is no point in parallelizing the head unification of parallelized predicates, since it represents such a tiny fraction of the computation.

**Results**

The results of the experiments are summarized in the tables below. In the tables $P$ stands for number of workers, $T$ for runtime (in seconds), $S_R$ for relative speedup, and $S_N$ for normalized speedup. The sequential runtime for each program is given below each table.

| $P$ | $T$ | $S_R$ | $S_N$ | $T$ | $S_R$ | $S_N$ |
|---|---|---|---|---|---|---|
| 1 | 40.40 | 1.00 | 0.98 | 30.80 | 1.00 | 0.88 |
| 4 | 10.12 | 3.99 | 3.89 | 8.08 | 3.81 | 3.43 |
| 8 | 5.07 | 7.96 | 7.76 | 3.96 | 7.77 | 6.99 |
| 16 | 2.54 | 15.91 | 15.50 | 2.01 | 15.32 | 13.78 |
| 24 | 1.70 | 23.76 | 23.15 | 1.36 | 22.65 | 20.36 |

**Map**. (39.59 sec.) **Nrev**. (27.70 sec.)

| $P$ | $T$ | $S_R$ | $S_N$ | $T$ | $S_R$ | $S_N$ |
|---|---|---|---|---|---|---|
| 1 | 68.88 | 1.00 | 0.95 | 258.22 | 1.00 | 0.90 |
| 4 | 17.22 | 3.99 | 3.80 | 68.85 | 3.75 | 3.37 |
| 8 | 8.61 | 7.99 | 7.60 | 34.55 | 7.47 | 6.73 |
| 16 | 5.76 | 11.95 | 11.35 | 17.25 | 14.96 | 13.47 |
| 24 | 2.91 | 23.70 | 22.52 | 11.82 | 21.85 | 19.67 |

**Match**. (65.44 sec.) **Tsp**. (232.40 sec.)

**Discussion**

From the above results we calculate parallel overhead and efficiency of parallelization and make a comparison with other systems.

We compare Reform Prolog with the only other Prolog systems supporting deterministic dependent AND-parallelism that we are aware of, Andorra-I [164] and NUA-Prolog [151]. The Andorra-I system is an interpreter written in C. NUA-Prolog is a compiled system using a WAM-based abstract machine.

It should be noted that these systems to some extent exploit different forms of parallelism. Reform Prolog and NUA-Prolog exploit deterministic dependent AND-parallelism (recursion parallelism in the case of Reform Prolog). Andorra-I exploits deterministic dependent AND-parallelism and OR-parallelism (here we are only interested in the AND-parallel component of the system).

Unfortunately, we can make only a very limited comparison with NUA-Prolog, since the published benchmark programs stress the constraint-solving capabilites of the system, rather than its potential for raw AND-parallel speedup. However, we have compared their results on the nrev benchmarks with ours.

Let us define

$$\text{parallel efficiency} = (\text{speedup on } N \text{ processors})/N$$

The table below displays the parallel efficiency of Reform Prolog on 24 processors. It also indicates the parallelization overhead on a single processor as compared to the sequential Prolog implementation.

| Program | Relative efficiency | Normalized efficiency | Parallelization overhead |
|---|---:|---:|---:|
| Map | 99 % | 96 % | 2 % |
| Nrev | 95 % | 83 % | 12 % |
| Match | 99 % | 94 % | 5 % |
| Tsp | 91 % | 82 % | 10 % |

The Andorra-I system shows relative efficiency in the range of 47–83 % and normalized efficiency in the range of 35–61 % (assuming parallelization overhead of 35 %). We have excluded benchmarks that mainly exhibits OR-parallelism from this comparison. The figures are obtained on a Sequent Symmetry with 10 processors. NUA-Prolog shows a relative efficiency of 71 % and a normalized efficiency of 36 % on the nrev benchmark on an 11 processor Sequent Symmetry. (Note that the Reform Prolog figures were obtained using more than twice the number of processors the other systems used—using less processors improves the result.)

The Andorra-I system shows parallelization overheads in the range of 35–40 % on a set of benchmarks [164]. NUA-Prolog shows a parallelization overhead of 50 % on the nrev benchmark.

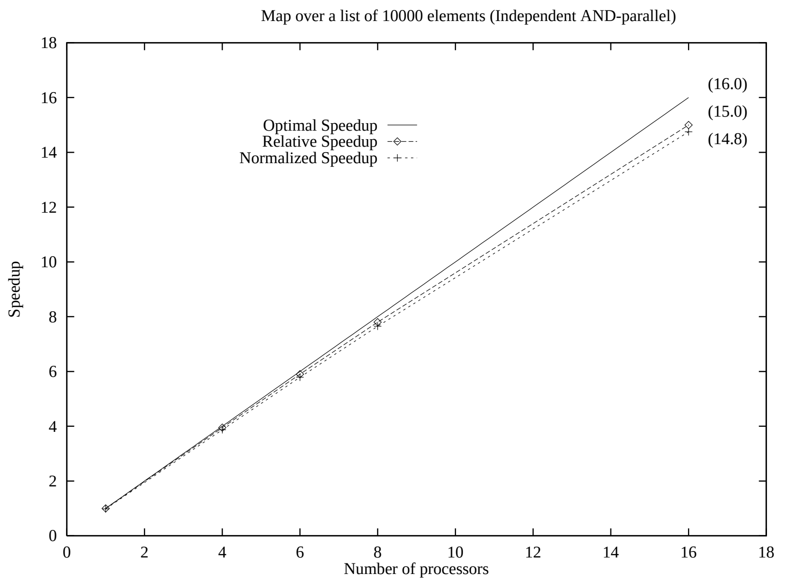
## 3.8 CONCLUSIONS

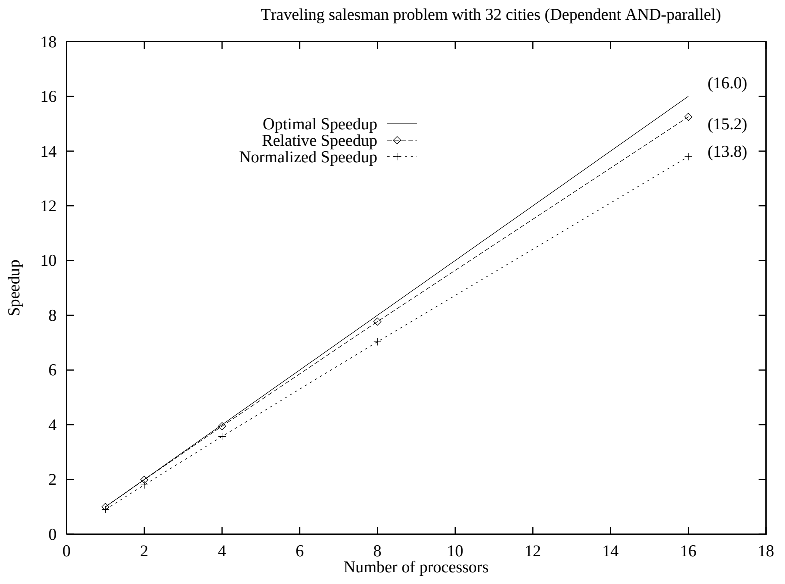
The developments and results discussed in this paper suggest that recursion-parallelism is an efficient method for executing Prolog programs on shared-memory multiprocessors. Our implementation exhibits very low overhead for parallelization (2–12 % on the programs tested).

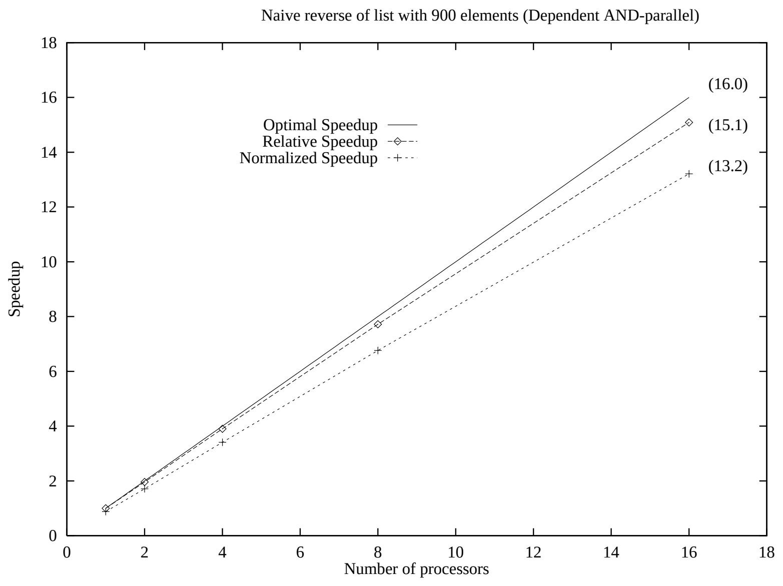
We believe that the high parallel efficiency of Reform Prolog is due to efficient process management and scheduling. An important factor is that all parallel processes can be initiated simultaneously. These properties of the system are due to the static recursion-parallel execution model made possible by Reform compilation.

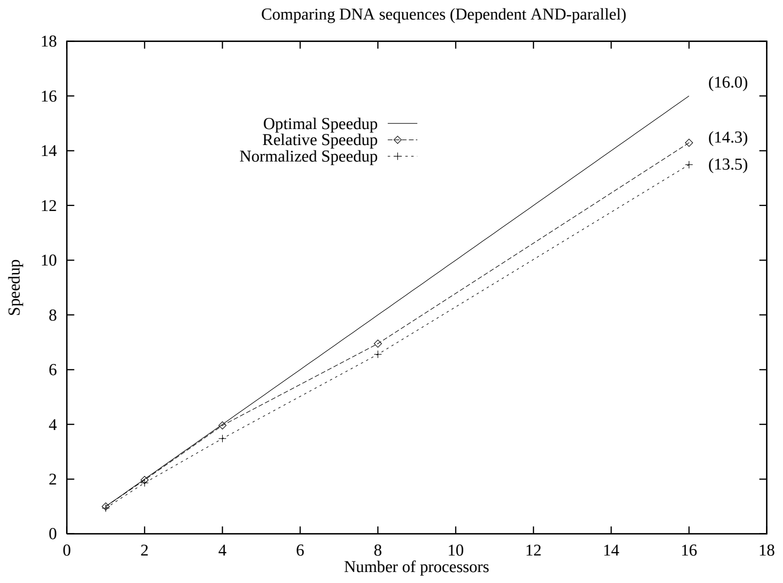
*Acknowledgments* We thank the Swedish Institute of Computer Science (SICS) for making their 26 processor Sequent Symmetry available to us.

## 3.9 APPENDIX: BENCHMARK PLOTS






Naive reverse of list with 900 elements (Dependent AND-parallel)
Optimal Speedup
Relative Speedup
Normalized Speedup
(16.0)
(15.1)
(13.2)
Speedup
Number of processors
0
2
4
6
8
10
12
14
16
18


Comparing DNA sequences (Dependent AND-parallel)
Optimal Speedup
Relative Speedup
Normalized Speedup
(16.0)
(14.3)
(13.5)
Speedup
Number of processors
0
2
4
6
8
10
12
14
16
18

# Compiler optimizations in Reform Prolog: experiments on the KSR-1 multiprocessor

Thomas Lindgren, Johan Bevemyr, Håkan Millroth

In *Proc. EURO-PAR'95 Parallel Processing*,
Springer LNCS 966, 1995.

We describe the compiler analyses of Reform Prolog and evaluate their effectiveness in eliminating suspension and locking on a range of benchmarks. The results of the analysis may also be used to extract non-strict independent and-parallelism.

We find that 90% of the predicate arguments are ground or local, and that 95% of the predicate arguments do not require suspension code. Hence, very few suspension operations need to be generated to maintain sequential semantics. The compiler can also remove unnecessary locking of local data by locking only updates to shared data; however, even though locking writes are reduced to 52% of the unoptimized number for our benchmark set, this has little effect on execution times. We find that the ineffectiveness of locking elimination is due to the relative rarity of locking writes, and the execution model of Reform Prolog, which results in few invalidations of shared cache lines when such writes occur.

The benchmarks are evaluated on a cache-coherent KSR-1 multiprocessor with physically distributed memory, using up to 48 processors. Speedups scale from previous results on smaller, bus-based multiprocessors, and previous low parallelization overheads are retained.

## 4.1 INTRODUCTION

An important challenge in parallel processing is to keep parallelization overhead low: it does not make sense to use up two or more processors just to break even with sequential code. Another challenge is to make sure that designs scale with the size of the machine. The design of the Reform Prolog system is an attempt to meet these challenges without making programming much harder than in the sequential case.

Reform Prolog has been implemented on a varity of shared address space multiprocessors. The parallel implementation is designed as extension of a sequential Prolog machine [32]. The implementation imposes very little overhead for process management, such as scheduling and load balancing.

We have previously described the compilation scheme [136, 138], execution model [33, 34], and parallel abstract machine [33, 34] of Reform Prolog. In this paper we describe a set of compiler analyses and optimizations for increasing available parallelism and reducing parallelization overheads. We discuss the effectiveness of the optimizations and the runtime space/time efficiency of the compiler. The system is evaluated on a cache-coherent large-scale multiprocessor with physically distributed memory.

## 4.2 REFORM PROLOG

The Reform Prolog system supports a parallel programming model where a single conceptual thread of control is mapped to multiple low-level threads. Each thread runs an instance of the same recursive program in an asynchronous parallel computation. This model is often called SPMD (Single Program Multiple Data). The programming model is realized by a compilation technique that translates a regular form of recursion to a parallelizable form of iteration [136, 138].

**Example.** The following program compares a sequence $B$ with a list of sequences. Each comparison, carried out by match/3, computes a similarity value $V$ that is stored in a sorted tree $T$ for later access. The tree is implemented as an incomplete data structure.

```
:- parallel match_seqs/3.
match_seqs([],_,_).
match_seqs([A|X],B,T) :-
    match(A,B,V),
    put_in_tree(T,V),
    match_seqs(X,B,T).
```

Assume that we invoke this program with a call containing an input list of four sequences. The programmer can then think of the recursive clause as being unfolded four times:

```
match_seqs([A1,A2,A3,A4|X],B,T) :-
    match(A1, B, V1), put_in_tree(T, V1),
    match(A2, B, V2), put_in_tree(T, V2),
    match(A3, B, V3), put_in_tree(T, V3),
    match(A4, B, V4), put_in_tree(T, V4),
    match_seqs(X,B,T).
```

Of course, this is not how Reform Prolog actually compiles the clause. However, the compiled code behaves *as if* the recursion is completely unfolded. When computing the call, four parallel processes are simultaneously spawned (the two calls within each process are executed sequentially):

$$\begin{array}{l}\mathrm{match}(A_1, B, V_1), \mathrm{put\_in\_tree}(T, V_1),\\ \mathrm{match}(A_2, B, V_2), \mathrm{put\_in\_tree}(T, V_2),\\ \mathrm{match}(A_3, B, V_3), \mathrm{put\_in\_tree}(T, V_3),\\ \mathrm{match}(A_4, B, V_4), \mathrm{put\_in\_tree}(T, V_4),\end{array}$$

The four calls to put_in_tree/2 are sequenced by synchronization on the shared variable $T$. However, the processes descend through the tree in parallel, temporally suspending when encountering not-yet-created subtrees. ■

The parallel execution model of Reform Prolog restricts the nondeterministic behaviour of parallel programs so that the following properties hold [33, 34]:

- *Parallel programs obey the sequential semantics of Prolog.* This implies that time-dependent operations (type tests, etc.) on shared, unbound variables are carried out only when leftmost is the sequential computation order.
- *Parallel programs do not conditionally bind shared variables.* This is similar to binding determinism as defined by Naish [146] in that shared variables can only be bound when the process is deterministic. However, in contrast to Naish's binding determinism, nondeterministic bindings to *local* variables are allowed.

In order to ensure these properties, the Reform Prolog compiler performs a global dataflow analysis and generates code that suspends processes and perform atomic updates only when necessary [125]. In particular, the compiler can generate precisely the code for a sequential Prolog machine when data are local.

## 4.3 COMPILER ANALYSES

The compiler analyses in the Reform Prolog compiler are based on *abstract interpretation* [58]. The abstract interpreter for Reform Prolog shares most of the characteristics of an abstract interpreter for sequential Prolog. This is natural, since each parallel process executes almost as a sequential Prolog machine.

We have modified Debray's dataflow algorithms [61, 63] for analysis of parallel recursive predicates. These algorithms compute call and success patterns for each procedure in the program. Call and success patterns describe the abstract values of the variables in a procedure call at procedure entry and exit, respectively.

The compiler carries out four different analyses using the same basic algorithm. The abstract domains of these analyses are described below.

### Types

The type domain is similar to that of Debray and Warren [65], extended to handle difference lists [125]. For our present concerns it suffices to note that the type analysis can discover ground and nonvariable terms. The analysis precision is similar to that of Aquarius Prolog [210].

### Aliasing and linearity

The analyzer derives possible and certain aliases by maintaining equivalence classes of possibly or certainly aliased variables. This is similar to the techniques used by Chang [47].

A term is *linear* if no variable occurs more than once in it. To improve aliasing information, the analyzer tracks whether terms are linear [109]. Three classes of linearity are distinguished: **linear**, **nonlinear**, and **indlist**. The latter denotes lists where elements do not share variables. The domain is thus:

$$\textbf{linear} \sqsubseteq \textbf{indlist} \sqsubseteq \textbf{nonlinear}$$

Consider a variable $X$ that is 'split' into several subterms $\{X_1, \ldots, X_n\}$ by unification $X = [X_1, \ldots, X_n]$ or some similar operation. Assuming that there are no other live aliases of $X$ in the rest of the clause, the analyzer uses linearity information as follows in this situation:

- If $X$ is **linear**, then $X_1, \ldots, X_n$ are also **linear** and unaliased.
- If $X$ is **indlist**, then $X_1, \ldots, X_n$ are **nonlinear** and unaliased.
- If $X$ is **nonlinear**, then $X_1, \ldots, X_n$ are **nonlinear** and aliased.

Our domain does not express covering properties, so the compiler cannot exploit linearity information when there are still aliases of $X$ alive in the current clause.

Linearity information is maintained for each equivalence class of aliases.

### Determinism

The analyzer determines where each process may create a choice point. The compiler needs this information since it must keep track of variables that may be bound within the scope of a choice point. The determinism domain is quite simple:

$$\mathbf{det} \sqsubseteq \mathbf{nondet}$$

As long as there is no possibility that a process may have created a choicepoint, it has status **det**. When a choicepoint may have been created, the status is changed to **nondet**. Cuts reset the determinism status to a value saved when the predicate is entered, similarly to what is done in the concrete implementation.

To improve precision, the compiler uses *abstract indexing* to approximate the first-argument indexing that will occur at runtime. This technique selects the possible paths for the inferred types of the first argument, based on standard WAM indexing [214].

### Locality

The analyzer maintains a hierarchy of data locality information:

- shared variables exposed to time-dependent operations by another process (clause indexing, arithmetic, type tests, etc.) are **fragile** and cannot be modified out of the sequential order;
- shared variables not subjected to time-dependent operations are **robust**;
- robust variables become **wbf** (will-be-fragile) when subjected to time-dependent operations—to subsequent processes, **wbf** variables will be seen as **fragile**;
- unshared data are **local**.

The locality domain is thus:

$$\mathbf{local} \sqsubseteq \mathbf{robust} \sqsubseteq \mathbf{wbf} \sqsubseteq \mathbf{fragile}$$

This locality domain can furthermore be used to detect non-strict independent and-parallelism [100]. As long as a process does not contain fragile data, it is independent of the results of other processes. Such independent programs can still construct shared data structures in parallel.

## 4.4 COMPILER OPTIMIZATIONS

The Reform Prolog compiler uses the information obtained by the analyzer for two purposes: verifying parallelizability and reducing parallelization overheads.

Our experience is that the compiler is able to filter out almost all unintended violations of the parallelizability conditions. Moreover, the analyzer can often help the programmer to identify code sections that cause unintended process suspensions.

Suspension and locking overheads can be reduced by optimizations that exploit the results of the compiler analyses.

### Cost of process suspension

Suspension overheads occur in two situations, based on the execution model: (a) to preserve the sequential semantics, fragile variables cannot be bound unless the process is leftmost and (b) to ensure binding determinism, shared variables must not be bound conditionally.

The compiler can eliminate suspension instructions when data is **local** or known to be instantiated, or the process is deterministic and data are non-**fragile**.

### Cost of locking unifications

Locking overheads occur when trying to bind possibly shared variables. Write accesses to shared variables must be locked to avoid data races and premature access to shared structures.

A locking write is required only when the heap cell being assigned is not **local**. In the WAM, writes (get/unify-instructions) are conditional depending on whether the accessed cell is bound or not. The compiler exploits type and locality information to strength-reduce locking instructions into non-locking ones where possible.

On the KSR-1, locking unification is done by simulating an atomic exchange operation. At present, this is done by locking the cache line of the variable to be bound, writing the variable and releasing the lock. A shared variable binding is then done as follows.

```
swap_x = Atomic_Exchange(x,y);
if (swap_x != x) {
    if (!unify(swap_x,y))
      Global_Failure();
}
```

Structures that may be bound to shared variables are constructed privately, and then atomically bound to the variable as described above.

If both suspension and locking overheads can be optimized away, then procedures called from a parallel predicate execute standard WAM code at full sequential speed.

## 4.5 EXPERIMENTAL SETUP

We have evaluated the system using three small (0.5 KB and 20-50 lines of code) and three medium-sized (3-12 KB and 100-425 lines of code) benchmark programs. The small benchmarks are:

| | |
|---|---|
| **map** | A function is mapped over a list of 5000 elements, producing a new list. The function simply decrements a counter 1000 times. |
| **nrev** | A list of 1300 elements is reversed using the 'naive reverse' program. |
| **tree** | 50,000 elements are inserted into a sorted, incomplete, binary tree. |

The medium-sized benchmarks are:

| | |
|---|---|
| **tsp** | The travelling salesman problem of 48 cities is solved by an approximation algorithm that visits the nodes of the minimal spanning tree. |
| **ga** | A population of 48 individuals is evolved for 5 generations using a genetic algorithm. The application is the travelling salesman problem with 120 cities. |
| **sm** | A string is compared to 96 other strings using the Smith-Waterman string matching algorithm (a standard dynamic programming algorithm often used for comparing, e.g., DNA sequences). Each string contains 32 characters. Each comparison results in a similarity value, which is stored in a sorted, incomplete, binary tree. |

The source code of the benchmark programs is available by ftp from ftp.csd. uu.se in the directory pub/reform/benchmarks.

The runtime statistics of the compiler were obtained on a Sun 630/MP with 4 40-MHz Sparc-2 processors and 64 MB of memory during a normal workday. The compiler was run in native code compiled SICStus Prolog version 2.1.6.

The performance measurements of the compiled parallel programs were obtained on a Kendell Square Research KSR-1 with 64 processors, each with 32 MB of memory. However, we were not able to allocate more than 48 processors due to external constraints.

## 4.6 COMPILER PERFORMANCE

The time required for analysis and total compilation time were (ms):

| | Analysis | Total | Ratio |
|---|---|---|---|
| map | 260 | 620 | 0.37 |
| nrev | 480 | 840 | 0.57 |
| tree | 850 | 1360 | 0.62 |
| tsp | 6149 | 8519 | 0.72 |
| ga | 7670 | 13870 | 0.55 |
| sw | 3850 | 5390 | 0.71 |

Thus, the analysis requires around 60-70% of the total compilation time; we consider this a reasonable overhead. Furthermore, the absolute compilation times (0.6 to 14 seconds) are quite reasonable, in particular when considering that the SUN 630/MP is not a particulary fast machine by today's standards. Aquarius Prolog seems to have similar absolute analysis times on similar hardware [81].

## 4.7 ANALYSIS RESULTS

We measured analysis results for arguments in procedures called from parallel predicates.

The following table shows the percentages of ground arguments and the locality information of non-ground arguments. The 'total' percentage is weighted with respect to the total number of predicate arguments in all benchmarks.

| | ground | local | robust | wbf | fragile |
|---|---|---|---|---|---|
| map | 60 | 40 | - | - | - |
| nrev | 25 | - | 12 | 12 | 51 |
| tree | – | 60 | - | - | 40 |
| tsp | 38 | 52 | 10 | - | - |
| ga | 35 | 59 | 2 | - | 4 |
| sm | 34 | 60 | - | - | 6 |
| total | 35 | 55 | 4 | 1 | 5 |

All benchmark programs are deterministic in the sense that they do not leave choicepoints when finished. Shallow backtracking does occur. The analyzer was able to verify these facts.

From these results we can see that:

- 90% of the arguments do not refer to data that require locking (i.e. the data are ground or local);
- 95% of the arguments do not refer to data that require suspension (i.e. the data are not fragile).

The 90% of arguments that do not require locking translates to a reduction to 52% of the original number of executed locking instructions.

## 4.8 PERFORMANCE OF COMPILED PROGRAMS

We measured execution times of sequential code and parallel code using up to 48 processors. The results (in seconds walltime) are:

| | seq | 1 | 2 | 6 | 12 | 24 | 48 |
|---|---|---|---|---|---|---|---|
| map | 81.6 | 81.8 | 40.9 | 13.8 | 6.98 | 3.60 | 1.86 |
| nrev | 18.7 | 22.6 | 12.5 | 4.11 | 2.20 | 1.13 | 0.82 |
| tree | 44.9 | 50.1 | 25.7 | 9.42 | 6.10 | 5.20 | 10.1 |
| tsp | 104 | 107 | 54.8 | 17.8 | 9.06 | 4.74 | 2.64 |
| ga | 77.0 | 82.0 | 40.9 | 13.9 | 7.68 | 4.24 | 3.40 |
| sm | 73.1 | 75.5 | 37.8 | 12.6 | 6.30 | 3.46 | 1.76 |

The speedups w.r.t. parallel code on one processor are:

| | 1 | 2 | 6 | 12 | 24 | 48 |
|---|---|---|---|---|---|---|
| map | 1 | 2.00 | 5.95 | 11.7 | 22.7 | 44.0 |
| nrev | 1 | 1.81 | 5.52 | 10.5 | 19.9 | 27.6 |
| tree | 1 | 1.95 | 5.31 | 8.20 | 9.63 | 4.97 |
| tsp | 1 | 1.95 | 6.00 | 11.8 | 22.6 | 40.5 |
| ga | 1 | 2.00 | 5.90 | 10.7 | 19.3 | 24.1 |
| sm | 1 | 2.00 | 6.00 | 12.0 | 21.8 | 42.9 |

The speedups w.r.t. sequential code are:

| | 1 | 2 | 6 | 12 | 24 | 48 |
|---|---|---|---|---|---|---|
| map | 1.00 | 1.99 | 5.93 | 11.7 | 22.7 | 43.9 |
| nrev | 0.83 | 1.50 | 4.55 | 8.66 | 16.6 | 22.8 |
| tree | 0.90 | 1.75 | 4.77 | 7.37 | 8.60 | 4.50 |
| tsp | 0.98 | 1.91 | 5.87 | 11.6 | 22.1 | 39.6 |
| ga | 0.94 | 1.88 | 5.53 | 10.0 | 18.2 | 22.6 |
| sm | 0.97 | 1.93 | 5.81 | 11.6 | 21.1 | 41.5 |

We see that the parallelization overhead is very low: 0–17%, with the larger benchmarks in the range of 2–6%.

The absolute speedup is very good on three benchmarks (map,tsp,sm), ordinary on two programs (nrev,ga) and bad on one program (tree). The three programs with ordinary or bad speedup all suffer from lack of exploitable parallelism: the nrev and tree programs are sequentialized by fragile variables, and the ga program invokes a sequential sort routine after the parallel computation (this limits the speedup according to Amdahl's law).

The key characteristics of the programs are:

| | |
|---|---|
| map | no suspension and no communication |
| nrev | heavy suspension and communication |
| tree | almost only suspension and communication |
| tsp | some communication and little suspension |
| ga | sequential sort after parallel computation |
| sm | some suspension and little communication |

## 4.9 EFFECTIVENESS OF OPTIMIZATIONS

To measure the effectiveness of optimizations (elimination of suspension and locking instructions) we compiled the benchmark programs without exploiting global analysis.

### Elimination of suspension instructions

To evaluate the effectiveness of removing suspension instructions, we rewrote the benchmarks to suspend whenever unbound heap cells were read by time-dependent operations, or written. Since there is no knowledge on what data is shared or fragile, this is required to retain sequential semantics.

The execution times without analysis were the following. We were unable to allocate 48 processors for this experiment and so show execution times for up to 24 processors.

| Program | 1 | 2 | 6 | 12 | 24 |
|---|---|---|---|---|---|
| map | 90.9 | 45.3 | 15.4 | 7.90 | 4.10 |
| nr | 30.5 | 34.4 | 37.0 | 42.3 | 51.3 |
| tree | 49.3 | 30.0 | 9.90 | 5.80 | 5.80 |
| tsp | 123.4 | 126 | 136 | 148 | 197 |
| ga | 74.5 | 37.7 | 14.1 | 7.70 | 6.10 |
| sm | 86.9 | 87.8 | 93.0 | 103 | 131 |

On 24 processors, there is comparable efficiency in the map, tree and ga benchmarks. For the map benchmark, the parallel computation consists of independent and trivial work that does not allocate heap data. The tree program can only write data when leftmost and so is essentially the same with or without analysis. The ga program performs all computations using scalar or ground data; all unbound variables are found on the stack. Since data on the stack cannot be shared, the engine can determine the outcome of suspension instructions even without analysis. The presence of extra suspension operations still has a cost: on 24 processors, the analyzed version of ga is 44% faster than the unanalyzed one.

Nrev, tsp and sm show net slowdowns when parallelized without global analysis. Tsp and sm perform considerable local work, which the no-analysis version delays until leftmost almost immediately and so sequentializes the program. For nrev without analysis, only the leftmost worker of nrev can write data at any time. With analysis, a 'pipeline' of processes appears and allows quicker completion.

Our conclusion is that there are substantial benefits to detecting computations local to a process (e.g., tsp, sm), and that more complex communication patterns (e.g., the nrev 'pipeline' effect) can be exploited transparently by the compiler.

### Elimination of locking instructions

The elimination of locking instructions is ineffective: there is no measurable difference in execution times when the number of locking unifications are reduced to 52% of the original number. Two factors contribute to this phenomenon:

First, locking instructions are infrequent even in unoptimized code. Locking is spatially infrequent, since assignments to heap variables is a small fraction of the total amount of data written in Prolog implementations [192, 79]. Locking is temporally infrequent, since our Prolog implementation is based on byte-code emulation of WAM [214] instructions. In unoptimized code, 2100–3700 machine instructions were executed for each locking operation on the three larger benchmarks.

Second, single locking instructions are, on average, fast due to cache organization: cache lines owned by a single processor do not need global invalidations. Shared cache lines are infrequently invalidated by our programs since shared variables are written at most once (bindings of shared variables cannot be undone by backtracking). This means that a shared cache line is invalidated at most $k$ times, where $k$ is the number of variables per line ($k = 4$ in our implementation on the KSR-1).

## 4.10 CONCLUSION

We have described compiler analyses that allow us to:

- verify that 95% of the predicate arguments do not refer to data that require suspension instructions;
- verify that 90% of the predicate arguments do not refer to data that require locking instructions.

The elimination of suspension instructions is very effective, since it makes otherwise quite sequential programs highly parallel. However, the elimination of locking instructions is ineffective, in that there is no measurable difference in execution times when locking instructions are removed. The most important contributing factor to this effect is that the single-assigment, non-backtrackable shared variables of Reform Prolog result in few invalidations of shared cache lines.

This cache behaviour is notable, since an increasingly important problem in parallel processing is to hide the latencies of physically distributed and comparatively slow memories. It is reasonable to expect that most future single adress-space architectures will have distributed memories and to expect a continuing increase in processor to memory speed ratio [153].

The performance measurements on the KSR-1 can be summarized as follows.

- Low parallelization overhead (0–17%, with the larger benchmarks in the range of 2–6%).
- Good absolute parallel efficiency on 48 processors (82–91%) provided that there is enough parallelism in the program.

Our data indicate that each process executes in a mostly sequential fashion: suspension and locking is rare. Hence, sequential compiler technology should be largely applicable to our system. We intend to employ such techniques to improve absolute execution speeds further. Future work also includes a detailed quantitative characterization of the memory system behaviour of Reform Prolog programs on different architectures.

**Acknowledgements.** We thank the University of Manchester for providing access to their KSR-1 computer.

## 4.11 ANALYSIS MEASUREMENT DATA

Predicates and total number of predicate arguments in benchmark programs.

| Program | Predicates | Total args |
|---|---|---|
| map | 3 | 5 |
| nrev | 3 | 6 |
| tree | 2 | 5 |
| tsp | 15 | 79 |
| ga | 19 | 128 |
| sm | 9 | 35 |

**Type domain.** The abstract domain used by the analyzer is that of Debray and Warren:

$$\bot \sqsubseteq \mathbf{gnd} \sqsubseteq \mathbf{nv} \sqsubseteq \mathbf{any}$$

$$\bot \sqsubseteq \mathbf{free} \sqsubseteq \mathbf{any}$$

with the addition of lists on the form

**list_{any,gnd,nv,free}_{free,nil,free+nil} nil**, **free+nil**

Using this expanded domain allows the analyzer to derive precise types for programs using difference lists. The analyzer uses list types to derive better information for list-recursive loop bodies, and can use nonvariable terms to eliminate some repeated suspension instructions (e.g., due to indexing on the first argument, followed by unification once the clause is selected).

**Information derived by analyzer.** We measured only predicates that execute during the parallel phases of the programs, "parallel predicates", since only they were affected by the optimizations described in the paper.

| | gnd | nv | free | list_*_n | list_*_f | any |
|---|---|---|---|---|---|---|
| map | 3 | - | 2 | - | - | - |
| nrev | 1 | - | 2 | 1 | 2 | - |
| tree | - | - | - | - | - | 5 |
| tsp | 27 | 2 | 18 | 5 | 2 | 25 |
| ga | 36 | - | 47 | 16 | 6 | 23 |
| sm | 10 | 7 | 5 | 3 | - | 10 |

Locality information derived by analyzer for parallel predicates.

| | pred | args | gnd | local | robust | wbf | fragile |
|---|---|---|---|---|---|---|---|
| map | 2 | 4 | 2 | 2 | - | - | - |
| nrev | 2 | 6 | 2 | - | 1 | 1 | 2 |
| tree | 2 | 5 | - | 3 | - | - | 2 |
| tsp | 15 | 79 | 31 | 40 | 8 | - | - |
| ga | 19 | 128 | 45 | 75 | 2 | - | 6 |
| sm | 9 | 35 | 12 | 21 | - | - | 2 |

# A Scheme for Executing Nested Recursion Parallelism

Johan Bevemyr

In *Workshop on Parallelism and Implementation Technology for (Constraint) Logic Programming Languages*, 1996.

We propose an extension for nested parallelism in the Reform Prolog execution model. The proposed modification preserves the simplicity and efficiency of the original design and its performance model.

This is achieved by exploiting properties of deterministic execution and careful scheduling of work.

## 5.1 INTRODUCTION

We propose a scheme for executing nested recursion parallelism. The scheme requires only minimal extensions to the current execution model of Reform Prolog [34].

It is possible to transform some nested recursions into a single recursive loop, as shown by Millroth [135]. Flattening nested data-parallel computations has been investigated by Blelloch [37], Palmer, Prins and Westfold [152] among others.

However, it is not always feasible to flatten a nested recursion in Reform Prolog, e.g. when the size of the nested recursion cannot be statically determined. Suppose we have the following program:

```
p([],[]).
p([H|T],[NewH|NewT]) :-
        state_size(H,State,N),
        q(N,State,NewH),
        p(T,NewT).

q(0,_,Result) :- !, Result = [].
q(N,State,[H|T]) :-
        foo(N,State,H),
        N2 is N - 1,
        q(N2,State,T).
```

Then we would have to choose between parallelising the outer loop (p/2) or the inner loop (q/2). p/2 is the obvious choice provided that not both p/2 and q/3 are shallow recursions. It would then be desirable to parallelise both and distribute the work among the workers.

We investigate how this nesting of parallel loops can be supported while maintaining the efficient implementation and the simple execution model of Reform Prolog.

Note that the intention of Reform Prolog is not to parallelise arbitrary Prolog programs. The programmer is responsible for rewriting the programs into a form suitable for parallel execution. Reform Prolog provides a clean execution and performance model which makes it possible for the programmer to coherently express parallelism with predictable performance.

## 5.2 REFORM PROLOG: THE ORIGINAL DESIGN

Reform Prolog concentrates on parallelising structure recursion, e.g. recursion over lists or integers, in a manner similar to loop parallelisation in imperative languages. This gives the programmer a simple and predictable execution model.

The execution of a recursive loop can be seen as a four step computation (or as program transformation through unfolding).

1. Perform all initial head unifications, building the necessary vectors for parts 2 and 4.
2. Execute the goals to the left of the recursive call in a parallel loop.
3. Execute the base case.
4. Execute the goals to the right of the recursive call in a parallel loop.

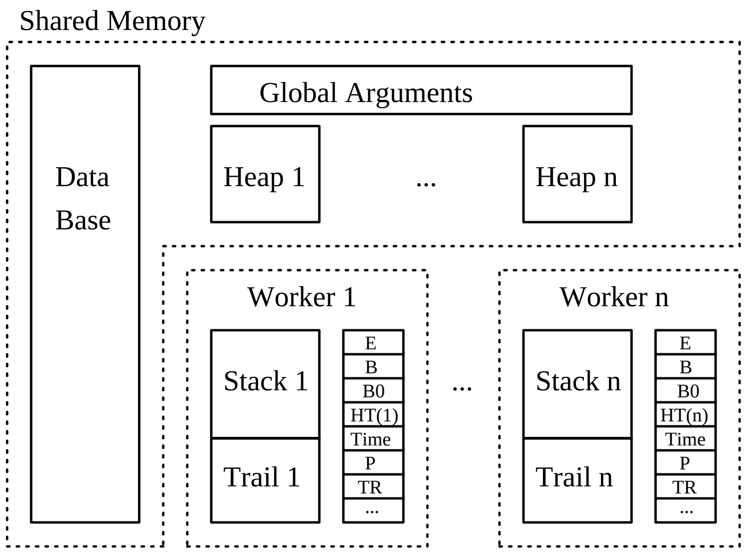


Figure 5.1: Overview of the recursion parallel machine.

This model is efficiently implemented using a set of WAM [214] engines (workers) which compute one recursion level at a time (see Figure 5.1). Instructions are added to WAM for performing the vectorising unifications in step 1 and for synchronisation in step 2 and 4 to keep the sequential semantics of the program.

No nesting of parallel recursive calls is allowed. Data sharing and communication is allowed as long as bindings of shared variables are deterministic. Scheduling, memory and process management are simple and efficient. Almost sequential code is executed by each worker, with the exception of some synchronisation instructions (which are only present in programs with data dependencies between recursion levels). Bevemyr [32] describes the full machinery in detail.

All this result in small overheads for parallel execution.

## 5.3 NESTED RECURSION PARALLELISM

How can this machinery be extended to support nested parallel execution?

We make a few observations about the current implementation:

1. The stack is restored to its initial state when a recursion level terminates, i.e. no choice points or environments are left by a recursion level.

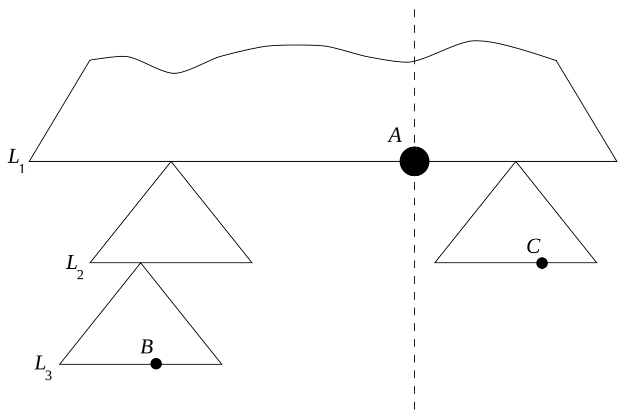


Figure 5.2: $L_1$, $L_2$ and $L_3$ represent parallel recursive loops where $L_2$ is initiated by an iteration level in $L_1$ and $L_3$ is run inside an iteration level in $L_2$. $A$ is a suspended recursion level in $L_1$. $B$ and $C$ represent available work.

2. The trail and the heap grow when executing forward and shrink on backtracking.
3. Work to the left is less speculative than work to the right in the tree (resolvent).

Let us consider the following situation in the nested case: a worker $w$ suspends at point $A$ waiting for some data to be produced or to become leftmost (see Figure 5.2). Work is available both to the left ($B$) and the right ($C$) in the tree (resolvent).

If $B$ is chosen then we can execute $B$ until it terminates. This means that the stack will be restored to the state it was in when $A$ suspended. However, the heap and the trail will not be restored. Portions of data and trail entries will be left on the heap and the trail after executing $B$, as shown in Figure 5.3. First there is a portion produced when executing $A$, then a portion belonging to $B$ followed by a section produced when continuing to execute $A$.

Consider what happens if the entire parallel recursion $L_3$ fails. Then the trail section belonging to $B$ must be unwound. This is tricky since it resides between sections produced by $A$. A solution is to only allow $L_3$ to be executed in parallel if the initiating worker is in a deterministic state, i.e. no choice point has been created in the current thread since the parallel execution started. If $L_3$ fails, then the entire parallel execution fails ($L_1, L_2$

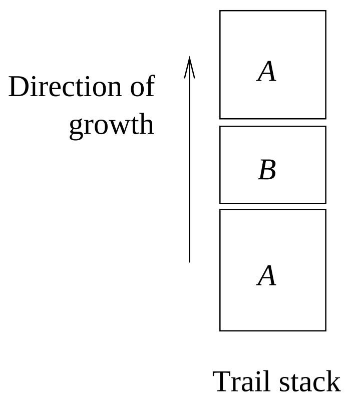


Figure 5.3: Trail entries are first created while executing $A$ which suspends and $B$ is executed, leaving new trail entries. $A$ is then resumed creating another section of trail entries.

and $L_3$) and all trail entries created during the current parallel phase are undone.

A similar situation occurs if $A$ fails back to a point before $w$ executed $B$. The trail entries created while running $B$ are unwound since they are in the middle of the trail stack. This situation can be treated in the same manner as above. A worker is only allowed to search for work when it is suspended in a deterministic state. If the worker fails to a point before the suspension, then the entire parallel phase fails and all conditional bindings are undone.

If $C$ is chosen then we might end up in a situation where $C$ must be suspended and $A$ resumed, since $C$ may wait for data produced by $A$. That is, we must be able to switch between suspended recursion levels which is not a trivial task.

To summarise: no new data areas or data structures are needed to support nested parallel execution if the following three restrictions are used.

1. Nested recursion parallelism is only allowed if no choice points have been created in the current thread of recursion levels.
2. Suspended workers are only allowed to search for work if they are in a deterministic state, i.e. no choice points have been created by the worker in the current iteration level.
3. Suspended workers are only allowed to search for work to their left in the tree.

The first restriction can be verified at compile time using information obtained by the existing compiler [126].

The drawback of this approach is that some nested parallelism might not be exploited. On the other hand, the machinery for nested parallelism is extremely simple and no extra overheads are associated with its implementation.

### Optimisations

The compiler generates suspension instructions to enforce the sequential semantics for producer-consumer dependencies. A worker is suspended until a variable becomes sufficiently instantiated, or until the current recursion level becomes leftmost. The leftmost condition can sometimes be reduced to waiting until the current level is locally leftmost. That is, until it is the leftmost unfinished recursion level in the current recursive loop. This is possible when a variable is locally shared and not shared between recursion levels at a higher level.

Lindgren [126] shows how specialised versions of the suspension instructions can be generated by the compiler.

## 5.4 DISCUSSION

### Restrictions

Are the restrictions posed on nested recursion parallelism reasonable?

Let us consider the first restriction. Nested parallel recursive loops are only allowed to be initiated in deterministic states. We do not think this is a severe restriction. The condition is intuitive and programmers would have no problem determining when it is satisfied.

This would allow the two loops in the program on page 75 to be nested provided that state_size/3 and foo/3 are deterministic. Whereas programs such as testing, in parallel, if each element in a list satisfies some condition are not allowed inside an outer parallel loop.

The second restriction states that a worker is only allowed to search for work if it is idle, or suspended, in a deterministic state. This restriction is harder to reason about for the programmer. It is not as obvious when and how a worker suspends. However, removing this restriction requires some form of "cactus" heaps and trails which is less efficient than standard stacks.

The third condition is a restriction on the scheduler. It means that some workers will be unable to obtain parallel work since they are almost leftmost (the leftmost worker is never suspended).

It would be possible to allow workers to obtain work from the right, but such work might have to be suspended in favour of work to the left. The machinery for this is cumbersome and it is questionable whether the gain is worth the added complexity.

### Related Work

Hermenegildo [95, 96, 100] and others have noted the significance of left-biased scheduling mainly to ensure the no-slowdown criteria, meaning that less speculative work is performed first. Combined with determinism, we argue that parallel implementation overheads can be made minimal.

Our scheme can be viewed as merging deterministic parallel subgoals with the topmost parallel computation. This is similar to AKL [108, 107, 140] where deterministic AND-boxes are promoted. Gupta, Pontelli [156, 158] and Tang [157] use a similar technique in &ACE for reducing the overheads for independent AND-parallelism.

Carro, Gupta, Hermenegildo, Pontelli, Santos Costa, Tang [95, 94, 99, 100, 159, 157, 87] and others have proposed different schemes for reducing the overheads for nested independent AND-parallelism. These schemes reduce the performance difference between various independent AND-parallel systems and Reform Prolog, at the cost of added implementation complexity.

Our restriction on nested parallelism greatly simplifies the design compared with other nested AND-parallel systems like &-Prolog [95, 94, 99], DASWAM [178, 176] and ACE [85].

- No distributed stacks with markers [40, 95, 177] or multiple stack sets [94] are needed.
- No intelligent backtracking, or any other non-standard sequential backtracking scheme is needed.
- Very small overheads for suspension are possible.
- Busy wait can be used without deadlock detection machinery.
- Large number of processes can be created simultaneously.

Admittedly other designs may be able to exploit more implicit parallelism. Our design relies on programmers to rewrite their programs to fit our execution model.

Hermenegildo and Carro [98] observe that &-Prolog can be extended with support for data-parallel execution, similar to Reform Prolog.

There are some significant differences.

1. Reform Prolog is a (somewhat restricted) dependent AND-parallel system (not as general as DASWAM [176] which, however, imposes some significant performance penalties). It is possible to have producer-consumer dependencies in parallel loops.

   Data dependencies are carefully tracked by the compiler and necessary synchronisation primitives are generated.

   It is not clear how difficult it would be to incorporate these compilation techniques into &-Prolog.

2. We argue that the performance and execution models for Reform Prolog are clearer than for implicitly AND-parallel systems.

Possible future work include investigating to which extent the results for scheduling data-parallelism in imperative languages, e.g. Blelloch, Gibbons and Matias [39], can be applied in Reform Prolog.

## 5.5 CONCLUSION

This paper aims to show that there is no need for a complicated implementation technique to efficiently take advantage of nested data parallelism, and consequently a substantial part of nested *dependent* AND-parallelism using Lindgren's transformation [126].

We have proposed simple extensions to the execution model of Reform Prolog which make it possible to efficiently exploit nested recursion parallelism.

Initiating parallel execution, and obtaining work when suspended, is only allowed in deterministic states. Nested work is only allowed to be obtained from the left of the suspension point in the resolvent.

These restrictions make it possible to efficiently exploit nested parallelism with exceptionally small overheads in term of execution time and implementational complexity.

It is reasonable to assume that our approach will be easy to adopt in high performance implementations due to its streamlined design and the minimal changes to the sequential execution model.

## 5.6 ACKNOWLEDGEMENTS

Thanks to Thomas Lindgren and Håkan Millroth for valuable discussions, and to Manuel Hermenegildo and Enrico Pontelli for pointers to related research.

# Executing Bounded Quantifications on Shared Memory Multiprocessors

Jonas Barklund, Johan Bevemyr

In *Proc. PLILP'93*, Springer LNCS 714, 1993.

We present a schema for executing bounded quantifications on shared memory multiprocessors, using the WAM-based abstract machine designed for recursion parallel execution developed by Bevemyr, Lindgren and Millroth. Preliminary results indicate almost linear speed-up on 24 processors for some quantified expressions. The parallelization overhead is low, in the order of 10-15%.

A best speed-up of 34.38 for a bounded quantification program compared with sequential execution of a corresponding recursive program, and 19.18 compared with sequential execution of the bounded quantification, was achieved using 24 processors.

## 6.1 INTRODUCTION

The data parallel model for parallel computation offers high efficiency for an interesting class of computations. So far, this model has been exploited almost exclusively through programming languages belonging to the imperative paradigm. In order to program computers supporting this model in a logic programming language it is necessary to treat the language's *repetition constructs* as concurrent. Horn clause programs have recursion as their sole means for repetition, but a more general logic programming language may also employ quantified expressions for this purpose. Bounded quantification, a limited class of quantified expressions, have been proposed as a concurrent linguistic construct for logic programs [26].

We have previously shown how bounded quantifications on a sequential computer can be given operational semantics in terms of definite iterations [20], and sketched how they can be run on a Connection Machine, a SIMD multiprocessor that directly supports data parallel computation [11]. In this paper it is shown how we can exploit the affinity between bounded quantification and recursive programs to run bounded quantifications on an existing implemented abstract machine that was developed to run a class of recursive programs efficiently on shared-memory MIMD multiprocessors [34].

In order to do so, we shall describe a quite straightforward transformation from bounded quantifications to recursive programs. The resulting recursive program can be executed efficiently using the abstract machine developed for recursion parallel execution [32].

Note, however, that the parallel implementation cannot take full advantage of the simplicity of bounded quantifications when only the transformed programs are available. Some implementation methods needed for optimal execution of bounded quantifications may not be reasonable for arbitrary recursive programs. Hence, the concurrency in programs may not be detected and some programs may not be run in the most efficient way. In the future it therefore seems worthwhile to investigate methods that are applied directly to bounded quantifications. Until such methods are available, the methods described herein are appropriate.

In this paper we propose to use instructions already present in the Reform Prolog abstract machine for creating and executing similar processes on a set of processors, in a data-parallel fashion. Such instructions can be added to a more general AND-parallel implementation using other synchronization and process management routines. However, such a system may not exploit the data-parallel style of execution as efficiently as our specialized implementation.

Due to lack of space in this paper we do not otherwise compare our implementation with other AND-parallel implementations. See Bevemyr's exposition of the Reform Prolog Engine [32] for a more detailed comparison.

## 6.2 BOUNDED QUANTIFICATIONS

In its most general form a *quantification* is an expression that applies an abstraction (its body) to a set of values, combining the results in a way that depends on the quantifier. A *bounded quantification* is a quantification where the set of values is finite. Voronkov has shown how a Horn clause language with SLD-resolution can be extended to incorporate a form of bounded quantifications with an inference procedure, SLDB-resolution, that handles the whole extended language [211, 212].

In the context of logic programming, the most obvious quantifiers are the logical universal and existential quantifiers of first-order predicate calculus that combine the values of the body by conjunction and disjunction, respectively. However, we can also think of other quantifiers. In mathematics, sums and products are readily thought of a quantifiers, combining the values of the body by addition and multiplication, respectively. There are several other possible quantifiers from mathematics that combine the values of the body through reduction with some binary operator (note that universal and existential quantification can also be thought of in this way).

In this paper we shall consider only the universal quantifier and a class of arithmetic quantifiers, exemplified here with the sum, product, minimum and maximum quantifiers. We will also only consider a small number of notions for expressing the finite sets: an integer range [26], the elements of a list, and the suffixes of a list [211, 212].

## 6.3 PROLOG WITH BOUNDED QUANTIFICATIONS

We will use an abstract syntax for bounded quantifications in Prolog that is slightly different from the syntax we have used before [11, 26]; an actual concrete syntax will probably have to be a compromise between several objectives.

- We will write a universal quantification as `all(`$\rho[i]$`,` $\beta[i]$`)` where $\rho[i]$ is a *range expression* and the body $\beta[i]$ is a goal. The expression $i$ must be a variable, called the *iteration variable*, that is considered locally scoped in the quantification. A range expression is one of the following.
  - An integer range expression $m$≤$i$<$n$, where $m$ and $n$ are integer expressions; the iteration variable $i$ ranges over the integers $m$, $m + 1, \ldots, n - 1$.[1]
  - A list membership expression $i$ `in` $\ell$, where $\ell$ is a list `[`$v_1$`,`$v_2$`,...,` $v_k$`]`; the iteration variable $i$ ranges over $v_1, v_2, \ldots, v_k$.
  - A list suffix expression $i$ `suffix_of` $\ell$, where $\ell$ is a list `[`$v_1$`,`$v_2$`,` `...,`$v_k$`]`; the iteration variable $i$ ranges over the lists `[`$v_1$`,`$v_2$`,...,` $v_k$`]`, `[`$v_2$`,...,`$v_k$`]`, ..., `[`$v_k$`]`, `[]`.

[1]Expressions on the form $m$≤$i$≤$n$, $m$<$i$<$n$ or $m$<$i$≤$n$ can, of course, be trivially rewritten to the form $m$≤$i$<$n$.

The universal quantification succeeds if $\beta[v_1]$, $\beta[v_2]$, ..., $\beta[v_k]$ all succeed (with the same instantiation of any global variables), where $v_1$, $v_2$, ..., $v_k$ are the values of $i$ satisfying $\rho[i]$. Just as for Prolog's `setof` construct, the body may be prefixed with $i_1$`^`$i_2$`^`...$i_p$`^`, $p \geq 0$, denoting that the variables $i_1$, $i_2$, ..., $i_p$ are locally existentially quantified.

- We will write an arithmetic quantification as $\kappa$`(`$\rho[i]$`, `$\beta[i]$`, `$\tau$`)`, where $\kappa$ is an arithmetic quantifier such as `sum`, `prod`, `min` or `max`, the range expression $\rho[i]$ is as above, the body $\beta[i]$ is an arithmetic expression, and $\tau$ is a variable or a number. The arithmetic quantification succeeds if the result of reducing $\beta[v_1]$, $\beta[v_2]$, ..., $\beta[v_k]$ with the appropriate binary operator equals the value of $\tau$ (possibly instantiating $\tau$).

Some examples of bounded quantifications are

- `all(X in Z, (p(X), q(X,V)))`, which is equivalent to `p(`$v_1$`),q(`$v_1$`, V),p(`$v_2$`),q(`$v_2$`,V),...,p(`$v_k$`),q(`$v_k$`,V)`, if `Z` is the list `[`$v_1$`,`$v_2$`,...,`$v_k$`]`,
- `prod(1≤I≤N, I, F)`, which is equivalent to `F is 1*2*···*N`, and
- `all(X suffix_of L, A^B^Q^(X=[A,B|Q] -> A ≤B; true))`, which (after some rewriting) is equivalent to $v_1$`≤`$v_2$`,`$v_2$`≤`$v_3$`,...,`$v_{k-1}$`≤`$v_k$, if `L` is the list `[`$v_1$`,`$v_2$`,...,`$v_k$`]`.

Some of our example programs use arrays. A predication `size(`$n$`,`$a$`)` is true if $a$ is an array with $n$ elements. An expression $a$`[`$i$`]` denotes the $i$th (zero-based) element of the array $a$. For this paper it is irrelevant whether these arrays are implemented as ordinary structures (in which case `size` is implemented in terms of `functor` and array element expressions are implemented in terms of `arg`) or as a separate class of data structures. We discuss this matter in more depth elsewhere [20].

## 6.4 TRANSFORMING BOUNDED QUANTIFICATIONS TO RECURSIVE PROGRAMS

It is obvious that any bounded quantification can be replaced by a call to a recursively defined predicate. In this section we will present examples of bounded quantifications and corresponding recursively defined predicates. A more precise definition of the transformation can be found in appendix A.

*Example 1.* A bounded quantification `all(X in Z, (p(X), q(X,V)))` can be transformed to an atom `r(Z,V)` and the two program clauses

```
r([],V).
r([X|T],V) :- p(X), q(X,V), r(T,V).
```

*Example 2.* A bounded quantification `prod(1≤I≤N, I, F)` can be transformed to an atom `f(1, N+1, 1, F)` and the two program clauses

```
f(I, K, R0, R) :- I ≥ K, R0 = R.
f(I, K, R0, R) :- I < K, R1 is R0*I, J is I+1,
                  r(J, K, R1, R).
```

*Example 3.* A bounded quantification `all(X suffix_of L, A^B^Q^(X=[A,B|Q] -> A ≤ B; true))` can be transformed to a goal `r(L)` and the two clauses

```
r([]) :- [] = [A,B|T] -> A ≤ B; true.
r([H|T]) :- ([H|T]=[A,B|Q] -> A ≤ B; true), r(T).
```

The two clauses can obviously be simplified. As this example shows, existentially quantified variables of the body of a universal quantification need no such quantification in the transformed program, since they occur only in the body of the resulting clauses.

## 6.5 SEQUENTIAL EXECUTION

The operational semantics of bounded quantifications on a uniprocessor is analogous to that of an iteration in an imperative programming language, particularly for deterministic computation. Integer range expressions yield determinate iteration ("**for**-loops") while list member and list suffix range expressions yield special cases of indeterminate iteration ("**while**-loops").

We are assuming that the limits of integer ranges, or lists of list member and list suffix range expressions, are instantiated. We could devise more complete methods that could compute these ranges, but it seems preferrable to delay execution of bounded quantifications until the range information is available.

Elsewhere [20] we have shown show how bounded quantifications can be compiled on a sequential machine, both for deterministic and nondeterministic programs. The method is essentially to introduce iteration instructions in WAM.[2] For example, a bounded quantification `and(N≤I<M,` $\phi$`[I])` is compiled to code that corresponds quite directly to the following schema.

```
for(I = N ; I < M ; I++) {
    /* code for Φ[I] */
}
```

[2]Warren's Abstract Machine [214]; our base implementation was provided by Bevemyr [31].

For execution of arithmetical quantifiers, e.g., sums and products, an accumulator is used to hold the accumulated sum or product.

## 6.6 PARALLEL EXECUTION

The general idea for running bounded quantification on parallel computers is analogous to loop parallelization in, e.g., Fortran. Often many iterations of a bounded quantification, sometimes all iterations, can be run simultaneously.

We view each iteration of a bounded quantification as a separate task that can potentially be run in parallel. Synchronization primitives are used to handle data dependencies between iterations. Although we can certainly translate bounded quantifications directly to instruction sequences, we can understand the compilation process here as a transformation of bounded quantifications to recursive predicates. These recursive predicates are subsequently compiled using methods developed for running recursive programs [34, 32, 136]. These methods allow parallel execution of recursive programs in which binding of shared variables are deterministic. This restricts the set of bounded quantifications that can be executed in parallel using this technique. However, we believe a large set of interesting programs can be written without using nondeterministic binding of shared variables.

Unfortunately, the transformed arithmetic bounded quantifications use accumulators for computing sums and products, etc. Clearly, these accumulators introduce strict data dependencies. These can be dealt with using synchronization primitives: an efficient method is to let each worker have a local accumulator and add (or multiply) all local accumulators when the parallel execution terminates. For example, a bounded quantification `sum(0≤I<N, A[I])` is compiled corresponding to this schema:

```
for(I = 0, localacc = 0 ; I < N ; I++) {
    /* code for localacc = localacc+A[I] */
}
/* code for summing all local accumulators
 */
```

## 6.7 ABSTRACT MACHINE

We use the abstract machine designed for recursion-parallel execution developed by Bevemyr, Lindgren and Millroth [34, 32]. This machine consists of a set of workers numbered $0, 1, \ldots, n-1$; one per processor. Each worker is implemented as a separate process running a WAM-based Prolog engine with extensions that support parallel execution. One worker is responsible for sequential execution (the *sequential worker*). During sequential execution all other workers (the *parallel workers*) are idle.

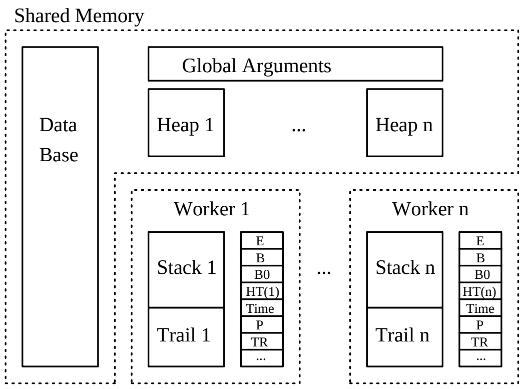


Figure 6.1: Overview of the recursion parallel machine.

## Data Sharing

A common code area is used and all workers have restricted access to each others heaps. All other data areas are private to each worker (see Figure 6.1). The shared heaps are used to communicate terms created during sequential and parallel execution.

Since more than one worker is allowed to bind a (shared) heap variable, care must be taken when binding such variables. We use an atomic exchange operation when binding variables to ensure that a worker does not destroy another worker's binding. This method is also discussed by Naish [146]. Using several shared heaps poses another problem: it is no longer possible to use a single heap pointer for determining whether to trail a binding or not. This problem is dealt with by associating a timestamp with each variable (either by extending the representation of each variable with a timestamp or by using an special tag for unbound variables).

The parallel workers have access to the temporary registers used by the sequential worker. These registers are used to communicate arguments for the parallel execution from the sequential worker to the parallel workers. Each parallel worker has an internal register in which the current iteration number of the worker is stored. This register is readable by all workers.

## Scheduling

There are two forms of scheduling, static scheduling and dynamic scheduling. It is possible to use static scheduling since the set of parallel processes is known when the parallel execution is initiated. There are situations in which dynamic scheduling is desirable, for example, when there is a great difference between the execution time of each iteration. We have therefore experimented with both static scheduling and dynamic scheduling. However, when executing our benchmarks we could not observe any difference between dynamic scheduling and static scheduling. We used the *self-scheduling* algorithm described by Tang and Yew [187] for dynamic scheduling. This algorithm allocates one iteration at a time.

## 6.8 ADDITIONS TO WAM

The instructions used for executing recursion-parallelism are also used for executing bounded quantifications. Bevemyr [32] describes the extended instruction set in detail. Here we give only a brief overview of the instructions required for executing bounded quantifications.

We describe each instruction in terms of its function when used for executing bounded quantifications. This differs slightly from the description given in the context of recursion parallelism. However, this is only a matter of perspective, the same implementation is used in both cases.

`start_left_body(`$L$`)`
: This instruction initiates parallel execution of the (body) code at label $L$. Sequential execution continues with the next instruction. This is the only new instruction executed by the sequential worker.

`initialize_left(`$S$`)`
: Each worker is initialized for parallel execution. The step $S$ is used for calculating the initial iteration number for the worker.

`spawn_left(`$S$`, `$i$`, `$n$`)`
: The next iteration to be executed by the worker is calculated by incrementing the internal register by the step $S$. If the result is greater than the value stored in the sequential worker register $\mathtt{X}_n$, the parallel execution is finished and the worker awaits the next parallel phase, otherwise the value is stored in $\mathtt{X}_i$.

`jump(`$L$`)`
: Execution is unconditionally continued at label $L$.

`put_global_arg(`$g$`, `$i$`)`
: The value of the sequential worker's register $\mathtt{X}_g$ is stored in the current worker's register $\mathtt{X}_i$.

`unify_global_arg(`$g$`)`
: The value of the sequential worker's register $\mathtt{X}_g$ is written on the heap. This instruction never occurs in read mode.

`lock_and_get_structure(`$F$`, `$i$`, `$n$`)`
: If $\mathtt{X}_i$ contains a variable, then write mode is entered and a structure

with the functor $F$ is initiated on the heap. A pointer to the new structure is stored in $\mathtt{X}_n$. If $\mathtt{X}_i$ contains a reference to a structure and the functor of the structure is $F$, the `S` register is set to point to the first argument of the structure, otherwise failure occurs.

`lock_and_get_list(`$i$`, `$n$`)`
: If $\mathtt{X}_i$ contains a variable, then write mode is entered and a list is initiated on the heap. A pointer to the new list is stored in $\mathtt{X}_n$. If $\mathtt{X}_i$ contains a list, the `S` register is set to point to the first argument of the list, otherwise failure occurs.

`unlock(`$i$`, `$n$`)`
: This instruction is ignored in read mode. In write mode the contents of $\mathtt{X}_i$ is unified with the contents of $\mathtt{X}_n$.

`await_leftmost`
: The current worker is forced to wait until all preceding iterations have finished.

`await_nonvar(`$i$`)`
: The current worker is forced to wait until the dereferenced value of $\mathtt{X}_i$ is a non-variable, or until all preceding iterations have finished.

`await_strictly_nonvar(`$i$`)`
: The current worker is forced to wait until the dereferenced value of $\mathtt{X}_i$ is a non-variable. If all preceding iterations finish and $\mathtt{X}_i$ remains a variable then a runtime error is signaled and the execution is aborted.

`await_variable(`$i$`)`
: The current worker is forced to wait until all preceding iterations have finished. If the variable referred to by $\mathtt{X}_i$ becomes bound during suspension, a runtime error is signaled and the execution is aborted.

In addition, the `builtin` instruction that is used to encode primitive predicates and operations is extended with some new cases that involve limited parallel computation within a sequential phase. In the examples, `builtin reduce_+` $\mathtt{X}_i$ $\mathtt{X}_j$ is used in the sequential phase to compute (presumably through parallel reduction) the sum of the contents of every parallel worker's $\mathtt{X}_i$ register, unifying the result with the contents of the sequential worker's $\mathtt{X}_j$ register.

## 6.9 EXAMPLES

To illustrate the use of some of these instructions we show the machine code for our benchmark programs.

**Inner Product**

The relation *e_i_p* holds for two vectors $X$ and $Y$, and a number $S$ if $S$ is the Euclidean inner product of $X$ and $Y$ (i.e., $S$ is the sum of the products of corresponding elements of $X$ and $Y$). We may express this as a bounded quantification.

```
e_i_p(X, Y, S) :-
    size(X, N), size(Y, N),
    sum(0≤ I<N, X[I] * Y[I], S).
```

The program can be compiled to the following sequence of machine instructions.

```
e_i_p/3:
  put_x_void X3              % X3 <- N
  builtin size X0 X3         % size(X,N)
  builtin size X1 X3         % size(Y,N)
  start_left_body L1         % execute L1 in parallel
  builtin reduce_+ X2 X2     % sum worker's X2 and
  proceed                    % unify with S
L1:
  initialize_left 1          % init worker
  put_global_arg G0 X0       % X0 <- X
  put_global_arg G1 X1       % X1 <- Y
  put_constant 0 X2          % localacc <- 0
L2:
  spawn_left 1 X3 G3         % X3 <- I++; while(I < N) do
  builtin array_ref X5 X0 X3 % Tmp1 <- X[I]
  builtin array_ref X6 X1 X3 % Tmp2 <- Y[I]
  builtin plus X5 X5 X6      % X5 <- X[I] * Y[I]
  builtin plus X2 X2 X5      % localacc<-localacc+X[I]+Y[I]
  jump L2                    % perform next iteration
```

**Numerical Integration**

The relation *intsimp* holds for three numbers $A$, $B$ and $I$, and a positive integer $N$, if $I$ is an approximation to the integral $\int_a^b f(x)dx$ where $A$ and $B$ are the limits $a$ and $b$, and $N$ is the number of intervals. We assume that the relation $r(X, Y)$ holds if and only if $f(X) = Y$, where $f$ is the function being integrated. We may express this as a bounded quantification

```
intsimp(A, B, N, I) :-
    W is (B-A)/N, Z is 2*N+1,
    size(G, Z),
    and(0≤ J<Z, r(A+J*W/2, G[J])),
    sum(0≤ J<N, W * (  G[2*J] +
                          4*G[2*J+1] +
                            G[2*J+2])/6, I).
```

that can be compiled to the following machine instructions (we have to omit some parts due to lack of space).

```
intsimp/4:
  /* code for calculating W in X4 */
  /* code for calculating 2*N+1 in X5 */
  put_x_void X6              % X6 <- G
  builtin array_size X6 X5   % size(G,2*N+1)
  start_left_body L1         % execute L1 in parallel
  start_left_body L3         % execute L3 in parallel
  builtin reduce_+ X3 X3     % sum workers' X3 and
  proceed                    % unify with I
L1:
  initialize_left 1          % initialize worker
L2:
spawn_left 1 X2 G5           % X2 <- J++; while(J<2*N+1) do
  put_global_arg G4 X0       % X0 <- W
  /* code for calculating J*W/2 in X0 */
  put_global_arg G0 X1       % X1 <- A
  builtin plus X0 X0 X1      % X0 <- A+J*W/2
  put_global_arg G6 X1       % X1 <- G
  builtin array_ref X1 X1 X2 % X1 <- Y <- G[J]
  call r/2,0                 % r(A+J*W/2,Y)
  jump L2                    % next iteration
L3:
  initialize_left 1          % initialize worker
  put_global_arg G4 X1       % X1 <- W
  put_global_arg G6 X2       % X2 <- G
  put_constant 0 X3          % localacc <- 0
L4:
  spawn_left 1 X0 G2         % X0 <- J++ ; while(J<N) do
  /* code for calculating
        W*(  G[2*J]+
           4*G[2*J+1]+
             G[2*J+2])/6
     in X5               */
  builtin plus X3 X3 X5      % localacc <- localacc+X5
  jump L4                    % next iteration
```

**Roots of an oriented forest**

The relation *find* holds for two vectors $P_0$ and $P$ if $P_0$ and $P$ are oriented forests representing the same equivalence relation and the depth of $P$ is one. We may express this as a bounded quantification

```
find(P0, P) :-
    size(P0, N),
    and(0≤ I<N, P0[I] = P0[P0[I]]),
    !,
    P0 = P.
find(P0, P) :-
    size(P0, N), size(P1, N),
    and(0≤ I<N, J^(P0[I] = J, P1[I] = P0[J])),
    find(P1, P).
```

The program can be compiled to the following machine instructions.

```
find/2:
 try_me_else L3
  put_x_void X2               % X2 <- N
  builtin array_size X0 X2    % size(P,N)
  start_left_body L1          % execute L1 in parallel
  cut                         % !
  get_x_value X0 X1           % P0=P
  proceed                     % done
L1:
  initialize_left 1           % initialize worker
  put_global_arg G0 X0        % X0 <- P0
L2:
  spawn_left 1 X3 G2          % X3 <- I++ ; while(I<N) do
  builtin array_ref X5 X0 X3  % X5 <- P0[I]
  builtin array_elt X5 X0 X5  % P0[I] = P0[P0[I]]
  jump L2                     % next iteration
L3:
 trust_me
  put_x_void X2               % X2 <- N
  builtin array_size X0 X2    % size(P0, N)
  put_x_void X6               % X6 <- P1
  builtin array_size X6 X2    % size(P1, N)
  start_left_body L4          % execute L4 in parallel
  put_x_value X6 X0           % X0 <- P1
  execute find/2              % find(P1,P)
L4:
  initialize_left 1            % initialize worker
  put_global_arg G0 X0         % X0 <- P0
  put_global_arg G6 X6         % X6 <- P1
L5:
  spawn_left 1 X3 G2           % X3 <- I++ ; while(I<N) do
  builtin array_ref X4 X0 X3   % X4 <- J <- P0[I]
  builtin array_ref X5 X0 X4   % X5 <- P1[I]
  builtin array_elt X3 X6 X5   % P1[I] = P0[J]
  jump L5                      % next iteration
```

## 6.10 BENCHMARKS

We use the same set of benchmarks that were used by Arro, Barklund and Bevemyr [11] to evaluate the parallel implementation on a SIMD multiprocessor (Thinking Machines Corp. Connection Machine model CM-200).

These benchmarks have been executed on a 26 processor Sequent Symmetry. This is a cache-coherent shared memory multiprocessor equipped with Intel 80386 processors. We used 24 of the available 26 processors, leaving 2 processors for operating system activities.

We compare the execution time of each program executed in parallel with the execution time of a sequential recursive program executed on a sequental WAM (without any extensions for parallel execution). We also compare the parallel implementation with the sequential implementation of bounded quantifications.

Relative speed-up expresses the ratio of single-processor execution time of the program (compiled with parallelization) to the time using $p$ processors.

Normalized speed-up expresses the ratio of single-processor execution time of the original sequential program (compiled without parallelization) to the time using $p$ processors on the program compiled with parallelization.

Let us also define

$$\begin{aligned}\text{parallel efficiency} &= \frac{\text{speedup on } N \text{ processors}}{N}\\ \text{normalized parallel efficiency} &= \frac{\text{normalized speedup on } N \text{ processors}}{N}\end{aligned}$$

### Performance

*Inner Product* This program calculates the Euclidian inner product of two vectors consisting of 120000 elements each. The program is described above.

Recursive program: 25.79 s.
Sequential BQ: 14.39 s.
Sequential part of parallel execution 0.0 s.

| Ws | Execution time (s) | Relative speed-up | Normalized speed-up Rec. | BQ | Relative efficiency | Normalized efficiency Rec. | BQ |
|---|---|---|---|---|---|---|---|
| 1 | 16.92 | 1 | 1.52 | 0.85 | 100 % | 152 % | 85 % |
| 2 | 8.74 | 1.94 | 2.95 | 1.65 | 97 % | 147 % | 82 % |
| 4 | 4.38 | 3.86 | 5.89 | 3.29 | 97 % | 147 % | 82 % |
| 8 | 2.19 | 7.73 | 11.77 | 6.57 | 97 % | 147 % | 82 % |
| 16 | 1.11 | 15.24 | 23.23 | 12.96 | 95 % | 146 % | 81 % |
| 24 | 0.75 | 22.56 | 34.38 | 19.18 | 94 % | 144 % | 80 % |

The parallel efficiency of this program is very high for all sizes of worker sets. The reason for this behaviour is that only a negligible fraction of the program is executed sequentially.

*Numerical Integration* This program calculates an approximation to the integral $\int_a^b f(x)dx$, as described above. We approximated the integral

$$\int_1^3 4/(x^2+1)dx$$

using 20000 intervals.

Recursive program: 30.50 s.
Sequential BQ: 23.38 s.
Sequential part of parallel execution 0.17 s.

| Ws | Execution time (s) | Relative speed-up | Normalized speed-up Rec. | BQ | Relative efficiency | Normalized efficiency Rec. | BQ |
|---|---|---|---|---|---|---|---|
| 1 | 26.19 | 1 | 1.16 | 0.89 | 100 % | 116 % | 89 % |
| 2 | 13.29 | 1.97 | 2.29 | 1.76 | 98 % | 114 % | 88 % |
| 4 | 6.72 | 3.90 | 4.54 | 3.47 | 97 % | 114 % | 87 % |
| 8 | 3.45 | 7.59 | 8.84 | 6.77 | 95 % | 110 % | 85 % |
| 16 | 1.80 | 14.55 | 16.94 | 12.99 | 91 % | 106 % | 81 % |
| 24 | 1.26 | 20.78 | 24.20 | 18.55 | 87 % | 101 % | 77 % |

The parallel efficiency decreases as the set of worker grows. This is due to the sequential part of the computation.

*Oriented Forest* This program was given a degenerated tree of 10000 elements. The program is as described above.

Recursive program: 22.78 s.
Sequential BQ: 21.51 s.
Sequential part of parallel execution 4.6 s.

| | | | Normalized | | | Normalized | |
|---|---|---|---|---|---|---|---|
| | Execution | Relative | speed-up | | Relative | efficiency | |
| Ws | time (s) | speed-up | Rec. | BQ | efficiency | (rec.) | (BQ) |
| 1 | 25.70 | 1 | 0.89 | 0.84 | 100 % | 89 % | 84 % |
| 2 | 15.62 | 1.64 | 1.46 | 1.38 | 82 % | 73 % | 69 % |
| 4 | 10.20 | 2.51 | 2.23 | 2.11 | 63 % | 56 % | 53 % |
| 8 | 7.60 | 3.38 | 3.00 | 2.83 | 42 % | 38 % | 35 % |
| 16 | 6.40 | 4.01 | 3.56 | 3.36 | 25 % | 22 % | 21 % |
| 24 | 6.13 | 4.19 | 3.72 | 3.51 | 17 % | 15 % | 15 % |

The reason behind the low efficiency when using 24 workers for this program is that it contains a large sequential part. A closer look at the sequential execution reveals that it is dominated by the time to create the temporary array. That operation can be speeded up by using uninitialized arrays and writing values into the array, instead of unifying each value with a distinct unbound variable in the array. We did this modification as an experiment, obtaining significantly better execution times and speed-up values.

Recursive program: 20.45 s.
Sequential BQ: 14.63 s.
Sequential part of parallel execution 0.03 s.

| | | | Normalized | | | Normalized | |
|---|---|---|---|---|---|---|---|
| | Execution | Relative | speed-up | | Relative | efficiency | |
| Ws | time (s) | speed-up | Rec. | BQ | efficiency | (rec.) | (BQ) |
| 1 | 19.13 | 1 | 1.07 | 0.76 | 100 % | 107 % | 76 % |
| 2 | 10.02 | 1.90 | 2.04 | 1.49 | 95 % | 102 % | 74 % |
| 4 | 5.14 | 3.72 | 3.98 | 2.85 | 93 % | 100 % | 71 % |
| 8 | 2.77 | 6.90 | 7.38 | 5.28 | 86 % | 92 % | 66 % |
| 16 | 1.72 | 11.12 | 11.89 | 8.51 | 70 % | 74 % | 61 % |
| 24 | 1.49 | 12.84 | 13.73 | 9.82 | 54 % | 57 % | 41 % |

The parallel efficiency increased accordingly.

**Parallelization Overhead**

The parallelization overhead is low for all programs above when compared with a sequential bounded quantification program. When compared with a sequential recursive program, the recursive program is often considerably slower. This is due to the fact that recursion, as implemented in WAM, has a considerably higher overhead than the iteration instructions used to encode bounded quantifications on sequential and parallel machines.

| Program | Overhead Rec. | Overhead BQ |
|---|---:|---:|
| Inner product | -52 % | 15 % |
| Numerical integration | -16 % | 11 % |
| Root find (initialized) | 11 % | 16 % |
| Root find (uninitialized) | -7 % | 24 % |

## 6.11 CONCLUSION

We show that the bounded quantifications and recursion (compiled with Reform Compilation [136]) can be implemented efficiently using the same extensions to WAM. From this we conclude that from a performance point of view, there is no significant difference between recursion and quantifications, and the programmer should be free to use the programming construct (bounded quantifications or recursion) that is suitable for a given problem—both can be efficiently executed on a shared memory multiprocessor. It may be possible to extend the instruction set with instructions supporting the data parallel execution that are more readily exploited when compiling bounded quantification programs, due to their simpler structure.

## 6.12 ACKNOWLEDGMENTS

We thank the Swedish Institute of Computer Science for making their 26 processor Sequent Symmetry available to us.

This research was supported financially by the Swedish Research Council for Engineering Sciences under contract No. 221–91–344.

# Prolog with Arrays and Bounded Quantifications

Jonas Barklund, Johan Bevemyr

To appear in Journal of Programming Languages.

It is proposed to add *bounded quantifications* to Prolog. The main reason is one of natural expression; many algorithms are expressed more elegantly in a declarative way using bounded quantifications than using existing means, i.e., recursion. In particular this is true for numerical algorithms, an area where Prolog has been virtually unsuccessful so far. Moreover, bounded quantification has been found to be at least as efficient as recursion, when applicable. We outline an implementation of some bounded quantifications through an extension of Warren's abstract Prolog machine and give performance figures relative to recursion. Finally, we have shown elsewhere that bounded quantification has a high potential for parallel implementation and we conclude in this paper that one can often run the same program efficiently on a sequential computer as well as on several kinds of parallel computers.

## 7.1 INTRODUCTION

In logic programming languages, such as Prolog [56], the only means for repetition has been through recursion. Recursion is well known to be theoretically sufficient (a universal Turing machine can be expressed in Horn clauses using recursion [198]) but there are many algorithms that can be expressed more concisely using definite iteration. By this we mean repeating some computation a predetermined[1] number of times, usually letting some variable range over the elements of a finite set, and combining the results. In an imperative programming language this usually means to repeat execution of a command for all integers in some range, combining the effects of the command invocations by serialization (e.g., the **for** loop of Pascal).

[1]That is, determined before the repetition commences, not necessarily at compile time.

Barklund & Millroth [26, 17] and Voronkov [211, 212] have proposed *bounded quantifications* as a computational device for repetition by iteration in logic programs. In this paper we propose a useful extension of the Prolog programming language with bounded quantifications and show how to implement it through an extension of Warren's abstract Prolog machine [214]. We also argue that such an extension is naturally followed by an introduction of arrays in Prolog.

Although we propose sequential operational semantics for bounded quantifications in this paper, we have shown elsewhere that many Prolog programs expressed with bounded quantifications can also be run efficiently on parallel computers using a data parallel model of computation (just as many Fortran programs can be parallelized in spite of the sequential operational semantics of that language). We have run programs on both SIMD computers (Connection Machine Model CM-2 by Thinking Machines Corp.) [11] and shared-memory MIMD computers (SUN 630MP and Sequent Symmetry) [19]. Much of the beauty of bounded quantifications lies in the fact that they have a clear declarative semantics, while at the same time they behave well operationally on this wide range of sequential and parallel computers.

The rest of the paper is structured as follows. Section 7.2 introduces syntax and informal semantics for bounded quantifications in Prolog. In Section 7.3 an extension of Prolog with arrays is proposed and motivated. Section 7.4 discusses sequential operational semantics for bounded quantifications. The implementation of Warren's abstract Prolog machine in which our experiments have been carried out is briefly described in Section 7.5. In Section 7.6 a set of instructions extending Warren's machine is proposed; compilation of bounded universal quantifications to the extended machine instruction set is defined in Section 7.7. Section 7.8 contains seven examples of Prolog programs with bounded quantifications and their resulting code, in some cases with corresponding recursive programs for comparison, ended with a list of measured run times. In Section 7.9 some further quantifiers and iterators are discussed. There are some related developments, reviewed in Section 7.10. Section 7.11 ends the paper with some conclusions and prospects for future work.

This paper is a significantly extended and slightly revised version of a paper that was presented at the Fourth International Conference on Logic Programming and Automated Reasoning and can be found in its proceedings [20].

## 7.2 BOUNDED QUANTIFICATION

Following Tennent [189, 190] in spirit, we define a *quantification* to be an expression that has three parts: a *quantifier*, which is a symbol from a given alphabet of quantifiers, a locally scoped *iteration variable*, and a *body*, which is itself an expression. The semantics of a quantification is that obtained by evaluating the body for every value ranged over by the local variable, combining the results according to the quantifier. This combination is typically a reduction with an associative and commutative binary function, in which case the quantifiers can alternatively be seen as a reduction operator, but it is not always the case. A *bounded quantification* is a quantification where the local variable ranges over a *finite* set of values, given by a *range expression*.

Some examples of bounded quantifications using ordinary mathematical notation are

$$(\forall\, 5 \le i < 10)\, p(i) \leftarrow q(i),$$
$$(\exists\, x \in l)\, x > 54,$$
$$\text{and} \quad \textstyle\sum_{0 \le k < z} \frac{1}{(2k+1)(2k+3)}$$

where $i$ in the first and $k$ in the third quantification should be considered implicitly restricted to take on integer values only. The first two quantifications are truth-valued expressions while the last expression is number-valued (approximating $\pi/8$ for large values of $z$).

We prefer to stress the similarities, rather than the differences, between quantifications having different quantifiers. Nevertheless, when proposing to add bounded quantifications to the established programming language Prolog it is appropriate to provide a slightly different syntax for logical bounded quantifications and arithmetical bounded quantifications. Moreover, in order to be coherent with related existing Prolog constructs, such as `setof`, we propose a syntax where the quantifiers actually appear as if they were predicate symbols.[2]

The syntax of a *bounded universal quantification* is

`all(`$\rho[i]$`, `$\beta[i]$`)`

where $\rho[i]$ is a range expression containing an iteration variable $i$ and $\beta[i]$ is a truth-valued expression (a goal) that typically contains $i$. If the range expression $\rho[i]$ is true for the values $v_1$, $v_2$, …, $v_k$ of $i$, then the bounded quantification is logically equivalent to the conjunction $\beta[v_1]$`,` $\beta[v_2]$`,` …`,` $\beta[v_k]$. If the range expression is true for no value of $i$ then the bounded universal quantification succeeds trivially.

The syntax of a *bounded existential quantification* is

`some(`$\rho[i]$`, `$\beta[i]$`)`

[2]In other logic programming languages, other design choices may be appropriate [23].

where $\rho[i]$ is a range expression and $\beta[i]$ is a goal. If the range expression $\rho[i]$ is true for the values $v_1$, $v_2$, …, $v_k$ of $i$ then the bounded quantification is logically equivalent to the disjunction $\beta[v_1]$; $\beta[v_2]$; …; $\beta[v_k]$. If the range expression is true for no value of $i$ then the bounded existential quantification fails.

The syntax of a *bounded arithmetical quantification* is

$\kappa(\rho[i],\ \beta[i],\ v)$

where $\kappa$ is a predicate symbol representing a quantifier (we propose to include at least `sum`, `product`, `max` and `min`), $\rho[i]$ is a range expression, $\beta[i]$ is a numeric expression, and $v$ is a numeric expression or a variable. If the range expression $\rho[i]$ produces the values $v_1$, $v_2$, …, $v_k$, then the bounded quantification is logically equivalent to the goal

$v$ `is` $\beta[v_1] \oplus_\kappa \beta[v_2] \oplus_\kappa \cdots \oplus_\kappa \beta[v_k]$,

where $\oplus_{\texttt{sum}}$ is `+`, etc. If the range expression produces no values the bounded quantification is logically equivalent to

$v$ `is` $c_\kappa$,

where $c_{\texttt{sum}}$ is `0`, etc.

A simple range expression is an *iterator*. We propose three kinds of iterators; others are certainly conceivable but many of those can be expressed in terms of these three. An *integer iterator* $\rho[i]$ is an expression

$m \le i < n$,

where $m$ and $n$ are integer expressions (that could be evaluated with `is`), and $i$ is a variable; $i$ will take on the integer values $m$, $m+1$, …, $n-1$, in that order (if $m \geq n$ then $i$ will take on no values).[3] A *list head iterator* is an expression

$i$ `in` $l$,

where $l$ is a list expression. If $l$ is the list `[`$v_1$`,`$v_2$`,`…`,`$v_k$`]` then $i$ will take on the values $v_1$, $v_2$, …, $v_k$, in that order. A *list tail iterator* is an expression

[3]The obvious extensions $m \le i \le n$, $m < i < n$ and $m < i \le n$ should also be recognized as iterators and can be defined in terms of $m \le i < n$; we will not discuss them further here.

$i$ `suffix_of` $l$,

where $l$ is a list expression. If $l$ is the list `[`$v_1$`,`$v_2$`,...,`$v_k$`]` then $i$ will take on the values `[`$v_1$`,`$v_2$`,...,`$v_k$`]`, `[`$v_2$`,...,`$v_k$`]`, ..., `[`$v_k$`]`, `[]`, in that order.

Bounded quantifications are readily extended to have several iteration variables, in which case the body will be computed for every combination of values of the individual iteration variables.

A *range expression* is a nonempty conjunction of iterators, possibly conjugated with an arbitrary goal that functions as a discriminating test. Without loss of generality, assume that each range expression is on the form

$\rho[i_1]$, ..., $\rho[i_k]$, $\gamma[i_1,\ldots,i_k]$.

It will generate every combination of values for $i_1$, ..., $i_k$ generated by the iterators, satisfying the test $\gamma[i_1,\ldots,i_k]$.

As pointed out by Voronkov [211, 212], it will often be convenient to write unbounded existential quantifications in the bodies of bounded quantifications. (In repetition expressed as a recursive program clauses this is never required. However, Prolog's `setof` facility, which can be seen as a kind of quantification, has already motivated such an extension.)

For example,

```
all(1 ≤ I ≤ K, J^r(I, m(I,J)))
```

is a bounded universal quantification that is equivalent to the conjunction

r(1, $J_1$), r(2, $J_2$), ..., r(K, $J_k$),

where $J_1$, ..., $J_k$ are new distinct variables, and the goal (see below concerning array syntax)

```
sum((J index_of X, K index_of Y), X[J]*Y[K], S)
```

unifies `S` with the sum of every combination of elements from the one-dimensional arrays `X` and `Y`.

## 7.3 ARRAYS IN PROLOG

Much as recursion, as a means for repetition, goes naturally with inductively defined data structures, so does iteration over integer ranges go naturally with arrays. In order to still have a well-composed language, after the addition of bounded quantifications, it therefore makes good sense to also have arrays in Prolog.

It can be argued that Prolog already has arrays, for compound terms of arity $k$ can be indexed with the integers 1 to $k$. An 'array' term can then be written directly as a $k$-ary term, or declared using the built-in `functor` predicate. Array element access is then provided through the `arg` predicate. In the interest of keeping the collection of language concepts small, one could therefore argue against extending Prolog with arrays as a separate datatype.

However, we feel that there are stronger arguments *for* adding arrays to Prolog.

- With arrays implemented using the ordinary internal representation of terms there is no support for efficiently "updating" an array, in order to obtain an array that differs somewhat from a given array, possibly in a single element. Constant-time mechanisms with a reasonable overhead for that operation have been investigated, e.g., by Eriksson & Rayner [74]; note also the work on multiple reference management by Chikayama & Kimura [50]. The cost for supporting such declarative updates is not motivated for arbitrary data structures.

- Arrays may be multidimensional, which is awkward to accomplish when arrays are represented by ordinary terms; at the very least one should extend `arg` or provide another predicate for convenient access to components of multidimensional arrays.

- Array operations, in particular bounded quantifications that operate upon array elements, have a high potential for efficient parallel computation. Having arrays as a separate data type may simplify parallel implementation, as it will often make sense to spread the representation of arrays over several memory units on distributed computers.

- There is nothing fundamental in logic programming theory or practice that forces identification of arguments to predicates and subterms to be position-based, rather than identifier-based. In future developments of Prolog the compound terms might become more like the records of conventional programming languages. In such a scenario the addition of arrays as a separate datatype is inevitable.

Prolog's current set of terms obeys some very simple rules:

- The predefined ternary relation *functor* retrieves principal symbol and arity of any term, or declares a term with a given principal symbol and arity.

- For any term whose arity (according to the *functor* relation) is nonzero, each proper subterm can be accessed through the predefined ternary *arg* relation.

In our earlier research, we investigated how arrays, tables and other data structures could be incorporated into Prolog by making them obey the rules above in such a way that they were denoted by terms of a fixed arity, in the same way as for lists and trees [25]. The reason was that we wanted to be able to use recursion also for traversing and constructing arrays, tables, etc. We have abandoned that effort and propose instead to use bounded quantification for this purpose, in accordance with the discussion at the beginning of this section. The syntax for arrays is not very critical, because it is unlikely that it will be used for matching. A reasonable syntax for a Prolog array with $k$ elements is

`(/` $t_0$`,` $t_1$`, ...,` $t_{k-1}$ `/)`,

similar to Fortran 90 [134]. If each element $t_i$, $0 \leq i < k$, is a $d$-dimensional array with $l$ elements, we say that this array is $d{+}1$-dimensional, otherwise it is one-dimensional. For example, `(/(/a, b, c/), (/d, e, f/)/)` denotes a two-dimensional $2 \times 3$-element array.

For the rest of the paper we assume that

- there is a ternary relation *size* such that $size(i, a, d)$ is **true** whenever $i$ is a non-negative integer, $a$ is an array and $d$ is the number of elements in the $i$th dimension of $a$;
- there is a ternary relation *elt* such that $elt(j, a, x)$ is **true** whenever $j$ is a non-negative integer, $a$ is an array and $x$ is the $j$th element of $a$. Array indices thus always go from 0 to $k - 1$ where $k$ is the number of elements in the first dimension of the array;
- an expression $i$ `index_of` $a$, where $a$ is a one-dimensional array of $s$ elements, is an iterator that is equivalent to `0 ≤` $i$ `<` $s$;
- an expression $(i_1, \ldots, i_k)$ `index_of` $a$, where $a$ is a $k$-dimensional $s_1 \times \cdots \times s_k$ array, is an iterator that is equivalent to `0 ≤` $i_1$ `<` $s_1$, ..., `0 ≤` $i_k$ `<` $s_k$;
- the equality theory of Prolog has been extended in a way similar to the Gödel logic programming language [101]. More specifically:
  - The value of any expression of the form $e_1 + e_2$, $e_1 - e_2$, $e_1 \times e_2$ or $e_1/e_2$, where the values of $e_1$ and $e_2$ are numbers, is the value of applying the corresponding arithmetical operation to these values.

- The value of an expression $e[e_0]$ is the unique value $x$ for which the relation $elt(e_1, e_0, x)$ holds, if any such value exists.
- The value of an expression $e[e_0, e_1, \ldots, e_{k-1}]$ is the same as that of the expression $x[e_1, \ldots, e_{k-1}]$, where $x$ denotes the value of the expression $e[e_0]$.

These conventions are easily implemented and many programs become easier to read and understand.

## 7.4 OPERATIONAL SEMANTICS

Most quantifications are naturally concurrent constructs, there is nothing in a bounded universal quantification as such that dictates that the instances of the body should be proved in a particular order. However, Prolog is a sequential programming language where a programmer expects that side effects will be performed and multiple solutions generated in a certain order. When adding bounded quantifications to Prolog it is therefore natural and even important to decide at least a weak operational semantics that specifies the order in which the instantiations of iteration variables are applied. In a parallel implementation this order should, of course, be ignored when there are no side effects in the bodies of bounded quantifications, in order not to hinder parallelization. This is, of course, the same problem as with any other proposal for adding parallelism to full Prolog, rather than to a pure Horn clause language.

For Prolog we propose that integer iterators produce values in increasing order, and that list head and list tail iterators produce subsequent list element and list tails, respectively. A range expression $\rho[i_1]$, …, $\rho[i_k]$, $\gamma[i_1, \ldots, i_k]$, which contains iterators for the $k$ distinct iteration variables $i_1$, …, $i_k$, should produce collections of values of the iteration variables in lexicographic order: for the first value of $i_1$ producing all combinations of $i_2$, …, $i_k$, then using the second value for $i_1$, etc.

As it is sometimes convenient to iterate with a range of integers in decreasing order, we propose $m \geq i > n$ as an alternative integer iterator that has the same declarative semantics as an iterator $n + 1 \leq i < m + 1$ but produces the values $m$, $m - 1$, …, $n - 1$ for the iteration variable $i$, in that order.

There is a problem when an iterator is not sufficiently instantiated to determine the range of an iteration variable. Consider, for example, an iterator `3 ≤ J < R`, where `R` is not instantiated. One could devise a procedure that tried all possible values for `R` (i.e., all represented integers) but we think that, for Prolog, this is not a useful behaviour and such a situation should trigger a run-time error.

In a Prolog system that is capable of suspending computation of nondeterministic primitives that may create infinitely (or "too many") solutions (such as when using the `block` declarations of Sicstus Prolog), the same treatment should be applied to bounded quantifications having insufficiently instantiated iterators.

It should be clear to the reader at this point that we intend the machine code for bounded quantifications to form loops, where the code for a body is terminated by a conditional jump back to the beginning, or some variant thereof.

## 7.5 THE LUTHER WAM EMULATOR

Our experiments have been conducted in an implementation of Warren's abstract Prolog machine [214] which is based on an emulator written in the C programming language [31, 32].

To better understand our code sequences below one ought to know the following.

1. There are two instructions `builtin` and `inline` that do the work of various predefined relations, known to the compiler. Both instructions take a variable number of arguments but the first argument always identifies the operation to be performed. The difference between `builtin` and `inline` is that the second argument of `inline` is always a label giving a failure continuation. Therefore the equivalence

   `builtin` *op* $arg_1$ ... $arg_k$ $\equiv$ `inline` *op* `fail` $arg_1$ ... $arg_k$

   always holds.

2. Luther has one-dimensional arrays. A goal `size(0,A,S)` where `A` and `S` are in registers `X`$i$ and `X`$j$, respectively, is compiled to `builtin size0` $i$ $j$. A goal `elt(I,A,X)` where `I`, `A` and `X` are in registers `X`$i$, `X`$j$ and `X`$k$, respectively, is compiled to `builtin elt` $i$ $j$ $k$.

3. The argument/temporary registers and stack variables are numbered from zero and upwards, unlike the original WAM.

## 7.6 NEW INSTRUCTIONS FOR BOUNDED QUANTIFICATIONS

This section proposes nine instructions that can be added to WAM in order to support efficiently bounded universal and arithmetical quantifications. The effect of the instructions is described here and they are described in context in the next section.

**Integral iterators**

The `iterate_int` and `iterate_int*` instructions are issued at the end of loops. The `zerop` instruction is not strictly necessary but covers a frequent special case. The `allocate*` instruction is exactly the same an ordinary `allocate` instruction except that it receives the size of the topmost environment explicitly. The instruction `save_registers` simply replaces a sequence of `get_variable` instructions; in an emulator-based implementation this can make a difference. `repeat` is the most complex of these instructions. Before jumping to the beginning of the loop, it restores the registers saved by a `save_registers` instruction (except the counter and the upper limit) and, if necessary, allocates a new environment and saves the registers there too.

```
iterate_int i j label =
    if (Xi++ < Xj) goto label;

iterate_int* i j label =
    if (Xi++ ≥ Xj) goto label;

zerop i label =
    if (Xi <= 0) goto label;

allocate* k =
    CE = E;
    E = (CE < B ? B : CE + k+2);
    E->CP = CP;
    E->CE = CE;

save_registers k =
    Y0 = X0;
    Y1 = X1;
    ⋮
    Y(k − 1) = X(k − 1);

repeat label =
{   int k = env_size(CP);

    X0 = Y0;
    X1 = Y1;
    ⋮
    X(k − 3) = Y(k − 3);
    if (E < B)
    {   CE = E;
        E = B;
        E->CP = CP;
        E->CE = CE;
        Y0 = X0;
        Y1 = X1;
        ⋮
        Y(k − 1) = X(k − 1);
    }
    goto label
}
```

**List Iterators**

As will be explained below, it is not a good idea to transform list iterators to integer iterators. We therefore need the instructions `listp`, `iterate_list` and `iterate_list*` that go down a list, (almost) analogously with `zerop`, `iterate_int` and `iterate_int*`. (Actually `listp` and `iterate_list*` are identical.)

```
listp i label =
    if (!list(Xi)) goto label;

iterate_list i label =
    if (list(Xi)) goto label;

iterate_list* i label =
    if (!list(Xi)) goto label;
```

## 7.7 COMPILING UNIVERSAL AND ARITHMETICAL BOUNDED QUANTIFICATIONS

The compilation schemas below cover explicitly only bounded universal quantifications having a range formula that is an integer iterator, an `in` iterator or a `suffix_of` iterator. Other iterators, such as `index_of`, are readily transformed to integer iterators with the help of additional goals for determining the limits. Conjunctions of iterators are straightforwardly compiled to nested loops. Any tests that are part of range formulas are compiled as such and put first in the code for the body.

The code for the various arithmetical quantifiers is similar but uses one more register for accumulating values. It should be clear from the examples in the next sections how to compile bounded arithmetical quantifications.

We will discuss compilation of a bounded universal quantification on the form

all($m$ ≤ I < $n$, $\Phi$).

We distinguish between three cases, depending on the body of the bounded quantification.

1. The body does not call any out-of-line predicates and the in-line predicates cannot introduce choice points.
2. The body calls out-of-line predicates but for some reason they are known not to create any choice points.
3. The body calls out-of-line predicates that may introduce choice points.

It would be sufficient to always compile according to the third case but the special cases are more efficient and should be used when possible.

**All In-line**

As there are no out of line calls the body requires no environment and, because no choice points can be allocated, any failure will go back past entering the quantification. We assume that the registers X$i$ and X$j$ are free.

```
    code for storing the value of m in Xi
    code for storing the value of n in Xj
    inline '<' L2 Xi Xj
L1:     code for Φ
    iterate_int Xi Xj L1
L2:
```

Register X$i$ is used for the counter while register X$j$ contains the upper limit; they are set up by the first two instructions (which may be `put_value` instructions if the limits are given as variables). The following inline test verifies that there are values to try. If the compiler can show that $m < n$ the instruction can be omitted. The `iterate_int` instruction is the only overhead in the loop.

**Deterministic**

There are predicates in the body and an environment must be allocated for the clause in which the bounded quantification occurs. It is presumed that no choice points can be introduced in the body, thus ensuring that backtracking will never occur inside it.

The compiler should compile the body of the bounded quantification making sure that the register allocation when the loop is entered the first time is compatible with that at the end of the loop.

Permanent local variables are special in that their previous values are normally overwritten at the beginning of each iteration. (Temporary local variables are of course also overwritten, but that is not particular to bounded quantifications.) The exception is the iteration variable, as will be seen below.

The following code assumes that there is an environment, that the registers X$i$ and X$j$ are free, and that slots Y$k$ and Y$l$ are free in the environment.

```
    code for storing the value of m in Xi
    code for storing the value of n in Xj
    inline '<' L2 Xi Xj
    get_variable Yl Xj
L1: get_variable Yk Xi
        code for Φ
    put_value Yk Xi
    put_value Yl Xj
    iterate_int Xi Xj L1
L2:
```

The registers are used in the same way as for the previous case. A loop invariant at L1 is that X$i$ contains the counter and that both X$j$ and Y$l$ contain the upper limit. Apart from the cost of accessing permanent variables, the total overhead in the loop is the four instructions `get_variable`, `put_value`, `put_value` and `iterate_int`. This should be contrasted with a corresponding recursive program for which one must allocate and deallocate environments at every recursive call, even if it if known that the body of the recursive clause is deterministic.

### General

There are predicates in the body and an environment must be allocated for the clause in which the bounded quantification occurs. The body may introduce choice points and one must therefore be careful not to overwrite local variables when the previous iteration has introduced a choice point. This is completely analogous to tail recursion optimisation: the environment must not be deallocated if there is a more recent choice point. This is accomplished below by the `repeat` instruction, which may allocate an environment of the same size as the current environment and copy (most of) it.

The following code assumes that $k$ permanent variables are needed in the body (apart from the counter and the upper limit) and that the compiler has placed the values for the non-local variables in registers `X0` to `X`$(k-1)$. The registers `X`$k$ and `X`$(k+1)$ are thus free.

```
    code for storing the value of $m$ in X$(k+1)$
    code for storing the value of $n$ in X$k$
    inline '<' L3 X$(k+1)$ X$k$
    allocate* $k+2$
    save_registers $k+1$
L1:     code for $\Phi$
    put_value Y$k$ X$k$
    put_value Y$(k+1)$ X$(k+1)$
    iterate_int* X$(k+1)$ X$k$ L2
    repeat L1
L2: deallocate
L3:
```

This is similar to the deterministic case, but a new environment is allocated for the bounded quantification. Because any previous environment then becomes inaccessible it is important that everything that is needed has been moved to registers, much as for an ordinary predicate call. The new environment is initialized from the temporary registers (`get_variable` instructions are not needed for the locally quantified variables). If there is no current environment where the bounded quantification occurs, the `allocate*` instruction above should be replaced by an ordinary `allocate` instruction.

The `repeat` instruction copies the current environment if there is a more recent choice point; it obtains information about the size of the environment through the `cp` register, as usual (although one could make it an argument to the instruction, because the size is statically available). Every `call` instruction in the code for $\Phi$ should thus claim the environment size to be $k+2$.

If the `repeat` instruction finds that the current environment is (still) more recent than the choice point, that test is the only extra overhead as compared with the previous case. In case of nondeterminacy there is the cost of copying the environment, but that cost is comparatively low due to that copying is done by a very tight loop inside the `repeat` instruction, rather than by a sequence of instructions. In addition, the `repeat` instruction always copies `Y0`, ...`Y`$k$ to `X0`, ..., `X`$k$, so all information is always available in registers at the beginning of each iteration.

## 7.8 EXAMPLES

These examples show how to express some algorithms as bounded quantifications, comparing with corresponding recursive programs. Machine code for the programs is also given, illustrating the compilation schemas above.

### Inner Product

The relation *e_i_p* holds for two vectors $x$ and $y$, and a number $s$ if $s$ is the Euclidean inner product of $x$ and $y$ (i.e., $s$ is the sum of the products of corresponding elements of $x$ and $y$).

```
e_i_p(X, Y, S) :-
   size(0, X, N), size(0, Y, N),
   sum(0 ≤ I < N, X[I] * Y[I], S).
```

It can be compiled to the following machine instructions.

```
e_i_p/3:
    put_void X3            {initialize N}
    builtin size X0 X3     {compute size of X in N}
    builtin size X1 X3     {verify that size of Y is N}
    zerop X3 L2            {void loop?}
    put_constant 0 X4      {set accumulator=0}
    put_constant 0 X5      {set I=0}
L1: put_void X6            {initialize}
    builtin elt X5 X1 X6   {access Y[I]}
    put_void X7            {initialize}
    builtin elt X5 X0 X7   {access X[I]}
    builtin '*' X6 X6 X7   {compute X[I]*Y[I]}
    builtin '+' X4 X4 X6   {add it to accumulator}
    iterate_int X5 X3 L1   {increment I, loop again?}
L2: get_value X2 X4        {unify S with accumulator}
    proceed                {return}
```

One corresponding recursive program for the same relation is

```
e_i_p(X, Y, S) :-
   size(0, X, N),
   size(0, Y, N),
   e_i_p_rec(N, X, Y, 0, S).

e_i_p_rec(0, _, _, S0, S) :- !, S0 = S.
e_i_p_rec(I, X, Y, S0, S) :-
```

```
   J = I-1,
   S1 = S0 + X[J] * Y[J],
   e_i_p_rec(J, X, Y, S1, S).
```

for which a compiler might produce the instructions below. We include this (not particularly original) code sequence to convince the reader that the same degree of optimisation has been applied in the compilation of these programs.

```
e_i_p/3:
    put_void X5
    builtin size0 X0 X5
    builtin size0 X1 X5
    put_value X5 X0
    put_value X0 X1
    put_value X1 X2
    put_constant 0 X3
    put_value X2 X4
    execute e_i_p_rec/5

e_i_p_rec/5:
    switch_on_term L2 L4 L1 L4 L4
L1: switch_on_constant 1 (0 L4) L5
L2: try L4
    trust L5
L3: get_constant 0 X0
    get_value X3 X4
    proceed
L4: builtin '1-' X0 X0
    put_void x6
    builtin elt X0 X1 X6
    put_void X7
    builtin elt X0 X2 X7
    builtin '*' X5 X6 X7
    builtin '+' X3 X3 X5
    execute e_i_p_rec/5
```

**Factorial**

The relation *factorial* holds for two nonnegative integers $n$ and $f$ if $f$ is the factorial of $n$.

```
factorial(N, F) :-
   product(1 ≤ I ≤ N, I, F).
```

which can be compiled to the following machine instructions.

```
    try_me_else L3          {alternatively try 2nd clause}
factorial/2:
    put_constant 1 X2       {set accumulator=1}
    builtin '1+' X0 X0      {compute N+1}
    put_constant 1 X3       {set I=1}
    inline '<' L2 X2 X0     {void loop?}
L1: builtin '*' X3 X3 X2    {multiply accumulator with I}
    iterate_int X2 X0 L1    {increment I, loop again?}
L2: get_value X1 X3         {unify F with accumulator}
    proceed                 {return}
```

Note that this bounded quantification is compiled to a quite tight loop, consisting of only two instructions. One corresponding recursive program computing the same relation follows.

```
factorial(N, F) :-
   factorial_rec(N, 1, F).

factorial_rec(0, F0, F) :- !, F0 = F.
factorial_rec(N, F0, F) :-
   F1 = F0 * N,
   M = N-1,
   factorial_rec(M, F1, F).
```

If one were to code this recursive program to count up (like the bounded quantification program), instead of down, it would become somewhat slower. It would become slower still if we assumed that the compiler was not clever enough to never generate any choice points when the first argument is ground.

```
factorial/2:
    put_value X1 X2
    put_constant 1 X1
    execute factorial_rec/3

factorial_rec/3:
    switch_on_term L2 L4 L1 L4 L4
L1: switch_on_constant 1 (0 L3) 4
L2: try L3
    trust L4
L3: get_constant 0 X0
```

```
    get_value X1 X2
    proceed
L4: builtin '*' X1 X0 X1
    builtin '1-' X0 X0
    execute factorial_rec/3
```

Still, when running each program 5 000 times with input 1000, the bounded quantification program is on average more than twice as fast as the recursive program.[4]

## Numerical Integration

The relation *intsimp* holds for three numbers $a$, $b$ and $i$, and a positive integer $n$ if $i$ is an approximation to the integral $\int_a^b f(x)\,dx$ where $n$ is the number of intervals. We assume that the proposition $\varphi[x, y]$ is true if and only if $f(x) = y$, where $f$ is the function being integrated.

```
intsimp(A, B, N, I) :-
   W = (B-A)/N,
   size(0, G, 2*N+1),
   all(0 ≤ J < 2*N+1, φ[A+J*W/2,G[J]]),
   sum(0 ≤ J < N, W * (  G[2*J] +
                       4*G[2*J+1] +
                         G[2*J+2])/6, I).
```

which can be compiled to the following machine instructions (we omit the least interesting parts in the interest of brevity).

```
intsimp/4:
    allocate                {make stack frame, max size 7}
    get_variable Y2 X2      {store N}
    get_variable Y3 X3      {store I}
    get_variable Y0 X0      {store A}
    builtin '-' X1 X1 X0    {compute B-A}
    builtin '/' X1 X1 X2    {compute W=(B-A)/N}
    get_variable Y1 X1      {store W}
    put_constant 2 X4       {load 2}
    builtin '*' X2 X2 X4    {compute 2*N}
    builtin '1+' X2 X2      {compute 2*N+1}
    put_variable Y5 X3      {store G}
    builtin size0 X3 X2     {G is now an array}
```

[4]The results of both programs are garbage in our implementation because it does not provide integers of arbitrary precision; the run times are still relevant.

```
    get_variable Y4 X2      {store 2*N+1}
    zerop X2 L2             {void loop?}
    put_constant 0 X0       {set J=0}
L1: get_variable Y6 X0      {save J}
    put_value Y5 X2         {load G}
    put_void X1             {initialize}
    builtin elt X0 X2 X1    {access G[J]}
    put_value Y1 X2         {load W}
    builtin '*' X0 X0 X2    {compute J*W}
    put_constant 2 X2       {get 2}
    builtin '/' X0 X0 X2    {compute J*W/2}
    put_value Y0 X2         {get A}
    builtin '+' X0 X0 X2    {compute A+J*W/2}
    call r/2 7              {call r(A+J*W/2,G[J])}
    put_value Y6 X0         {load J}
    put_value Y4 X2         {load 2*N+1}
    iterate_int X0 X2 L1    {increment J, loop again?}
L2: put_value Y2 X0         {load N}
    zerop X0 L4             {void loop?}
    put_constant 0 X1       {set J=0}
    put_constant 2 X10      {load 2}
    put_constant 4 X11      {load 4}
    put_constant 6 X12      {load 6}
    put_value Y5 X13        {load G}
    put_constant 0 X14      {set accumulator = 0}
L3: builtin '*' X2 X10 X1   {compute 2*J}
    put_void X3             {initialize}
    builtin elt X2 X13 X3   {access G[2*J]}
    builtin '1+' X2 X2      {compute 2*J+1}
    put_void X4             {initialize}
    builtin elt X2 X13 X4   {access G[2*J+1]}
    builtin '*' X4 X11 X4   {compute 4*G[2*J+1]}
    builtin '1+' X2 X2      {compute 2*J+2}
    put_void X5             {initialize}
    builtin elt X2 X13 X5   {access G[2*J+2]}
    builtin '+' X3 X3 X4    {compute G[2*J]+4*G[2*J+1]}
    builtin '+' X3 X3 X5    {compute G[2*J]+...+G[2*J+2]}
    put_value Y1 X4         {load W}
    builtin '*' X3 X4 X3    {compute W*(G[2*J]+...+...)}
    builtin '/' X3 X3 X12   {compute W*(G[2*J]+...+...)/6}
    builtin '+' X14 X14 X3  {add it to accumulator}
    iterate_int X1 X0 L3    {increment J, loop again?}
L4: get_value Y3 X14        {unify accumulator with I}
    deallocate              {remove stack frame}
    proceed                 {return}
```

**Fibonacci**

The relation *fibonacci* holds for two integers $n$ and $f$ if $f$ is the $n$th Fibonacci number (the pairs in the relation are $(1, 1)$, $(2, 1)$, $(3, 2)$, $\ldots$).

```
fibonacci(N, F) :-
   size(0, T, N),
   all(0 ≤ I < N,
         (I = 0 ->     T[I] = 1 ;
          I = 1 ->     T[I] = 1 ;
          otherwise -> T[I] = T[I-1]+T[I-2]),
   F = T[N-1].
```

which can be compiled to the following machine instructions.

```
fibonacci/2:
    put_void X2             {initialize T}
    builtin size0 X2 X0     {T is now an array}
    zerop X0 L5             {void loop?}
    put_constant 0 X3       {set I=0}
L1: put_constant 0 X4       {load 0}
    inline '=' L2 X3 X4     {is I=0?}
    put_constant 1 X4       {yes, load 1}
    builtin elt X3 X2 X4    {make T[I]=1}
    jump L4
L2: put_constant 1 X4       {load 1}
    inline '=' L3 X3 X4     {is I=1?}
    builtin elt X3 X2 X4    {yes, make T[I]=1}
    jump L4
L3: builtin '1-' X4 X3      {compute I-1}
    put_void X5             {initialize}
    builtin elt X4 X2 X5    {access T[I-1]}
    builtin '1-' X4 X4      {compute I-2}
    put_void X6             {initialize}
    builtin elt X4 X2 X6    {access T[I-2]}
    builtin '+' X5 X5 X6    {compute T[I-1]+T[I-2]}
    builtin elt X3 X2 X5    {make T[I]=T[I-1]+T[I-2]}
L4: iterate_int X3 X0 L1    {increment I, loop again?}
L5: builtin '1-' X0 X0      {compute N-1}
    builtin elt X0 X2 X1    {unify F with T[N-1]}
    proceed                 {return}
```

We have not compared this program directly with any recursive program and note that this is a quite natural program to write using bounded quantification. The naive recursive program

```
fibonacci(1, 1).
fibonacci(2, 1).
fibonacci(N, F) :-
   fibonacci(N-1, F1),
   fibonacci(N-2, F2),
   F = F1+F2.
```

is, on the other hand, horribly inefficient. The efficient recursive program is tail recursive and around three times faster than the bounded quantification program. We are contemplating the quantifiers needed to write that algorithm using a bounded quantification.

```
fibonacci(1, 1).
fibonacci(2, 1).
fibonacci(N, F) :- N > 2,
   fib_rec(N, 1, 2, F).

fib_rec(3, _, B, B).
fib_rec(N, A, B, F) :- N > 3,
   fib_rec(N-1, B, A+B, F).
```

**Oriented Forest**

The relation *find* holds for two vectors $p_0$ and $p$ if $p_0$ and $p$ are oriented forests representing the same equivalence relation and the depth of $p$ is one.

```
find(P,P) :-
   all(I index_of P, P[I] = P[P[I]]).
find(P0,P) :-
   size(0,P0,N),
   size(0,P1,N),
   all(I index_of P0,
       J^(P0[I] = J,
          (J = P0[J] -> P1[I] = J ;
           otherwise -> P1[I] = P0[J]))),
   find(P1,P).
```

The program can be compiled to the following machine instructions.

```
find/2:
    try_me_else L3          {alternatively try 2nd clause}
      get_value X0 X1       {unify arg 1 and 2}
      put_void X2           {initialize}
      builtin size0 X0 X2   {compute size of P}
      zerop X2 L2           {void loop?}
      put_constant 0 X3     {set I=0}
L1:   put_void X4           {initialize}
      builtin elt X3 X0 X4  {access P[I]}
      builtin elt X4 X0 X4  {unify P[P[I]] with P[I]}
      iterate_int X3 X2 L1  {increment I, loop again?}
L2:   proceed               {return}
L3: trust_me                {last clause}
      put_void X2           {initialize}
    builtin size X0 X3      {compute size of X in N}
    builtin size X1 X3      {verify that size of Y is N}
      builtin size0 X0 X2   {compute size of P0}
      builtin size0 X1 X2   {verify same size of P}
      put_void X6           {initialize}
      builtin size0 X6 X2   {P1 is now an array}
      zerop X2 L7           {void loop?}
      put_constant 0 X3     {set I=0}
L4:   put_void X4           {initialize J}
      builtin elt X3 X0 X4  {access P0[I]}
      inline elt L5 X4 X0 X4{is J=P0[J]?}
      builtin elt X3 X6 X4  {yes, make P1[I]=J}
      jump L6
L5:   put_void X5           {initialize}
      builtin elt X4 X0 X5  {access P0[J]}
      builtin elt X3 X6 X5  {make P1[I]=P0[J]}
L6:   iterate_int X3 X2 L4  {increment I, loop again?}
L7:   put_value X6 X0       {set up arg 1}
      execute find/2        {final call find(P1,P)}
```

**Lessall**

We will give one example that involves lists, the *lessall* relation, which holds for a number $a$ and a list of numbers $l$ if $a$ is less than every number in $l$.

```
lessall(A, L) :-
   all(B in L, A < B).
```

The program can be compiled to the following machine instructions.

```
lessall/2:
    listp X1 L2             {void loop?}
L1: get_list X1             {inspect list L}
    unify_variable X2       {set B to head of L}
    unify_variable X1       {set L to tail of L}
    builtin '<' X0 X2       {verify that A < B}
    iterate_list X1 L1      {if nonempty list, loop again}
L2: get_nil X1              {verify that L is empty}
    proceed                 {return}
```

**Tight Loop**

In the measurements below we have also included an extreme program: a loop or tail recursive program which does nothing at all, iterating 10 000 times.

```
:- all(0 ≤ I < 10000, true).
```

This tells us what the speed-up is in the repetition as such, for a particular implementation. The loop for the bounded quantification is one instruction:

```
L:  iterate_int X0 X1 L     {increment I, loop again?}
```

The loop for the recursive program is rather longer:

```
loop/1:
    switch_on_term L2 L4 L1 L4 L4
L1: switch_on_constant 1 (0 L3) L4
L2: try L3
    trust L4
L3: proceed
L4: builtin '1-' X0 X0
    execute loop/1
```

where the sequence `switch_on_term`, `switch_on_constant`, `builtin '1+'`, `execute` is actually run in each iteration.

**Run times**

Table 7.1 shows run times (on one processor of a SUN 630MP computer) for bounded quantification and recursive programs when applied to some sample data.

1. Inner product: average of 5 000 runs with two 30-element vectors. The figures indicate that the bounded quantification program is approximately 20% faster than the recursive program.

Table 7.1: Run times for sample programs.

| Program | Time (ms) Bounded quantification | Time (ms) Recursion | Relative time |
|---|---|---|---|
| Inner product | 4.8 | 6.1 | 79% |
| Factorial | 3.4 | 7.3 | 47% |
| Integration | 3.1 | 4.6 | 67% |
| Fibonacci | .49 | — | — |
| Forest find | 3.5 | 3.6 | 97% |
| Lessall | .20 | .23 | 87% |
| Tight loop | 7.8 | 47. | 17% |

2. Factorial: average of 5 000 runs with input 1000. The bounded quantification program is more than twice as fast as the recursive program.

3. Integration: average of 15 000 runs with $f(x) = 1/x^2$, $a = 0$, $b = 1$ and 30 intervals. The speed-up is approximately 1/3.

4. Fibonacci: average of 50 000 runs with input 20.

5. Oriented forest: average of 15 000 runs with a completely degenerated tree of depth 29 as input. The speed-up is marginal; presumably enough time is spent in backtracking operations that the gain in the bounded quantifications drowns.

6. Lessall: average of 250 000 runs with a 30 element list (whose elements are all greater than the first argument) as input. The average speed-up is approximately 15%, which is surprisingly high, considering how optimised Warren's abstract machine is for such computations.

7. Tight loop: average of 250 000 runs. The figures say that the speed-up of iteration instructions over ordinary recursion instructions for our implementation is approximately six times.

## 7.9 ADDITIONAL QUANTIFIERS AND ITERATORS

Apart from the quantifiers and iterators introduced in section 7.2 there are also others that may prove useful.

Consider a bounded quantification `array(`$\rho[i]$`, `$\beta[i]$`, `$a$`)`, unifying $a$ with an array containing one instance of $\beta[i]$ for each value of the iteration variable $i$ such that $\rho[i]$ is true. If the range expression $\rho[i]$ contains a test then the array may contain fewer elements than values produced by the iterators. For example, the goal `array(1 ≤ I ≤ 17, prime(I), I*3, A)` (assuming that `prime` is **`true`** for primes) would unify `A` with an array `(/ 3, 6, 9, 15, 21, 33, 39, 51 /)`.

In a Prolog system with finite sets as a data type it would clearly be interesting to have a `set` quantifier; the value of a bounded "setof" quantification would be a set of all instances of its body (interpreted as a term) such that the range formula is **`true`**. For example, the goal `set(4 ≤ I ≤ 17, prime(I), I, S)` would unify `S` with the set `{5, 7, 11, 13, 17}` (if that is the syntax for a set itemization).

Similarly one may think of a `list` quantifier which is analogous to the `array` and `set` quantifiers. (Incidentally, if the requirement that the range formula must contain an iterator as a conjunct is dropped we obtain the equivalent of Prolog's `bagof` "predicate".) The set and list quantifications are quite similar to the set and list comprehensions that can be found in some functional programming languages [197, 104].

A `count` quantifier, which gives the number of elements for which the range formula is **`true`**, can be included but could also be simulated by `sum(`$\rho[i]$`, 1, Z)`. Nevertheless, the meaningfulness of such a quantifier implies that one might consider extending the syntax to include bounded quantifications with no body, for such quantifiers.

We are considering ternary variants of the list head/tail iterators, which produce list heads/tails together with consecutive integers that identify the positions of the heads/tails in the list. Those positions are expensive to compute 'in retrospect' but can be obtained at little cost while the list is traversed. Further experience will show if such constructs are motivated.

It seems reasonable to allow `elt(`$i$`,`$a$`,`$x$`)` and `arg(`$i$`,`$t$`,`$x$`)` as iterators producing the indices of the term or array, together with the values at the corresponding positions. The `arg` iterator should also accept an atomic value, producing no values.

A Prolog system with finite sets could have iterators for set membership and subsethood, where the iteration variable ranges over the elements of the set and its powerset, respectively. The actual syntax would depend on the syntax for the corresponding concepts in the Prolog system.

In a Prolog system having a type system with user-defined discrete types (enumeration types) it would make sense to have iterators corresponding to these.

## 7.10 RELATED WORK

Voronkov [211, 212] has (independently from us) studied bounded quantifiers in logic programming, although from a somewhat different perspective. He introduces a class of generalized logic programs (where program clauses may contain bounded quantifications with list head/tail iterators) and a complete variant of resolution called SLDB-resolution for running them. He also mentions previous work in Russia.

Kluźniak has (also independently) proposed a specification language called SPILL [115] that includes certain bounded quantifications with integer ranges. Specifications written in SPILL are executable by translating them to Prolog, according to a set of translation rules. However, the quantifications are translated essentially as by Lloyd & Topor (cf. below). If the target Prolog system had bounded quantifications, we expect that the specifications could be eexecuted more efficiently and that the "generators" for quantified variables in SPILL could be extended.

Some authors have studied the use of bounded quantifiers with sets, for example in SETL [170] and {log} [70].

Barklund & Hill [23] have studied how to incorporate restricted quantifications and arrays in Gödel [101], while Apt [10] has studied how bounded quantifications and arrays could be used also in constraint based languages.

Lloyd & Topor [128] have studied transformation methods for running more general quantifier expressions, although their method will 'flounder' for some examples that can be run using this method (Lloyd, personal communication). Sato & Tamaki [167] have an interpreter that will run more general quantifier expressions although the method is currently not so efficient (Sato, personal communication). In comparison our approach is only applicable to range-restricted formulas, but is quite efficient for that case.

The methods by Bevemyr, Lindgren & Millroth [135, 33] and Meier [133] for compiling recursive programs to iterative code are also relevant. Millroth's method is based on an analysis of variable binding patterns while Meier's method, as presented, seems somewhat more ad hoc. For tail recursive programs it seems to us that Meier gets code which is quite similar to that of ours for the corresponding bounded quantification (except that we have defined a small collection of new abstract machine instructions while his code is a mix of WAM, C, etc.). Meier also considers "backtracking" iteration, a subject we have not discussed here (compilation of bounded existential quantifications). It seems to us that the instruction set we use would be appropriate as a base also for Millroth's and Meier's methods.

The work on array comprehensions in Haskell and their compilation [5] also seems to be relevant for our work, although their context is lazy functional programming.

The Common LISP language [184] (and some earlier LISP dialects), as well as Standard ML [91], contain iteration, mapping and reduction constructs that in some cases resemble ours.

The idea for using bounded quantification in logic programming was inspired by Tennent's proposed use of them in ALGOL-like languages [189, 190], and also by an exposition of Gries [82].

Hermenegildo & Carro [98] discuss how parallel execution of bounded quantifications relates to more traditional (AND) parallel execution of logic programs.

Finally, we should mention that transforming recursive programs to iterative programs is an activity that has been studied extensively in computing science. This often involves tabulation techniques [30, 80, 36] and has also been applied in logic programming [218, 18]. We are interested in using bounded quantification and arrays as a target language for transformation of recursive logic programs to iterative logic programs with tabulation.

## 7.11 CONCLUSIONS AND FUTURE WORK

An extension of Prolog with bounded quantifications and arrays has been proposed, together with a detailed implementation.

Our examples illustrate that many of our bounded quantification programs are more elegant than the corresponding recursive programs, although we have also seen several algorithms that we do not know how to express elegantly using bounded quantifications. (This was expected, because some algorithms are expressed more elegantly using recursion, or have loops with complicated termination tests, also in imperative programming languages.)

The speed-ups observed for bounded quantification over recursion are satisfactory. The overhead in loops is much lower for bounded quantifications and the experiments show, as we expected, that we get big or very big speed-ups (up to 50%) for small loop bodies and small or very small speed-ups for big loop bodies.

Even though we consider bounded quantifications justified in sequential logic programming languages, for the reasons explained above, another important motivation for studying bounded quantification is their potential for parallelization. We have begun investigating in detail how to incorporate them in parallel logic programming languages and run them on various parallel computers. We think that bounded quantifications will be an important construct for elegant and efficient logic programming in the future.

## 7.12 ACKNOWLEDGEMENTS

We want to thank the referees of LPAR'93 and our colleagues for valuable comments on earlier versions of this paper, and Manuel Hermenegildo for valuable discussions. J. Ba. is forever in debt to Anna and Ellin for their patience.

# A Simple and Efficient Copying Garbage Collector for Prolog

Johan Bevemyr, Thomas Lindgren

In *Proc. PLILP'94*, Springer LNCS 844, 1994.

We show how to implement efficient copying garbage collection for Prolog. We measure the efficiency of the collector compared to a standard mark-sweep algorithm on several programs. We then show how to accomodate generational garbage collection and Prolog primitives that make the implementation more difficult.

The resulting algorithms are simpler and more efficient than the standard mark-sweep method on a range of benchmarks. The total execution times of the benchmark programs are reduced by 4 to 11 percent.

## 8.1 INTRODUCTION

Automated storage reclamation for Prolog based on Warren's Abstract Machine (WAM) [214] has several difficulties. Let us consider the architecture of a typical WAM: 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, while variable bindings, which are implemented as assigning a cell representing the variable, 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.

A garbage collector for Prolog might thus retain the segment ordering to enable fast storage reclamation on failure by deallocating a segment allocated after the topmost choice point was created. Furthermore, since there are primitives (such as `@</2`) that compare variables, e.g., by creation time, most systems elect to preserve the heap ordering of data after garbage collection.

Garbage collection is done by starting at a set of *root pointers*, such as registers and the local stack, and discovering what data are reachable from these pointers, or *live*. Memory is reclaimed by compacting the live data [144], copying them to a new area [49] or putting the dead data on a free list. Memory allocation can then be resumed.

## 8.2 RELATED WORK

Prolog implementations such as SICStus Prolog use a mark-sweep algorithm that first marks the live data, then compacts the heap. We take the implementation of Appleby et al. [9] as typical. This algorithm works in four steps and is based on the Deutsch-Schorr-Waite [169, 55] algorithm for marking and on Morris' algorithm [144, 55] for compacting.

1. All live data are marked through roots found in registers, choice points, environments, and value trail entries (entries in the trail where the old value have been recorded, e.g., as a result of using `setarg/3`). A live tree is marked using a nonrecursive pointer-reversing algorithm that does not require any extra space to operate.

2. The registers, choice points, environments, and the trail are examined for references into the heap. All such references are put into reallocation chains, with the heap cell as root, to be updated when the heap cell is moved.

3. The heap is scanned upwards and all upward references are put into reallocation chains in order to be updated when the cell they refer to is moved.

4. The heap is scanned downwards and all marked data are moved to their new locations. All references to a moved object are found through the reallocation chains and updated. All references downward are also

put into reallocation chains so that they may be updated when the object further down the heap is moved.

Touati and Hama [196] developed a generational copying garbage collector. The heap is split into an old and a new generation. Their algorithm uses copying when the new generation consists of the top most heap segment, i.e., no choice point is present in the new generation, and no troublesome primitives have been used (primitives that rely on a fixed heap ordering of variables). For the older generation they use a mark-sweep algorithm. The technique is similar to that described by Barklund and Millroth [27] and later by Older and Rummell [150].

We show how a simpler copying collector can be implemented, how the troublesome primitives can be accomodated better and how generational collection can be done in a simple and intuitive way. However, our view is also more radical than theirs. Where Touati and Hama still wish to retain properties such as memory recovery on backtracking, we take a more radical approach: ease of garbage collection is more important than recovering memory on backtracking. We show below that memory recovery by backtracking is still possible, and that the new approach in practice recovers approximately as much garbage by backtracking as the conventional approach.

Bekkers, Ridoux and Ungaro [28] describe an algorithm for copying garbage collector for Prolog. They observe that it is possible to reclaim garbage collected data on backtracking if copying starts at the oldest choice point (bottom-to-top). However, their method has several differences to ours.

- Their algorithm does not preserve the heap order, which means primitives such as `@</2` will work incorrectly. They do not indicate how this problem should be solved.
- Their algorithm (the version that incorporates early reset) copies data twice, while our algorithm visits data once and then copies the visited data. We think our approach leads to better locality of reference. However, we have not found any published measurements of the efficiency of the Bekkers-Ridoux-Ungaro algorithm.
- Variable shunting is used to avoid duplication of variables inside structures. This may introduce new variable chains, as shown in Appendix A. We want to avoid this situation.

Their algorithm does preserve the segment-structure of the heap (but not the ordering within a segment). Hence, they can reclaim all memory by backtracking. In contrast, our algorithm only supports partial reclamation of memory by backtracking. Our measurements indicate that this is sufficient: the copying algorithms we describe do not reclaim appreciably less memory on backtracking than the standard mark-sweep algorithm on the measured benchmarks.

Appel [6, 7] describes a simple generational garbage collector for Standard ML. The collector uses Cheney's garbage collection algorithm, which is the basis of our algorithm as well. However, his collector relies on assignments being infrequent. In Prolog, variable binding is assignment in this sense. Our algorithm handles frequent assignments efficiently.

Sahlin [162] has developed a method that makes the execution time of the Appleby et al. [9] algorithm proportional to the size of the live data. The main drawback of Sahlin's algorithm is that implementing the mark-sweep algorithm becomes more difficult, not to mention guaranteeing that there are no programming errors in its implementation. To our knowledge it has never been implemented. We also believe that Sahlin's algorithm is not as efficient as ours since it requires an extra pass over the live data, beyond the passes in the Appleby algorithm. Since our algorithm is almost 70 % faster than the Appleby algorithm even when the heap is filled with live data, it is unlikely that Sahlin's algorithm will be more efficient than ours.

## 8.3 ALGORITHM

We assume the standard term representation of WAM [214]. Our algorithm requires the existence of two tag bits for each cell on the heap, reserved for the use of the garbage collector. These tag bits may either be stored in each cell or in some separate area. One of the bits is used for marking copied cells as forwarded, the other is used for indicating that a cell appears inside a live structure.

### Avoiding the heap ordering

Variables in the WAM are represented as self-referring heap cells. The WAM uses the location of a variable for deciding if trailing is required when binding the variable. Hence, variables should not move out of their heap segments. Since our algorithm does not preserve heap segments we must find another solution.

Our solution is simply to trail bindings of variables copied during the last garbage collection. The method to do this efficiently is described in Section "Recovering memory on backtracking". Bindings to the surviving variables from the topmost heap segment will be trailed unnecessarily (as compared to the compacting approach), but other bindings will not be affected. The unnecessary trail entries are deleted by the next collection.

### Mark-and-copy

The copying collector is a straightforward adaption of Cheney's algorithm [49] and works in three phases. The algorithm allows the standard optimizations of early reset. The old data reside in `fromspace` and are evacuated into `tospace`.

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 `fromspace`, do the following:

   (a) Scan backward (towards lower addresses) until an unmarked cell is found.

   (b) Scan forward and evacuate marked cells into `tospace` 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.

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.

The collector thus visits (and writes) the data once, then writes the copy in `tospace`. We believe locality of reference to be quite good: in the second pass, the marked data will already reside in the cache if the data are sufficiently small.

If we do not mark all cells that occur inside live data structures then duplication of cells could occur. Suppose we have both a reference to a variable in a structure and a reference to the structure, e.g., see Figure 8.1. Suppose we copy the variable before the structure; then we would introduce an extra reference, e.g., see Figure 8.2. This is undesirable since the result of doing garbage collection might then be that more space is required! To solve this, we mark all cells that occur in live data structures. When a marked cell is copied the enclosing structure is also copied by step 2.

### Recovering memory on backtracking

A compacting collector preserves the heap segments (see Figure 8.3) and

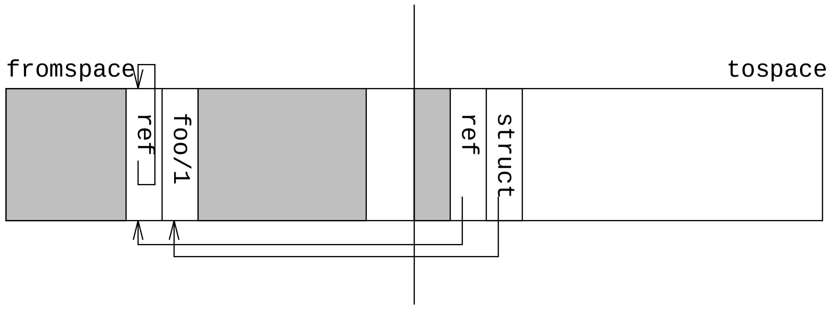


Figure 8.1: An internal cell is referred to twice, both directly by a variable and indirecly through a structure.

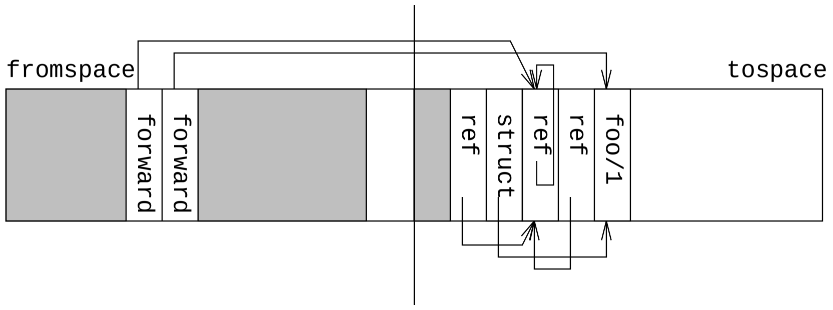


Figure 8.2: As a result of copying without marking internal cells some cells might be duplicated.

entire segments can be deallocated on backtracking. In general, a copying algorithm cannot recover memory on backtracking since the heap no longer preserves the required stack ordering. However, our algorithm can still recover some garbage by resetting the heap pointer, just as in a standard WAM implementation.

We note that *between* collections, memory is allocated just as in a WAM. Using this observation, we can arrange to recover memory allocated after the last collection on backtracking. After a collection, we set the saved heap top of all choice points to the top of the heap space, making this one segment (see Figure 8.4). Terms allocated after this segment can be reclaimed upon backtracking. In this way, we retain most of the advantages of resetting the heap upon backtracking without having to tailor our runtime system and garbage collector to ensure this property at every point. Precisely the same test for trailing a binding can be used as in a standard WAM. Collection may also split a single segment into two, which leads to extra trailing. We measure the efficiency of our system below.

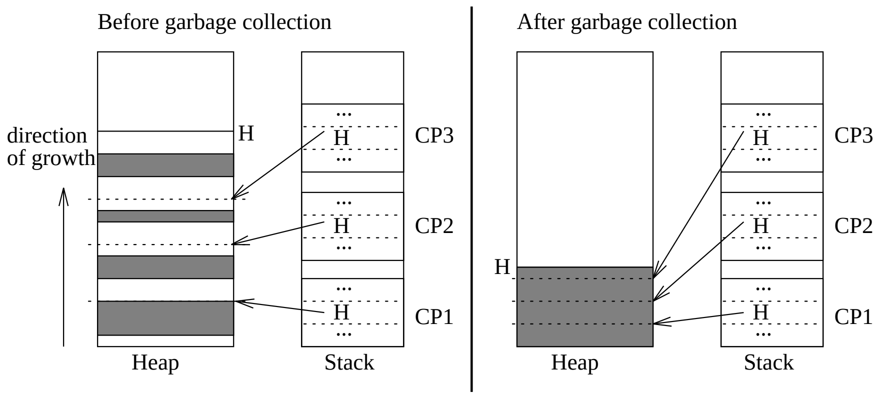


Figure 8.3: Saved heap top pointers (H) in choice points before and after **compacting** garbage collection. Heap segments are preserved.

### Handling troublesome primitives

Touati and Hama recognized that the generic comparison operators, such as `@</2`, required that variables preserve their relative ordering (since variable ordering is usually the comparison criterion when two unbound variables are compared). Their method was either to disable copying collection when these primitives were used and revert to using a compacting collector, or to generate identifiers for variables when needed. The identifiers were to be kept in a hash table to be updated when variables were relocated.

Others have proposed to associate a creation time with each variable. Our experience is that timestamps add a runtime overhead of approximately 5% in an emulator-based WAM implementation.

Our solution retains the use of copying collection, while requiring a small modification to the runtime system. When variables are compared, we arbitrarily order them if unordered. This is done by binding unordered variable cells to new variable cells on a small *compared-variable* stack (cv-stack), see Figure 8.5. Once this is done, we can just compare addresses. The binding is not trailed. (An unbound variable is unordered if it does not reside on the cv-stack and ordered if it is on the cv-stack.)

Once a variable is ordered it resides on the cv-stack. Subsequent unordered variables will be 'greater' than ordered variables when compared, since they are pushed on the cv-stack (where the ordering is kept) when the comparison occurs. Using compacting collection on the cv-stack retains the variable ordering (though the copying collector must take care not to migrate variables residing on the cv-stack), while dead variables disappear.

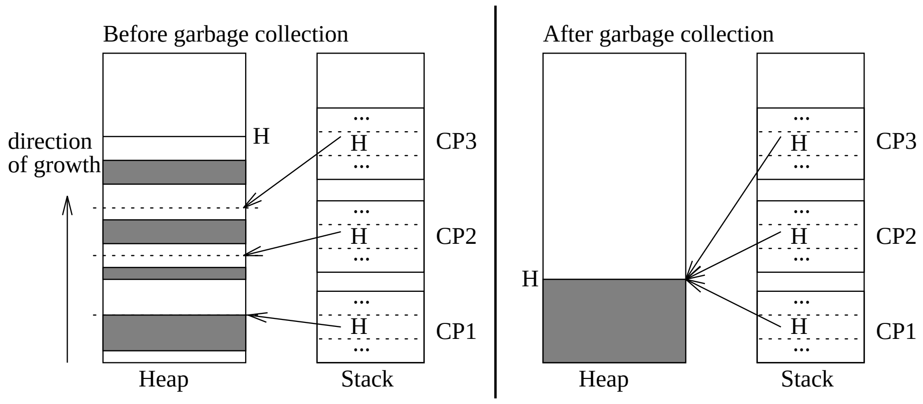


Figure 8.4: Saved heap top pointers (H) in choice points before and after **copying** garbage collection. After garbage collection all segments have been merged into one single segment. Note that only the active heap is shown here—the live data have been copied from the old heap to the new heap.

The space cost is proportional to the number of unbound variables compared by generic comparison operations. Naturally, if most of the live unbound variables have been compared in this way, the collector will have to spend more time in compacting the cv-stack. We believe this situation to be rare.

## 8.4 INTRODUCING GENERATIONAL GARBAGE COLLECTION

Generational garbage collection [121, 7] 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.

The difference from standard copying collection is that the collection roots also include the pointers from the older to the younger generation. In languages such as SML, most objects are immutable, and assignments that may cause cross-generational pointers can be compiled to special code that registers such a pointer if it appears.

In Prolog, there is a high incidence of assigning already created objects, so such a solution is likely to be expensive. Variable bindings involve assignment. In fact, we can arrange so that only *trailed* bindings may be

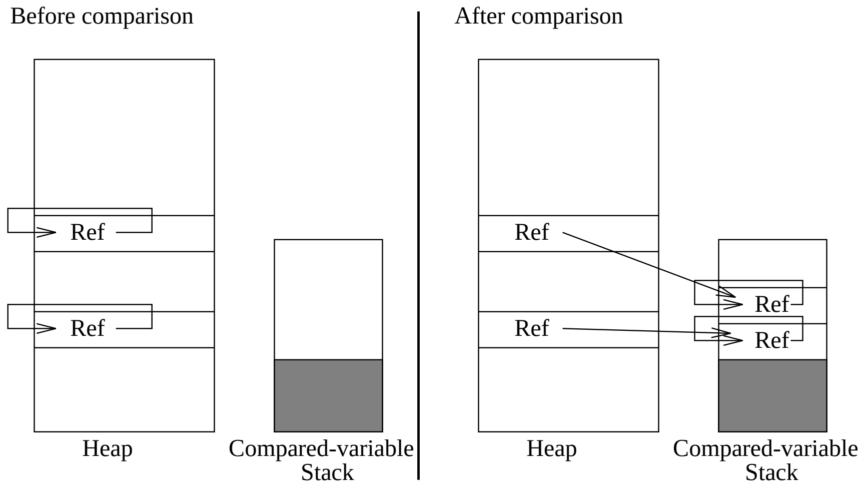


Figure 8.5: Compared variables are bound to variables on a compared-variable stack. This way their relative order is preserved.

cross-generational, by setting the limit where trailing occurs appropriately, see Figure 8.6. Usually, this limit is the start of the topmost heap segment, but this is not required: we can set it in the interior of the topmost segment at the cost of doing unnecessary trailings. This may be useful if the computation is deterministic but still has a large amount of live data.

We assume that objects are tenured (moved to the old generation) if they survive a collection of the new generation. Now, we can find cross-generation pointers by examining all new trail entries since the last garbage collection. These pointers point out root pointers from the older generation. Since the tenuring threshold is one, the old part of the trail need not be scanned; it can refer only to the old generation. If we were to allow several minor collections before tenuring, the cross generation pointers must be recorded for subsequent use, to avoid scanning the trail repeatedly.

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 result in a runtime cost for using generational garbage collection. In Prolog this overhead is already present in the form of trail tests and there is no extra runtime penalty for using generational collection.

## 8.5 EVALUATION

We have implemented a standard mark-sweep algorithm [9] and compared it to our copying algorithms. All garbage collection algorithms have been

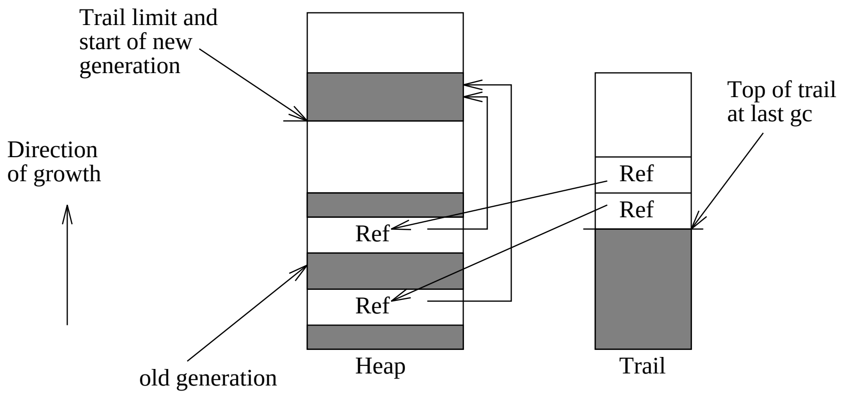


Figure 8.6: The limit for trailing is set in such a way that all cross-generational references are recorded on the trail. In the presence of a choice point in the new generation the trail limit is set as usual.

implemented in the same system, a sequential version of Reform Prolog. All algorithms implement early reset.

The TSP program implements an approximation algorithm for the Travelling Salesman Problem. A tour of 60 cities was computed. The MATCH program implements a dynamic programming algorithm for comparing, e.g., DNA-sequences. One sequence of length 32 was compared to 100 other sequences. BOYER is the Boyer-Moore theorem prover adapted by Evan Tick to Prolog. This program is part of the Berkeley benchmark suite.

BIG is a program that allocates a large data structure and then forces garbage collection 1000 times while the data structure is still live. The program is intended to compare the copying collectors with mark-sweep when the heap is filled with live data, i.e., when copying and compaction are working on a similar amount of memory and data.

When executing the TSP, MATCH, and BOYER programs we gave each version of the emulator (mark-sweep and copy) an equal amount of memory. This meant that the emulator using copying garbage collection used a heap that was half the size of the mark-sweep heap. One might argue that since modern computers have virtual memory it is reasonable to let the copy version temporarily allocate twice as much space as the mark-sweep version. However, we have found that even the size of the virtual memory (actually the size of the swap space) can be a limiting factor.

In our measurements of the generational collector, the new generation is given half the remaining free memory in the semi-space, after a minor collection. When the size of the new generation is less than 20K, a major collection is done.

When we executed the BIG program we gave the copying collectors twice the memory of the mark-sweep collector so that the collected data structures could have the same size. The from-space and mark-sweep heap thus were of the same size. This was done in order to compare the timing of a single collection, rather than the entire execution.

All times are in seconds user time.

| Program | Heap | Mark-Sweep | | Copy | | Gen. Copy | |
|---|---|---|---|---|---|---|---|
| | size | gc | run | gc | run | gc | run |
| MATCH | 750K | 12.0% | 22.02 | 2.6% | 20.01 | 0.6% | 19.82 |
| | 1500K | 11.2% | 21.89 | 1.4% | 19.70 | 0.3% | 19.64 |
| | 2250K | 10.4% | 21.43 | 0.7% | 19.96 | 0.1% | 19.87 |
| | 3000K | 10.0% | 21.64 | 0.6% | 19.89 | 0.05% | 19.61 |
| TSP | 750K | 5.3% | 54.73 | 0.9% | 51.99 | 0.1% | 51.11 |
| | 1500K | 4.8% | 54.62 | 0.4% | 52.12 | 0.1% | 51.81 |
| | 2250K | 4.7% | 53.75 | 0.3% | 51.96 | 0.08% | 51.48 |
| | 3000K | 4.6% | 55.38 | 0.2% | 51.77 | 0.07% | 51.65 |
| BOYER | 750K | 15.8% | 4.61 | 15.7% | 4.51 | 8.4% | 4.38 |
| | 1500K | 8.7% | 4.25 | 7.1% | 4.23 | 5.6% | 4.09 |
| | 2250K | 12.4% | 4.58 | 5.9% | 4.04 | 4.9% | 4.06 |
| | 3000K | 0% | 4.04 | 3.0% | 3.98 | 4.9% | 4.02 |
| BIG | 1500K | 100% | 11.22 | 100% | 6.34 | — | — |

The next table shows the difference in execution times for the three algorithms (using 1500K memory). The improvement of the total execution time when using a copying collector ranges from 4 to 11 percent.

We also measured how many times the different garbage collectors were invoked.

| Program | Heap | Number of garbage collections performed | | |
|---|---|---|---|---|
| | size | Mark-Sweep | Copy | Gen. Copy |
| MATCH | 750K | 25 | 51 | 130 |
| | 1500K | 12 | 25 | 48 |
| | 2250K | 8 | 16 | 30 |
| | 3000K | 6 | 12 | 22 |
| TSP | 750K | 22 | 46 | 113 |
| | 1500K | 11 | 22 | 51 |
| | 2250K | 7 | 14 | 33 |
| | 3000K | 5 | 11 | 25 |
| BOYER | 750K | 3 | 8 | 30 |
| | 1500K | 1 | 3 | 7 |
| | 2250K | 1 | 2 | 4 |
| | 3000K | 0 | 0 | 3 |

All times are in milliseconds user time.

| Program | GC time/run time | | |
|---|---|---|---|
| | Mark-Sweep | Copy | Generational Copy |
| MATCH | 2460/21890 | 280/19700 | 50/19640 |
| TSP | 2600/54620 | 220/52120 | 50/51810 |
| BOYER | 370/4250 | 300/4230 | 230/4090 |

Note that the total execution times are sometimes shorter when garbage collection is performed. We believe this to be due to improved data locality due to copying.

| Program | Memory allocated | Memory reclaimed on backtracking | | |
|---|---|---|---|---|
| | | Mark-Sweep | Copy | Generational |
| MATCH | 17770K | 793 (4%) | 793 (4%) | 793 (4%) |
| TSP | 18895K | 0 (0%) | 0 (0%) | 0 (0%) |
| BOYER | 4187K | 1798 (43%) | 1798 (43%) | 1798 (43%) |

Approximately the same amount of data is reclaimed on backtracking with all three algorithms. We believe the reason for this is that most of the memory is reclaimed during shallow backtracking.

We also measured the amount of extra trailing imposed by the copying collectors.

| Program | Number of trail entries | | | Ratio | |
|---|---|---|---|---|---|
| | Mark-Sweep | Copy | Gen. | Copy/Mark | Gen/Mark |
| MATCH | 112582 | 112767 | 112864 | 1.0016 | 1.0025 |
| TSP | 10560 | 10568 | 10568 | 1.0008 | 1.0008 |
| BOYER | 108352 | 108450 | 108441 | 1.0009 | 1.0008 |

Clearly, the extra trailing performed by the copying algorithms is insignificant compared with the total amount of trailing in our benchmark programs. The above measurements were made using a heap size of 750K bytes.

## 8.6 CONCLUSION

We have described a method for adapting conventional copying garbage collection to Prolog and how to add generational collection to this algorithm. Three problems have been solved, leading to efficient copying and generational copying collectors.

The first problem is interior pointers, which can lead to duplication of data if copied naively. Our method correctly handles interior pointers by marking, then copying data.

The second problem is that copying collection does not preserve the heap ordering. In theory, this means memory cannot be reclaimed by backtracking, and that bindings in the copied area must always be trailed (rather than occasionally).

Our collector exploits that data allocated since the last collection still retain the desired heap ordering. Hence, memory allocated after the last collection can still be reclaimed by backtracking. Our measurements show that our copying algorithm recovers as much memory by backtracking as a conventional ("perfect") mark-sweep algorithm on a range of realistic benchmarks.

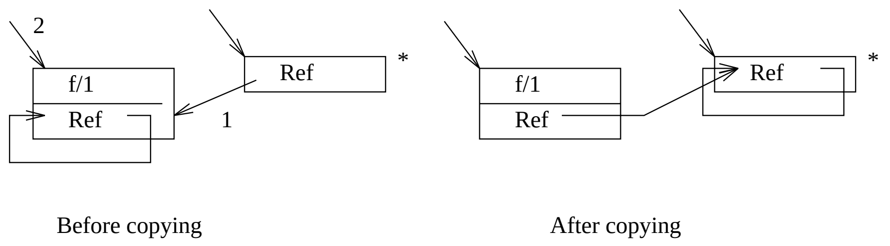
We have also measured the amount of extra trailing due to losing the order of the heap. This was negligible: less than one-quarter of a percent of the total number of trailings at most. We conclude that copying collection is a viable alternative to the conventional mark-sweep algorithm for Prolog.

Finally, we also showed how to extend the copying algorithm to generational collection. The crucial insight is that pointers from the old generation (in a two-generation system) can be found by scanning the trail. By adapting the trailing mechanism, we get an almost-free write-barrier. The only extra cost is some unnecessary trailings in certain situations. This cost is again negligible for our benchmarks.

## 8.7 ACKNOWLEDGMENT

We thank Ulrich Neumerkel for his comments on this paper, in particular on whether our method to deal with troublesome primitives was correct. We also thank Mikael Pettersson and the anonymous referees for their comments.

## 8.8 APPENDIX A



Variable shunting collapses a chain of pointers into a single cell when they are all in the same segment. Assume that pointer (1) in the figure is copied first. The chain of two pointers is collapsed into a single cell $*$ due to variable shunting. Subsequently, reference (2) copies the structure, which yields the situation shown after copying. The number of cells remains constant, but a new reference chain has appeared.

# A Parallel Generational Copying Garbage Collector for Shared Memory Prolog

Johan Bevemyr

A shorter version of the paper appeared in *ILPS Workshop on Parallel Logic Programming*, 1995

We present a parallel generational copying garbage collection scheme for Prolog. The algorithm use a mark and copy technique for handling internal pointers.

We describe how the collector is parallelised and load balanced.

The new garbage collector shows up to 5 times speed up on 12 processors for programs with large amounts of live data.

## 9.1 INTRODUCTION

Our experience is that Prolog programs spend on average 15% of their execution time in garbage collection, when run in SICStus Prolog. Some programs [1] 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 $1/0.15 = 6.67$.

[1] One of our abstract interpreters spent 70% of its execution time in garbage collection when analysing a semi-group program in SICStus.

It has been shown how a generational copying garbage collector can be used for sequential Prolog [35]. It has also been shown how to parallelise copying garbage collection for shared memory multiprocessors [3]. We present a collection scheme that combines these techniques resulting in a parallel generational copying garbage collector for Prolog. The collector is implemented and evaluated in Reform Prolog, a emulator based recursion parallel Prolog implementation.

The new collector has the following properties: two mark bits are used for each word, three barrier synchronisations are needed for each run of the collector, execution time is proportional to the size of the live data, and internal pointers and circular structures are handled.

### Preliminaries

Let us consider the architecture of a typical WAM [214]: 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.

## 9.2 SEQUENTIAL COPYING GARBAGE COLLECTION

A significant number Prolog garbage collectors are based on mark-slide algorithms [9]. The reason is that most Prolog implementations rely on that data maintains their relative positions throughout the execution. This has the advantages that:

1. Data allocated on the heap can be *instantly reclaimed* on backtracking, in a stack like fashion.
2. The location of a variable can be used for deciding if trailing is required when binding the variable.

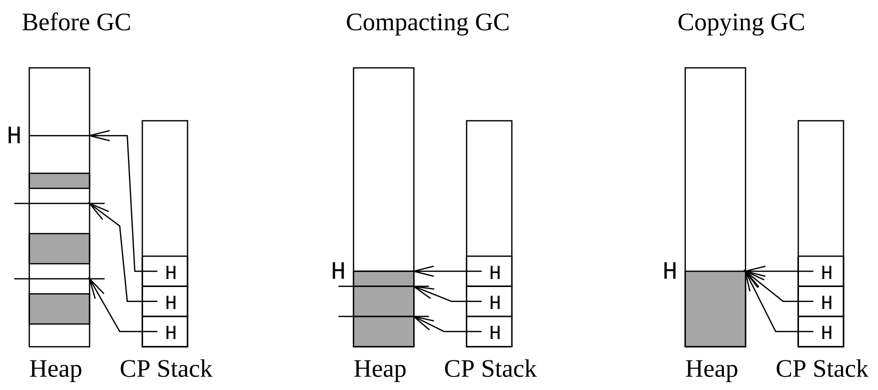


Figure 9.1: Saved heap top pointers (H) in choice points before and after **compacting** and **copying GC**. Heap segments are preserved using compacting but not using copying collection.

3. Variables can be ordered by their relative positions in the heap, for example, when generic comparison operators such as `@</2` are used.

The main concern with mark-slide collectors is that the collection time is proportional to the size of the total memory area.

The time complexity can be made proportional to the size of the *live* data either by linking together and sorting the live data [162] in a mark-slide collector (actually NlogN (N = live data)), or by using a copying collector [28, 35, 67].

We use a copying collector since it is more amenable to parallelisation. Copying collectors have the disadvantage that the relative positions of the data are not preserved.

We briefly discuss how instant reclaiming and trailing can be handled in a sequential copying collector. A richer discussion can be found in Bevemyr and Lindgren's [35] paper.

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

## Instant Reclaiming

A compacting collector preserves the heap segments (see Figure 9.1) and entire segments can be deallocated on backtracking.

Bekkers, Ridoux and Ungaro[28] 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. The technique has been further improved by Demoen, Engels and Tarau [67].

This technique requires barrier synchronisation after copying each choice point. It relies on that all data reachable from one choice point is copied (or processed) before the data reachable from the next is touched. Consequently, it is not possible to use the above technique in a parallel setting without imposing strict synchronisation between copying each choice point, severely limiting the available parallelism and introducing synchronisation overheads.

Bevemyr and Lindgren[35] 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 9.1).

### Trailing

In WAM, whether to trail a binding or not is decided by comparing the variables position in the heap with the current heap segment limit. Collapsing all segments into one will result in that all bindings of surviving variables are trailed. This will result in some extra trailing which can be removed during the next garbage collection. Bevemyr and Lindgren [35] showed that the extra trailing generated by this scheme is insignificant.

An alternative to trailing all bindings in the old segment is to associate a time stamp with each variable. The time stamp may then be used for deciding whether to trail or not. Such time stamps are already present in some Prolog implementations for other reasons.

Quintus Prolog solve this by sorting the heap to preserve the memory (moving portions of the data) whenever a new segment is allocated. This is both slow and cumbersome, and not possible in a parallel setting.

### Bevemyr and Lindgren's Mark-Copy Algorithm

Bevemyr and Lindgren's [35] copying collector is a straightforward adaption of Cheney's algorithm [49] and works in three phases; 1) marking all live data, 2) copying the live data, 3) updating external references. 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 implementations since they tend to allow external references into structures, e.g., to variables inside structures.

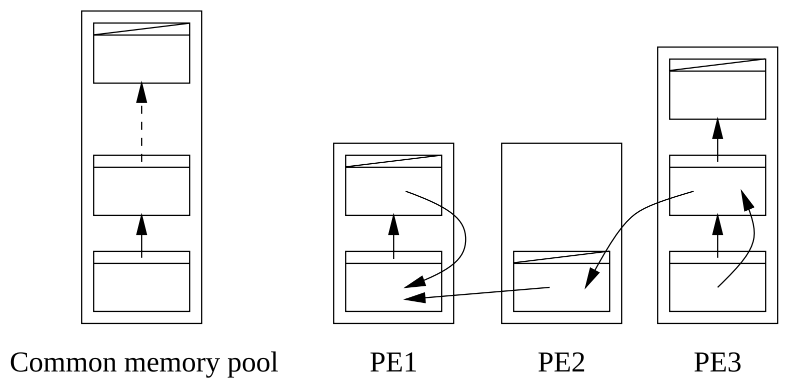


Figure 9.2: Memory is divided into segments and stored into a common pool. The processing elements allocate segments from the pool depending on their individual requirements.

These externally referenced cells can be reached and copied before the surrounding structure, resulting in duplicated memory for a single variable. 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.

Their 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 Deutsch-Schorr-Waite[169] pointer reversal algorithm.

An alternative solution would be to eliminate internal variables altogether. This would simplify the algorithm significantly: no marking phase would be necessary, one GC bit less could be used, and one barrier synchronisation point would disappear. However, Foster and Winsborough [78] have shown that eliminating internal variables results in approximately 28% higher memory consumption and about 15% increase in execution time due to increased dereferencing and memory allocation.

## 9.3 PARALLEL GARBAGE COLLECTION

Khayri Ali [3] has proposed an elegant scheme for parallel copying garbage collection on shared memory multiprocessors. The memory is divided into segments. Initially all segments are stored in a common pool. The processing elements (PEs) allocate segments from the pool and link them into their private memory area (see Figure 9.2). References between the areas are allowed.

Garbage collection is initiated when the number of free segments decrease 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.

- Duplicating structures is avoided by locking each substructure before copying.
- Work sharing is enabled by traversing the data depth first, creating pointer chains of remaining work. The chains can be divided and distributed to idle PEs.

## 9.4 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 [33, 34] is an example of this kind of implementation.

### Segmented Memory

To use Ali's scheme the heaps are divided into segments. This is desirable for other reasons as well, e.g, 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.

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. Garbage collection is initiated when the number of free segments decrease below a given limit.

   A special trail entry is used to record that a new segment has been allocated. The segment is deallocated on backtracking.

3. Variables are no longer ordered by their positions in memory. Deciding if a variable binding must be trailed or not cannot be done by comparing the variable's 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, resulting in a slight overhead (1-4%) [32] in an emulator based implementation.

   Several people, including anonymous referees, have pointed out that 1-4% is a surprisingly low figure. However, these time stamps should not be confused with the time stamps frequently used in AND- and OR-parallel implementations for managing multiple binding alternatives, e.g. the time stamps used by Tinker and Lindstrom [195].

   Our time stamps are implemented using unbound tagged variables where the value field stores a time stamp. The time is incremented when choice points are created. The time stamp is used for determining the age of unbound variables instead of using their address, which is usually done.

   The overheads originates from an occasional increment of the time and a slightly different dereferencing algorithm due to the unbound tag (we need to check for an unbound tag instead of a self referencing variable). All other operations remain the same.

Our measurements using two versions of Reform Prolog, with and without time stamps and segmented heap, show that the total overheads for segmenting the heap and using an unbound tag are small. We measured a ray tracer (3% speedup!), a genetic matching program (7% slowdown), the Reform compiler (9% slowdown), and naive reverse (8% slowdown). These overheads can be further reduced by using caching techniques for segment allocation. The speed and implementation technique of Reform Prolog is comparable to emulated SICStus Prolog.

### Modifications to the Sequential Algorithm

The sequential algorithm has to be modified in the following ways to run in parallel (the full parallel algorithm is described later):

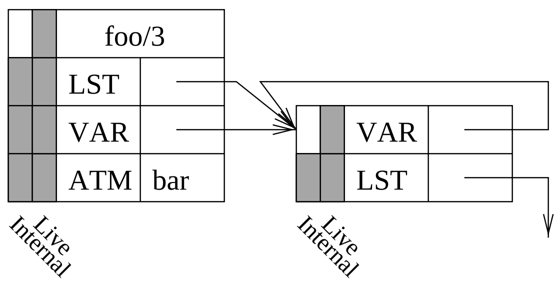


Figure 9.3: Data in `from-space` marked by the marking algorithm for the parallel Prolog collector. Internal elements are marked both as **L**ive data and as **I**nternal cells.

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

   Our 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 the tail recursive optimisation, described below, is applied. Let us consider the worst case: a structure of nested lists occupying the entire from-space. Each cons cell occupies two words. Each pushed entry also occupies two words. The stack would, at its worst, occupy the entire to-space.

2. A potential problem is that copying large chunks of surrounding live data may result in bad segment utilisation. One solution is to limit the size of the chunks to a single structure. This is done by only marking live cells internal to a structure and copy those when an internal cell is encountered. Structure boundaries can be identified by not marking the functor in a structure and the CAR cell in a CONS.

   CONS cells present a problem in that the first cell can be referenced both as a variable and as a list (see Figure 9.3), and it is not marked as internal being the first in the structure. 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.

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

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

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

   (c) Before terminating the gc. This is to ensure that all data have been copied before any worker start accessing it.

4. Existing techniques for early reset and variable shunting [46, 120] optimisations cannot be used. They rely on heap segments to be either marked or copied in choice point order. This requires barrier synchronisation after marking and copying each choice point, i.e., almost sequentialising the collector.

   Bekkers et al.'s scheme for maintaining heap segments cannot be used for the same reason.

**The Parallel Algorithm**

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 (each worker has a separate stack for this). All structure cells, except the first in each structure, are marked as internal-cells. All live data are also marked as *live* to detect circular structures.

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

3. Update the trails and the choice points, removing segment deallocation entries.

4. Deallocate from-space.

Barrier synchronisations are used before 1 and 2, and after 4.

*Marking* Two tag-bits are needed: one for marking **I**nternal cells and one for marking all **L**ive cells. Two variables are used; one to keep track of the *Current* cell and one to keep the number of *Remaining* arguments in the current structure.

There is also a *Stack* for saving *Current* and *Remaining* when substructures are entered.

Initially *Current* points to the root, *Remaining* is zero and the *Stack* is empty.

Forward:

1. If *Current* points to a marked cell then proceed with Backward.
2. Otherwise, mark *Current* as **L**ive. *Current* may point to:
   (a) An immediate value; proceed with Backward.
   (b) A structure; save *Remaining* and *Current* on the *Stack*. Point *Current* to the first cell in the structure and set *Remaining* to the arity of the structure.
   (c) A variable; save *Remaining* and *Current* on the *Stack*. Point *Current* to the variable cell and set *Remaining* to zero.

   Proceed with Forward.

Backward:

1. *Remaining* is zero. We have two alternatives:
   (a) The *Stack* is empty. We are done.
   (b) The *Stack* is not empty. Restore *Current* and *Remaining* from the *Stack* and proceed with Backward.
2. *Remaining* is positive. Decrement *Remaining* and move *Current* to the next cell in the structure. Mark the cell pointed to by *Current* as **I**nternal and proceed with Forward.

Marking must be done atomically using either a lock or some atomic exchange operation. If an atomic exchange operation is used, then care must be taken to merge concurrent updates.

The result of running the algorithm is that all live cells are marked as **L**ive and all internal cells, except the first in each structure, are marked as **I**nternal.

The algorithm can be tail recursively optimised by not saving *Remaining* and *Current* on the *Stack* if *Remaining* is zero.

*Load Balancing of Marking* The main parallelism comes from different workers marking the data they can reach in parallel. Work sharing only occurs when some worker terminates ahead of the others.

Load sharing is achieved by first filling a Work Sharing Stack with *Remaining* and *Current* pairs instead of pushing them on the *Stack*. Idle workers pop entries from the Work Sharing Stacks of active workers. Already marked data is ignored by the algorithm.

We experimented with an approach similar to the one proposed by Ali, i.e. pushing every $i$th pair on the sharing stack. However, this resulted in worse load balancing than our scheme. The experiment is presented in the performance section below.

*Copying* The algorithm traverses the copied data using pointer reversal. We need three variables to manage this: *Next* points to the next cell to be scanned, *Prev* points to the first cell in the backward chain, and *Tag* holds the tag of the pointer which led to the current structure.

Structures are scanned backwards from the last argument to the first.

We need one mark bit in `from-space` to indicate that a cell has been **F**orwarded into `to-space`. We need two mark bits in `to-space` to manage the pointer reversal scheme; one to mark the **H**ead of a structure and one to mark a the **E**ntry point to a structure. Copying the arguments to a structure starts with the last element and works its way up to the first, which is recognised by a **H**ead mark.

The entry point to a structure may not be the same as its first element, i.e. the functor, since the structure may have been reached through a pointer to one of its arguments. To correctly reverse such pointer chains it is necessary to find which cell in a structure the ancestor originally pointed to. This cell is recognised by an **E**ntry mark.

Initially 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`, the first cell in the copied object is marked with **H**ead to indicate that it is the last cell to be scanned in the new object, and the entry point is marked with **E**ntry. *Next* is pointed to the last cell in the copied object, *Prev* points to a **dummy** cell, and *Tag* is empty.

Forward: *Next* may point to

1. An immediate value. Proceed with Backward.
2. A structure or a variable. We have two possibilities:
   (a) The term is already copied which is indicated by a **F**orward marker in `from-space`. Update the cell pointed to by *Next*, keeping possible **E**ntry and **H**ead markers, and proceed with Backward.
   (b) The term is not copied. Lock the head of the surrounding block in `from-space`. We have three possibilities:
      i. A forwarded cell is encountered during the search for the head of the block. This indicates that some other worker is currently copying the block. Wait until the entry points is copied and proceed with Forward 2(a).
      ii. Some other worker has already locked the block. Proceed as above (Forward 2(b)i.).
      iii. We succeed in locking the block. Copy the block into `to-space`, install **F**orward pointers in `from-space`, and set **H**ead and **E**ntry markers in `to-space`. Unlock the block in `from-space`.
      The copied block is now added to the pointer reversed chain, i.e $\mathit{Next}_{\mathrm{tag}}, \mathit{Next}_{\mathrm{ptr}}, \mathit{Prev}, \mathit{Tag} = \mathit{Tag}, \mathit{Prev}, \mathit{Next}, \mathit{Next}_{\mathrm{tag}}$ and *Next* is pointed to the last cell in the copied block (see Figure 9.4).
      $\mathit{Next}_{\mathrm{tag/ptr}}$ denote the tag and the pointer parts of the cell pointed to by *Next*.
      Proceed with Forward.

Backward: *Next* may point to

1. The last cell in the block (marked with **H**ead). Point *Next* to the **E**ntry marked cell in the block and then remove the marks.

   We have two alternatives:
   (a) *Prev* points to the **dummy** cell. We are done.
   (b) *Prev* points to the previous block. Unwind the chain $\mathit{Prev}_{\mathrm{tag}}, \mathit{Prev}_{\mathrm{ptr}}, \mathit{Tag}, \mathit{Next}, \mathit{Prev} = \mathit{Tag}, \mathit{Next}, \mathit{Prev}_{\mathrm{tag}}, \mathit{Prev}, \mathit{Prev}_{\mathrm{ptr}}$ keeping the old markers (see Figure 9.4.3-4).
   Proceed with Backward.
2. An internal cell in the block. Point *Next* to the next cell to be scanned in the block and proceed with Forward.

The head of a block is found by searching upwards in memory for a cell not marked as **I**nternal.

After garbage collection, all choice points are updated such that the saved heap top pointer points to the top of `to-space`. This is necessary since we have merged all old heap segments.

*Load Balancing* Ali proposes that load balancing in the copying phase is achieved by periodically cutting off the backward chain and storing a reference to the cut off piece on a Work Sharing Stack. Idle workers pop backward chains from the stack and proceed to unwind them.

However, we found that the best result is achieved by pushing entries onto the load stack every step until the stack is filled up. Pointer reversal is then used while the stack is full.

The chain is cut during step Forward 2(a)iii. *Next*, *Tag* and *Prev* are stored on the stack. A properly tagged pointer to the copied block is installed in the *Next*-cell, *Prev* is pointed to the dummy cell, and *Next* is pointed to the last cell in the copied block.

*Managing the Load Sharing Stack* We considered using one shared stack on which all workers pushed and popped entries. However, experiments shows that load balancing and access locality improved when all workers have their own stack. For example, when compiling the compiler 75% more work were distributed to idle workers using a number of stacks instead of one single.

The stacks are managed as follows. Each worker push work on its own stack until it is filled up (it may never fill up if other workers are stealing work). An idle worker starts by examining its own stack. Other stacks are only examined when the local stack is empty. This procedure improves locality of both stack accesses and copied data.

Access to the stack is controlled using a spin-lock [132]. The spin-lock becomes a severe bottleneck when a single stack is used (e.g. over 1039366 accesses per second were recorded during garbage collection for the tree benchmark).

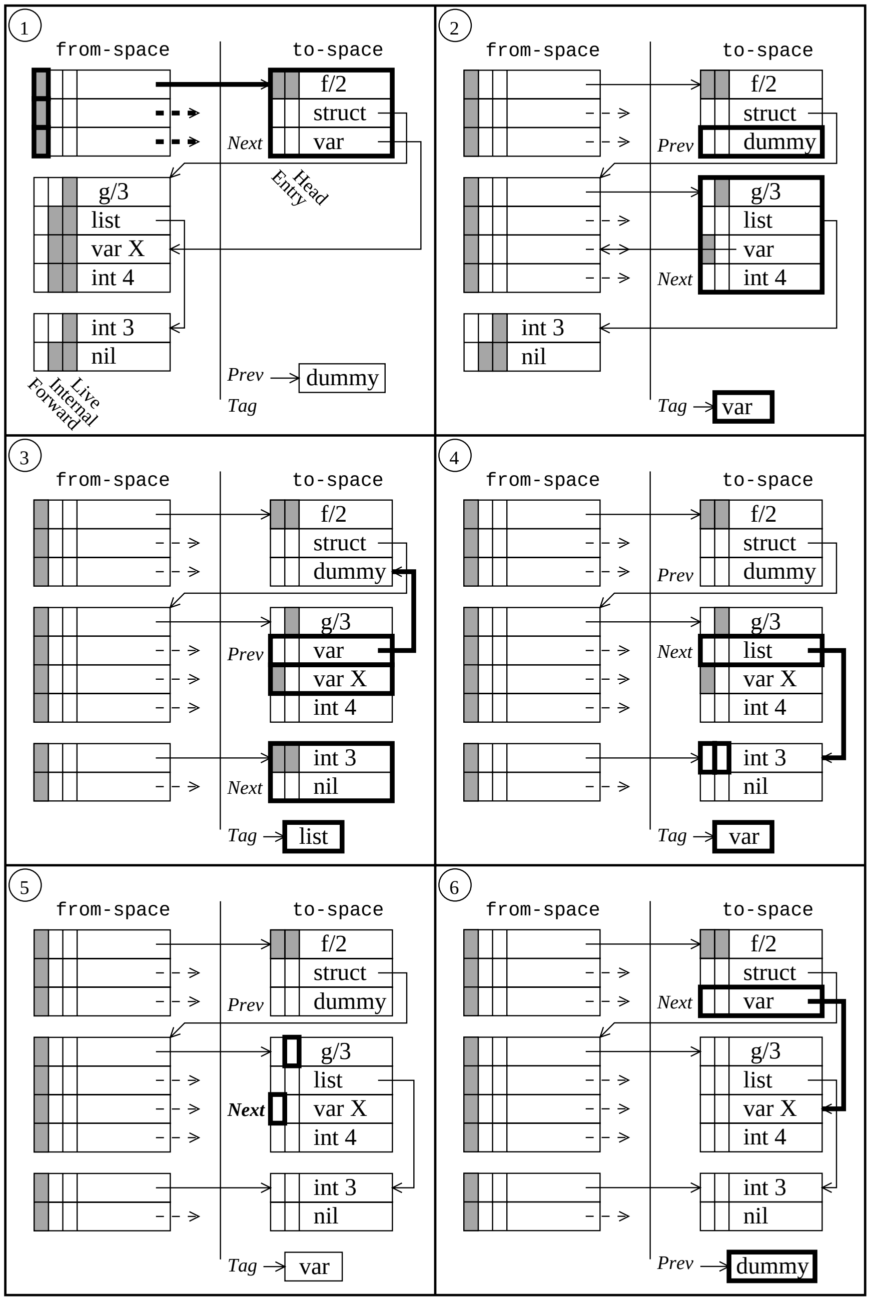


Figure 9.4: The term f(g([3],X,4),X) is copied into `to-space`. Three logical markers are used for each cell in from-space. These can be stored in two mark bits.

*Example of Copying* We illustrate how the algorithm works using an example, see square 1-6 in Figure 9.4. The term f(g([3],X,4),X) is copied into `to-space` after being determined to be reachable, e.g., through an environment register.

1. The top level structure f/2 is copied into `to-space`. Forward pointers are installed in `from-space`, and the head and the entry points are marked. *Next* is pointed to the last element in f/2, *Prev* points to a dummy cell and *Tag* is undefined.
2. Proceeding forward; the *Next*-cell contains a reference to a variable which reside inside a live structure. The entire block is copied. The entry point and the head are marked in `to-space`. The variable cell is linked into the backward chain and we proceed with the last element in the copied block.
3. We find an integer which is an immediate and is left alone. We back up to the variable which points to a forwarded cell. The new address is installed. Continuing backward we encounter a reference to a CONS which is copied into `to-space`. The cell is linked into the backward chain and we proceed forward with the last element in the CONS.
4. The CONS contains immediate values and requires no further processing. We back up, removing head and entry marks in the CONS, unwinding the backward chain, and installing the list pointer to the copied object.
5. Continuing backward we encounter the head of the g/3 block. All marks are removed and *Next* is moved to the entry point.
6. The backward chain is further unlinked: a variable pointer is installed and we continue with the first argument to f/2. What remains is to update the structure pointer and remove the marks (not shown).

*Removing Dead Trail Entries* Dead trail entries can be detected when the marking phase has terminated, e.g., when the trail is scanned for roots during copying. All entries pointing to unmarked cells are dead and must be removed.

*Arranging the Mark Bits* The mark bits in `from-space` can be arranged in the following way.

| | |
|---|---|
| `00` | unmarked (dead) |
| `01` | marked as **L**ive |
| `11` | marked as **L**ive and **I**nternal |
| `10` | marked as **F**orwarded |

The combination **I**nternal but not **L**ive can never appear. We are free to use it as **F**orward indicator.

We only need two bits in `to-space`. The first can indicate **E**ntry and the second **H**ead. The same bit positions can be used in both `to-space` and `from-space`.

Note that it is not necessary to clear the mark bits in `from-space`. This memory is overwritten by new data which have zeroed mark bits. The mark bits in `to-space` are restored by the pointer reversal scheme.

## 9.5 GENERATIONAL PARALLEL GARBAGE COLLECTION

Generational garbage collection [7, 121] relies on the observation that newly created objects are short-lived. Thus, garbage collection should concentrate on recently created data. The heap is split into two or more generations. 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 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 and Lindgren show how it can be arrange that only *trailed* bindings may be cross-generational by setting the limit where trailing occurs appropriately.

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.

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

### Additions to the Algorithms

The key concern is to limit marking and copying to the garbage collected generation.

*Marking* Alternative 1 in Forward is modified to read:

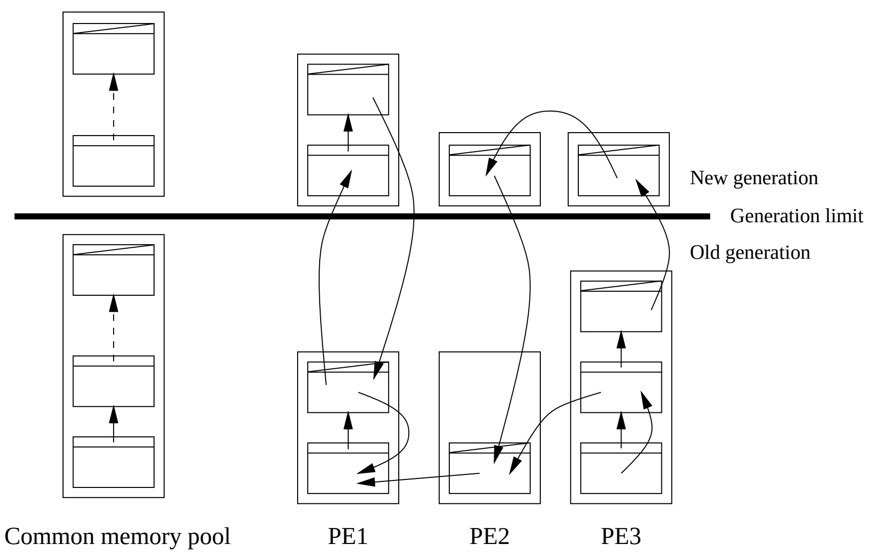


Figure 9.5: Memory organisation in a generational collector.

1. If *Current* points to a marked cell *or a [unmarked] cell in the old generation,* then proceed with Backward.

*Copying* Alternative 1 and 2 in Forward are modified to read:

1. An immediate value *or an object in the old generation.* Proceed with Backward.

2. A structure or a variable *in the new generation.* We have two...

**Finding Live Data**

The roots to the data in new generation are found in registers, choice points, trail entries added after the last gc, environments, and in the old generation (we call pointers from the old generation to the new *cross generational*). References to the different generations may appear in all these places. To limit the garbage collection to one generation, it is essential that the collector can differentiate between objects in different 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 9.5).

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

*Limiting the Search for Roots* 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 standard 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. Bevemyr and Lindgren [35] 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 unwinded to a point below the saved point, adding a slight overhead in the code for backtracking.

If time stamped variables are used then it is possible to remove trail entries which are only present to record cross generational bindings. They should be removed when the trails are updated after gc.

**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 [9] does).

After garbage collection, all choice points are updated such that the saved heap top pointer points to the start of the first segment in the new generation. This is necessary since we cannot deallocate memory in the old generation when backtracking.

A subtle point is that after garbage collection all saved heap top pointers need to be update in all choice points, even old surviving choice point which have not otherwise been visited. The reason is that the heap top must be restored to the beginning of the new generation on backtracking, i.e. the top of the old segments which have been collapsed into one large segment. The beginning of the new generation is not constant since workers may acquire different initial segments after each collection.

Visiting all choice points can be avoided by letting each worker keep its initial new generation segment (improving locality in the process). The heap top pointer of a choice point is then modified once when it survives its first garbage collection.

**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., surviving environments created before the last collection which have not been protected by a surviving choice point.
3. *Old protected environments*, i.e., surviving environments protected by a surviving choice point.

Environments in the first and second category have to be searched. References from the third category are found when examining the trail.

## 9.6 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.

   | | |
   |---|---|
   | ATM | ordering number |
   | HVA | unbound variable |

   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 Bevemyr-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.

```
insert(Key,t(Left,Element,Right)) :-
        Key @< Element,
        ...
insert(Key,t(Left,Element,Right)) :-
        Element @< Key,
        ...
```

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.

## 9.7 PERFORMANCE

We have implemented the algorithm in Reform Prolog. Each worker runs as a separate unix process, using mmap to create a shared memory. Spin-locks are used to implement barrier synchronisation. System semaphores can be used with a slight overhead.

All measurements were made on a bus-based shared memory multiprocessor: a SparcCenter 2000 configured with 20 processors. We used 12 processors in our experiments since we are only guests at the machine. All run times are in seconds wall time.

We used a fixed size for the new generation (3.5 MB), regardless of the number of processors used. It would be reasonable to scale the size of the new generation with a fixed amount for each worker. This reduces the number of garbage collections, which in its turn reduces the total collection time. However, this would make it difficult to distinguish the speed up due to parallelism from the speed up due to fewer garbage collections.

There are no published performance figures for other proposed copying schemes for sequential Prolog, which makes it hard to compare the collectors sequential performance. Compared with SICStus semi-generational mark-compact collector ours seem to be 5–10 times faster for a range of measured programs.

SICStus collector is generational in the sense that it only garbage collects data reachable from choice points created, or visited, since the last collection. A mark bit in the choice point is used to distinguish unvisited choice points from visited.

### Benchmarks

**Compiler** consists of the Reform Compiler compiling itself. This program has two phases. First, the code is read and pre-processed creating a large segment of live data. Then follows a phase where large amounts of heap is consumed but only a small fraction survives collection. 97,510,416 bytes are reclaimed in 26 garbage collections.

**Tree** consists of the search part of an othello program. The program creates several large live tree structures. A significant portion of the heap survive each collection. 5,864,080 are reclaimed in 4 collections.

**Ray Tracer** is a simple ray tracing program which handles shading, reflection and transparent objects. This program is very heap intensive. A large number of temporary floating point numbers are generated, but only a small amount of data survive. 42,201,496 bytes are reclaimed in 13 collections.

**TSP** implements an approximation algorithm for the Travelling Salesman Problem. Some local backtracking is performed. Only small amounts of data survive each garbage collection. 31,372,168 bytes are reclaimed in 6 collections.

**Load balancing**

We have measured the load balancing capabilities of the collector both in isolation, running only the collector in parallel, and in a parallel Prolog system.

During the isolated tests the worker running the sequential program is the only one which creates data. The others have to obtain work from the sequential worker through load balancing.

The following table shows the number of words *marked* by each worker when the programs were executed sequentially. (The number of marked words are larger then the number of copied since a small number of words are marked more than once.)

| Worker | Compiler | Tree | Ray Tracer | TSP |
|---|---|---|---|---|
| 1 | 799726 | 109267 | 19175 | 1641 |
| 2 | 134362 | 85101 | 5661 | 1722 |
| 3 | 88245 | 99378 | 4331 | 82 |
| 4 | 116348 | 102776 | 4434 | 164 |
| 5 | 99084 | 110109 | 3789 | 82 |
| 6 | 105615 | 122578 | 9799 | 82 |
| 7 | 140534 | 118109 | 5023 | 1804 |
| 8 | 111875 | 123442 | 8314 | 1887 |
| 9 | 117461 | 123117 | 4163 | 82 |
| 10 | 128068 | 119448 | 4382 | 246 |
| 11 | 205040 | 129844 | 4344 | 164 |
| 12 | 128359 | 121526 | 4324 | 0 |

The following table shows the number of words *copied* by each worker when the programs were executed sequentially.

| Worker | Compiler | Tree | Ray Tracer | TSP |
|---|---|---|---|---|
| 1 | 767041 | 121578 | 41350 | 0 |
| 2 | 207150 | 104200 | 2914 | 0 |
| 3 | 79167 | 59095 | 3612 | 0 |
| 4 | 88767 | 113815 | 2289 | 0 |
| 5 | 130295 | 119455 | 2784 | 0 |
| 6 | 88178 | 112095 | 1926 | 0 |
| 7 | 95636 | 126025 | 8228 | 0 |
| 8 | 79831 | 123380 | 2496 | 0 |
| 9 | 104115 | 71410 | 2068 | 0 |
| 10 | 102794 | 136895 | 1244 | 0 |
| 11 | 122509 | 131860 | 2764 | 0 |
| 12 | 122121 | 144820 | 4464 | 0 |

The load balancing schemes behaves well for the programs with large live data sets, Compiler and Tree, and less well for the programs with a small amount of live data; Ray Tracer and TSP. There are two factors at play in the case of Ray Tracer and TSP. Both have small sets of live data with small amounts of work to distribute to other workers. They both make extensive use of lists which the load balancing algorithms have difficulties with.

Total garbage collection times.

| Program | 1 | 2 | 4 | 8 | 12 |
|---|---|---|---|---|---|
| Compiler | 8.4 | 7.7 | 7.2 | 6.6 | 6.9 |
| Tree | 5.2 | 3.06 | 1.92 | 1.17 | 1.01 |
| Ray tracer | 0.28 | 0.29 | 0.29 | 0.30 | 0.31 |
| TSP | 0.089 | 0.086 | 0.079 | 0.093 | 0.135 |

The garbage collection times are improved up to 5 times for the program with large live data-structures (Tree).

We also measured the load balancing scheme during parallel execution. The Compiler program has not been parallelised yet and had to be left out. The table shows load distribution of marking during parallel execution, measured in words marked by each worker.

| Worker | Tree | Ray Tracer | TSP |
|---|---|---|---|
| 1 | 95078 | 6064 | 2583 |
| 2 | 167414 | 6946 | 1360 |
| 3 | 107451 | 5172 | 1473 |
| 4 | 154927 | 7323 | 2992 |
| 5 | 111866 | 6669 | 1107 |
| 6 | 113623 | 8029 | 1123 |
| 7 | 128216 | 6981 | 1180 |
| 8 | 172380 | 8586 | 1529 |
| 9 | 113401 | 7263 | 1138 |
| 10 | 108590 | 7692 | 1690 |
| 11 | 171517 | 7924 | 1332 |
| 12 | 175173 | 7688 | 1668 |

Load distribution of copying during parallel execution measured in words copied by each worker.

| Worker | Tree | Ray Tracer | TSP |
|---|---|---|---|
| 1 | 81128 | 5751 | 7062 |
| 2 | 113553 | 6708 | 1183 |
| 3 | 120905 | 6847 | 1025 |
| 4 | 148698 | 7158 | 1068 |
| 5 | 136935 | 7148 | 1069 |
| 6 | 143421 | 7768 | 1008 |
| 7 | 155275 | 6795 | 1072 |
| 8 | 155636 | 7805 | 1000 |
| 9 | 131793 | 7336 | 1021 |
| 10 | 147219 | 7160 | 1006 |
| 11 | 153602 | 7830 | 1130 |
| 12 | 148462 | 7734 | 1184 |

*A Comparison with Ali's Load Balancing Scheme* We have also implemented Ali's load balancing scheme where works is spawned every $i$th pointer reversal (or push as would be the analogy in the marking phase). We measured 6 different values of $i$ when running the Tree benchmark.

Marking (normalised with respect to the smallest number in each column):

| Worker | 1 | 2 | 4 | 8 | 16 | 32 |
|---|---|---|---|---|---|---|
| 1 | 1.0 | 1.5 | **2.6** | 2.3 | 1.2 | 2.7 |
| 2 | **1.7** | 1.0 | 1.5 | 1.1 | 1.7 | 1.0 |
| 3 | 1.1 | 1.0 | 1.0 | 1.1 | 1.9 | 1.1 |
| 4 | 1.5 | 1.2 | 1.6 | 1.0 | 1.2 | 1.0 |
| 5 | 1.1 | 1.2 | 1.6 | 1.1 | 1.2 | 1.2 |
| 6 | 1.1 | 1.2 | 1.9 | 1.4 | 1.9 | **3.3** |
| 7 | 1.3 | 1.3 | 2.0 | 1.3 | 2.0 | 1.3 |
| 8 | 1.7 | **1.5** | 1.7 | 1.3 | 1.3 | 1.3 |
| 9 | 1.1 | 1.2 | 1.7 | 1.2 | 1.0 | 1.3 |
| 10 | 1.1 | 1.2 | 2.0 | 1.3 | 1.3 | 1.2 |
| 11 | 1.7 | 1.3 | 2.3 | **1.6** | 1.4 | 1.3 |
| 12 | 1.7 | 1.3 | 1.9 | 1.5 | **2.2** | 1.3 |

Copying (normalised with respect to the smallest number in each column):

| Worker | 1 | 2 | 4 | 8 | 16 | 32 |
|---|---|---|---|---|---|---|
| 1 | 1.0 | **2.3** | 3.6 | 1.4 | 1.0 | 1.1 |
| 2 | 1.4 | 1.7 | 2.0 | 4.4 | 1.2 | 1.4 |
| 3 | 1.5 | 1.1 | 3.0 | 1.0 | 7.8 | 1.4 |
| 4 | 1.8 | 2.0 | 4.1 | 1.1 | 1.6 | 1.0 |
| 5 | 1.7 | 1.0 | 2.6 | 5.7 | 1.8 | 1.8 |
| 6 | 1.8 | 1.0 | **4.6** | **14** | 1.8 | 1.8 |
| 7 | **1.9** | 2.1 | 2.5 | 1.1 | 1.8 | 1.9 |
| 8 | 1.9 | 1.9 | 2.3 | 1.4 | 1.3 | 1.9 |
| 9 | 1.6 | **2.3** | 2.2 | 4.9 | 1.8 | 1.4 |
| 10 | 1.8 | 2.3 | 1.0 | 1.4 | 1.6 | 1.6 |
| 11 | 1.9 | 2.3 | 2.3 | 1.3 | 1.6 | **6.8** |
| 12 | 1.8 | 2.2 | 4.0 | 4.6 | 2.0 | 1.5 |

It appears that frequent spawning is of work is profitable, at least when garbage collecting Prolog.

*Problems* The load balancing schemes perform badly when large shallow lists are used since there is not much work to distribute. In the marking phase no work is spawn since no entries are pushed on the Work Sharing Stack. This is due to the tail recursive optimisation.

The copying phase behaves worse. Small list sections are spawn but all that remains is to unwind them, resulting in very fine grained parallelism. A solution to this might be to *not* spawn when lists are encountered, or at least spawn less often.

Programs using data structures such as binary trees behave much better. The collector was able to almost perfectly balanced the collection of the Tree program.

### Combined Parallel Performance

GC execution time

| Program | 1 | 2 | 4 | 8 | 12 |
|---|---|---|---|---|---|
| Tree | 5.2 | 2.60 | 1.36 | 1.01 | 0.98 |
| Ray tracer | 0.291 | 0.218 | 0.212 | 0.197 | 0.210 |
| TSP | 0.089 | 0.061 | 0.056 | 0.080 | 0.077 |

Total parallel execution time

| Program | 1 | 2 | 4 | 8 | 12 |
|---|---|---|---|---|---|
| Tree | 14.10 | 7.77 | 4.36 | 2.48 | 2.10 |
| Ray tracer | 28.47 | 14.62 | 7.62 | 3.97 | 2.84 |
| TSP | 137 | 72.2 | 34.3 | 19.3 | 13.1 |

The best results are obtained when the programs are well balanced, i.e., when all workers produce an equal amount of live data. The load balancing schemes will even out occasional peaks but they will not, by themselves, perfectly balance the garbage collector.

It should be noted that Reform Prolog parallelise programs in such a way that uniform amounts of data is likely to be created on each worker. However, it also use long lists when creating parallel processes, resulting in poor load balancing for those parts of the data.

Crammond [59] measures his collector without taking the initial synchronisation overheads into account. He argues that they can be made arbitrarily small. This is not completely true. There is always an runtime overhead for either keeping the machine in a controlled state or for frequently checking if garbage collection is needed. Our measurements include synchronisation overheads.

Crammond parallelises a mark-compact algorithm by pushing all external references on stacks placed above each original heap. These stacks might have to be almost as large as the garbage collected heaps. Crammond report almost linear speed up for his collector. This is not surprising since all workers collect equally sized heaps using an algorithm proportional, in execution time, to the size of each heap, i.e. all workers have an equal amount of work. The good speed up is a consequence of the slow algorithm.

## 9.8 CONCLUSION

We have shown how a copying garbage collector for Prolog can be designed using Ali's scheme for parallel copying collection. We have also shown how the resulting collector can be made generational. An improved strategy for load balancing for Prolog has been presented.

Our main contributions are:

- A reasonable way of dealing with direct references to internal cells; load balanced parallel marking of live data.
- Generational garbage collection.
- Prolog specific considerations, i.e. handling internal pointers, a modified load balancing scheme, and a scheme for ordering variables.
- An evaluation of how well the algorithm behaves in practice.

**Acknowledgements.** I would like to thank Khayri Ali for helping me through the trickier parts of his garbage collection scheme. Thanks also to

Thomas Lindgren, Per Mildner and Katrin Boberg Bevemyr for valuable discussions on the content and its presentation.

# Bibliography

The numbers inside braces indicate on which pages each citation occured.

1. H. Aït-Kaci, *The WAM: A (Real)Tutorial*, MIT Press, 1991. {4}

2. K. Ali, Incremental Garbage Collection for OR-Parallel Prolog Based on WAM, *Gigalips Workshop*, 1989. {22}

3. K. Ali, A Parallel Copying Garbage Collection Scheme for Shared-Memory Multiprocessors, *New Generation Computing*, 14(1):53–77, 1996. {12, 22, 142, 145}

4. K. Ali, R. Karlsson, The Muse OR-Parallel Prolog Model and its Performance, *Proc. North American Conf. Logic Programming*, MIT Press, 1990. {22}

5. S. Anderson, P. Hudak, Compilation of Haskell Array Comprehensions for Scientific Computing, *Proc. SIGPLAN'90 Conf. on Programming Language Design and Implementation*, ACM Press, 1990. {20, 124}

6. A. W. Appel, A Runtime System, *Lisp and Symbolic Computation*, 3(4):343–380, 1990. {21, 130}

7. A. W. Appel, Simple Generational Garbage Collection and Fast Allocation, *Software—Practice and Experience*, 19(2):171–183, 1989. {20, 21, 130, 134, 156}

8. A. W. Appel, Garbage Collection, *Advanced Language Implementations*, MIT Press, 1991. {21}

9. K. Appleby, M. Carlsson, S. Haridi, D. Sahlin, Garbage Collection for Prolog Based on WAM, *Communications of the ACM*, 31(6):719–741, June 1988. {20, 21, 128, 130, 135, 142, 158}

10. K. R. Apt, Arrays, Bounded Quantifications and Iteration in Logic and Constraint Logic Programming, *Science of Computer Programming*, 26(1-3):133–148, 1996. {19, 124}

11. H. Arro, J. Barklund, J. Bevemyr, Parallel Bounded Quantifications — Preliminary Results, *ACM SIGPLAN Notices*, 28(5):117–124, 1993. {20, 84, 85, 95, 100}

12. Arvind, K. P. Gostelow, The U-Interpreter, *IEEE Computer*, 15(2):42–49, 1982. {18}

13. L. Augustsson, T. Johnsson, Parallal Graph Reduction with the $\langle v, \mathrm{G}\rangle$-Machine, *Proc. Workshop on Implementation of Lazy Functional Languages*, 1988. {23}

14. L. Augustsson, T. Johnsson, The Chalmers Lazy-ML Compiler, *The Computer Journal*, 32(2):127–141,1989. {23}

15. H. G. Baker, List Processing in Real Time on a Serial Computer, *Communications of the ACM*, 21(4):280–294, 1978. {22}

16. J. Barklund, *Parallel Unification*, PhD thesis, Comp. Sci. Dept., Uppsala Univ., Uppsala, 1990. {18}

17. J. Barklund, Bounded Quantifications for Iteration and Concurrency in Logic Programming, *New Generation Computing*, 12(2):161–182, Springer-Verlag, 1994. {9, 100}

18. J. Barklund, *Tabulation of Functions in Definite Clause Programs*, UPMAIL Tech. Rep. 82, Comp. Sci. Dept., Uppsala Univ.,1994. {20, 125}

19. J. Barklund, J. Bevemyr, Executing Bounded Quantifications on Shared Memory Multiprocessors, *Proc. Intl. Conf. Programming Language Implementation and Logic Programming*, LNCS 714, Springer-Verlag, 1993. {100}

20. J. Barklund, J. Bevemyr, Prolog With Arrays and Bounded Quantifications, *Logic Programming and Automated Reasoning*, LNCS 698, Springer-Verlag, 1993 {84, 86, 87, 100}

21. J. Barklund, N. Hagner, M. Wafin, KL1 in Condition Graphs on a Connection Machine, *Proc. Intl. Conf. Fifth Generation Computer Systems*, Ohmsha, 1988. {18}

22. J. Barklund, N. Hagner, M. Wafin, *Connection Graphs*, UPMAIL Tech. Rep. 48, Comp. Sci. Dept., Uppsala Univ., Uppsala, 1988. {18}

23. J. Barklund, P. M. Hill, *Extending Gödel for Expressing Restricted Quantifications and Arrays*, UPMAIL Tech. Rep. 102, Comp. Sci. Dept., Uppsala Univ., Uppsala, 1995. {19, 101, 124}

24. J. Barklund, H. Millroth, *Nova Prolog*, UPMAIL Tech. Rep. 52, Comp. Sci. Dept., Uppsala Univ., 1988. {18}

25. J. Barklund, H. Millroth, Integrating Complex Data Structures in Prolog, *Proc. 1987 Symp. Logic Programming*, IEEE, 1987. {105}

26. J. Barklund, H. Millroth, Providing Iteration and Concurrency in Logic Programs Through Bounded Quantifications, *Intl. Conf. on Fifth Generation Computer Systems*, 1992. {9, 84, 85, 100}

27. J. Barklund, H. Millroth, Garbage Cut for Garbage Collection of Iterative Prolog Programs, *3rd Symp. on Logic Programming*, IEEE, 1986. {20, 129}

28. Y. Bekkers, O. Ridoux and L. Ungaro, Dynamic Memory Management for Sequential Logic Programming Languages, *Proc. Intl. Workshop on Memory Management 92*, LNCS 637, Springer-Verlag, 1992. {13, 20, 21, 129, 143}

29. G. Bell, Ultracomputers: A Teraflop Before Its Time, *Communications of the ACM*, 35(8)26–47, 1992. {45}

30. R. E. Bellman, *Dynamic Programming*, Princeton Univ. Press, Princeton, N.J., 1957. {20, 125}

31. J. Bevemyr, *The Luther WAM Emulator*, UPMAIL Tech. Rep. 72, Comp. Sci. Dept., Uppsala Univ., 1992. {87, 107}

32. J. Bevemyr, *A Recursion Parallel Prolog Engine*, PhL thesis, Uppsala Theses in Computer Science 16, Uppsala Univ., 1993. {52, 62, 77, 84, 88, 90, 107, 147}

33. J. Bevemyr, T. Lindgren, H. Millroth, Exploiting Recursion-Parallelism in Prolog, *PARLE-93*, LNCS 694, Springer Verlag, 1993. {34, 62, 63, 124, 146}

34. J. Bevemyr, T. Lindgren, H. Millroth, Reform Prolog: The Language and its Implementation, *Logic Programming: Proc. Tenth Intl. Conf.*, MIT Press, 1993. {8, 52, 62, 63, 75, 84, 88, 146}

35. J. Bevemyr and T. Lindgren, A Simple and Efficient Copying Garbage Collector for Prolog, *Proc. Programming Language Implementation and Logic Programming*, LNCS 844, Springer-Verlag, 1994. {142, 143, 144, 158}

36. R. S. Bird, Tabulation Techniques for Recursive Programs, *ACM Computing Surveys*, 12(4):403–417, 1980. {20, 125}

37. G. Blelloch, *Vector Models and Data-Parallel Computing*, MIT Press, 1990. {75}

38. G. Blelloch, Programming Parallel Algorithms, *Communications of the ACM*, 39(3), 1996. {18}

39. G. Blelloch, P. Gibbons, Y. Matias, Provably Efficient Scheduling for Languages with Fine-Grained Parallelism, *7th ACM Symp. on Parallel Algorithms and Architectures*, 1995. {82}

40. P. Borgwardt, Parallel Prolog Using Stack Segments on Shared-Memory Multiprocessors, *IEEE Symp. Logic Programming*, 1984. {14, 81}

41. P. Borgwardt, D. Rea, Distributed Semi-Intelligent Backtracking for a Stack Based AND-Parallel Prolog, *Proc. Symp. Logic Programming*, IEEE Computer Society, 1986. {14}

42. H. Burkhardt, S. Frank, B. Knobe, J. Rothnie, *Overview of the KSR1 Computer System*, Tech. Rep. KSR-TR-9202001, Kendall Square Research, 1992. {7}

43. M. Burke, R. Cytron, Interprocedural Dependence Analysis and Parallelization, *Proc. SIGPLAN'86 Symp. Compiler Construction*, 1986. {18}

44. M. Carlsson, *Design and Implementation of an Or-Parallel Prolog Engine*, PhD thesis, SICS-RITA/02, 1990. {22}

45. M. Carlsson, *SICStus Prolog Internals Manual*, Internal Report, Swedish Institute of Computer Science, 1989. {20}

46. M. Carlsson, *Variable Shunting for the WAM*, Tech. Rep. R91-07, Swedish Institute of Computer Science, 1989. {21, 149}

47. J.-H. Chang, *High Performance Execution of Prolog Programs Based on a Static Dependency Analysis*, PhD thesis, UCB/CSD 86/263, Univ. Calif. Berkeley, 1986. {14, 64}

48. J.-H. Chang, A. M. Despain, D. DeGroot, AND-Parallelism of Logic Programs Based on Static Data Dependency Analysis, *Proc. IEEE Symp. Logic Programming*, 1985. {14}

49. C. J. Cheney, A Nonrecursive List Compacting Algorithm, *Communications of the ACM*, 13(11):677–678, November 1970. {21, 22, 128, 131, 144}

50. T. Chikayama, Y. Kimura, Multiple Reference Management in Flat GHC, *Logic Programming—Proc. Fourth Intl. Conf.*, MIT Press, 1987. {16, 104}

51. K. L. Clark, F. McCabe, The Control Facilities of IC-Prolog, *Expert Systems in the Micro-Electronic World* (ed. D. Michie), Edinburgh University Press, 1979. {14, 16}

52. K. L. Clark, S. Gregory, A Relational Language for Parallel Programming, *Proc. ACM Symp. Functional Programming and Computer Architecture*, 1981. {16}

53. K. L. Clark, S. Gregory, *PARLOG: A Parallel Logic Programming Language*, Rep. DOC 83/5, Dept. Computing, Imperial College, London, 1983. {5, 16}

54. K. L. Clark, S. Gregory, Notes on the Implementation of PARLOG, *Journal of Logic Programming*, 2(1):17–42, 1985. {5}

55. J. Cohen, Garbage Collection of Linked Data Structures, *Computing Surveys*, 13(3):341–367, September 1981. {20, 21, 128}

56. A. Colmerauer, H. Kanoui, R. Pasero, P. Roussel, Un Système de Communication Homme-Machine en Français, Groupe de Recherche en Intelligence Artificielle, Univ. de Aix-Marseille, Luminy, 1972. {3, 99}

57. J. S. Conery, D. F. Kibler, Parallel Interpretation of Logic Programs, *Proc. ACM Symp. Functional Programming and Computer Architecture*, 1981. {14}

58. P. Cousot, R. Cousot, Abstract Interpretation: a Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints, *Conf. Record of the Fourth ACM Symp. on Principles of Programming Languages*, 1977. {64}

59. J. Crammond, A Garbage Collection Algorithm for Shared Memory Parallel Processors, *Intl. Journal of Parallel Programming*, 17(6):497–522, 1988. {22, 166}

60. J. Crammond, The Abstract Machine and Implementation of Parallel Parlog, *New Generation Computing*, 10(4):385–422, Springer-Verlag, 1992. {16}

61. S. K. Debray, Static Inference of Modes and Data Dependencies in Logic Programs, *ACM Transactions on Programming Languages and Systems*, 11(3):418–450, 1989. {64}

62. S. K. Debray, A Simple Code Improvement Scheme for Prolog, *Journal of Logic Programming*, 13(1):57–88, 1992. {34, 51}

63. S. K. Debray, Efficient Dataflow Analysis of Logic Programs, *Journal of the ACM*, 39(4):949–984, 1992. {64}

64. S. K. Debray, M. Jain, A Simple Program Transformation for Parallelism, *Intl. Symp. Logic Programming*, MIT Press, 1994. {17}

65. S. K. Debray, D. S. Warren, Automatic Mode Inference for Logic Programs, *Journal of Logic Programming*, 5(3):207–229, 1988. {64}

66. D. DeGroot, Restricted AND-parallelism, *Proc. Intl. Conf. Fifth Generation Computer Systems*, North-Holland, Amsterdam, 1984. {14}

67. B. Demoen, G. Engels, P. Tarau, Segment Preserving Copying Garbage Collection for WAM based Prolog, *Proc. 1996 ACM Symp. on Applied Computing*, ACM Press, 1996 {13, 21, 143, 144}

68. J. B. Dennis, Data Flow Supercomputers, *IEEE Computer*, 13(11):48–56, 1980. {18}

69. L. P. Deutsch, D. G. Bobrow, An Efficient Incremental Automatic Garbage Collector, *Communications of the ACM*, 19(9):522–526, 1976. {11, 20}

70. A. Dovier, E. G. Omodeo, E. Pontelli, G. Rossi, {log}: a Logic Programming Language with Finite Sets, *Proc. Intl. Conf. Logic Programming*, MIT Press, 1991. {19, 124}

71. M. Doprochevsky, *Garbage Collection in the OR-Parallel Logic Programming System ElipSys*, ECRC Tech. Rep. DPS-85, 1991. {22}

72. M. Dorochevsky, K. Schuerman, A. Véron, and J. Xu, Constraint Handling, Garbage Collection and Execution Model Issues in ElipSys, *Parallel Execution of Logic Programs, Proc. ICLP'91 Pre-Conf. Workshop*, LNCS 569, 1991. {22}

73. J.R. Ellis, K. Li, A.W. Appel, *Real-time Concurrent Collection on Stock Multiprocessors*, Tech. Rep. 25, Digital Systems Research Center, Palo Alto, 1988. {23}

74. L.-H. Eriksson, M. Rayner, Incorporating Mutable Arrays into Logic Programming, *Proc. Intl. Logic Programming Conf.*, Uppsala University, Uppsala, 1984. {104}

75. B. Fagin, *Data-Parallel Logic Programming Systems*, Tech. Rep., Thayer School of Engineering, Dartmouth Collage, New Hampshire, 1990. {18}

76. R. Fenichel, J. Yochelson, A LISP Garbage-collector for Virtual- memory Computer Systems, *Communications of the ACM*, 12(11):611–612, 1969.

77. I. Foster, S. Taylor, Strand: A Practical Parallel Programming Language, *North American Conf. Logic Programming*, MIT Press, 1989. {5, 16}

78. I. Foster, W. Winsborough, Copy Avoidance through Compile-Time Analysis and Local Reuse, *Proc. Intl. Symp. Logic Programming*, MIT Press, 1991. {145}

79. M. A. Friedman, *A Characterization of Prolog Execution*, PhD thesis, Univ. of Wisconsin at Madison, 1992. {71}

80. D. P. Friedman, D. S. Wise, M. Wand, Recursive Programming through Table Look-Up, *Symp. on Symbolic and Algebraic Computation*, ACM, 1976. {20, 125}

81. T. Getzinger, *Abstract Interpretation for the Compile-Time Analysis of Logic Programs*, PhD thesis, Tech. Rep. ACAL-TR-93-09, Univ. of South California, 1993. {68}

82. D. Gries, *The Science of Programming*, Springer-Verlag, 1981. {20, 125}

83. G. Gupta, V. Santos Costa, R. Yang, M. V. Hermenegildo, IDIOM: Intergrating Dependent and-, Independent And- and Or-parallelism, *Proc. Intl. Logic Programming Symp.*, MIT Press, 1991. {15}

84. G. Gupta, M. V. Hermenegildo, ACE: And/Or-Parallel Copying-Based Execution of Logic Programs, *Parallel Execution of Logic Programs*, LNCS 569, Springer-Verlag, 1991. {14}

85. G. Gupta, M. V. Hermenegildo, E. Pontelli, V. Santos Costa, ACE: And/Or-parallel Copying-based Execution of Logic Programs, *Proc. Intl. Conf. Logic Programming*, MIT press, 1994. {14, 81}

86. G. Gupta, B. Jayaraman, Combined And-Or Parallelism on Shared Memory Multiprocessors, *North American Conf. Logic Programming*, MIT Press, 1989. {15}

87. G. Gupta, E. Pontelli, Last Alternative Optimization, *8th IEEE Symp. on Parallel and Distributed Processing*, IEEE Computer Society, 1996. {17, 81}

88. R. H. Halstead, Implementation of Multilisp: Lisp on a Multiprocessor, *ACM Symp. LISP and Functional Programming*, 1984. {22}

89. S. Haridi, A Logic Programming Language Based on the Andorra Model, *New Generation Computing*, 7(2-3):109–125, 1990. {15}

90. S. Haridi, S. Janson, Kernel Andorra Prolog and its Computation Model, *Intl. Conf. Logic Programming*, MIT Press, 1990. {15}

91. R. Harper, D. MacQueen, R. Milner, *Standard ML*, Technical Report ECS-LFCS-86-2, Dept. Computer Science, Edinburgh Univ., 1986. {20, 124}

92. W. L. Harrison III, The Interprocedural Analysis and Parallelization of Scheme Programs, *Lisp and Symbolic Computation*, 2(3-4):176–396, 1989. {17}

93. M. P. Herlihy, J. E. B. Moss, Lock-Free Garbage Collection for Multiprocessors, *IEEE Transactions on Parallel and Distributed Systems*, 3(3):304–311, 1992. {22}

94. M.V. Hermenegildo, An Abstract Machine for Restricted AND-Parallel Execution of Logic Programs, *Third Intl. Conf. Logic Programming*, LNCS 225, Springer-Verlag, 1986. {14, 81}

95. M. V. Hermenegildo, *An Abstract Machine Based Execution Model for Computer Architecture Design and Efficient Implementation of Logic Programs in Parallel*, PhD thesis, University of Texas At Austin, 1986. {14, 15, 81}

96. M. V. Hermenegildo, Relating Goal Scheduling, Precedence and Memory Management in AND-Parallel Execution of Logic Programs, *Fourth Intl. Conf. Logic Programming*, MIT Press, 1987. {81}

97. M. V. Hermenegildo, D. Cabeza, M. Carro, Using Attributed Variables in the Implementation of Concurrent and Parallel Logic Programming Systems, *Proc. Intl. Conf. Logic Programming*, MIT Press, 1995. {16}

98. M. V. Hermenegildo, M. Carro, Relating Data-Parallelism and (And–) Parallelism in Logic Programs, *Journal of Computer Languages*, 22(2-3), 1996. {17, 20, 81, 125}

99. M. V. Hermenegildo, K. J. Greene, &-Prolog and its Performance: Exploiting Independent And-Parallelism, *Proc. Intl. Conf. Logic Programming*, MIT Press, 1990. {14, 81}

100. M. V. Hermenegildo, F. Rossi, Non-Strict Independent And-Parallelism, *Proc. Intl. Conf. Logic Programming*, MIT Press, 1990. {14, 15, 66, 81}

101. P. M. Hill, J. W. Lloyd, *The Gödel Programming Language*, MIT Press, 1993. {19, 105, 124}

102. W. D. Hillis, G. L. Steele Jr., Data Parallel Algorithms, *Communications of the ACM*, 29(12):1170–1183, 1986. {18}

103. L. F. Huelsbergen, J. R. Larus, A Concurrent Copying Garbage Collector for Languages that Distinguish (Im)mutable Data, *Principles and Practice of Parallel Programming*, ACM, 1993. {22}

104. P. Hudak, S. Peyton Jones, and P. Wadler (editors), Report on the Programming Language Haskell, A Non-strict Purely Functional Language (Version 1.2), *ACM SIGPLAN Notices*, 27(5), 1992. {123}

105. A. Imai, E. Tick, Evaluation of Parallel Copying Garbage Collection on Shared-Memory Multiprocessors, *IEEE Transactions on Parallel and Distributed Computing*, 4(9):1030–1040, 1993. {22}

106. K. E. Iverson, *A Programming Language*, Wiley, 1962. {18}

107. S. Janson, *AKL A Multiparadigm Programming Language*, PhD thesis, Comp. Sci. Dept., Uppsala Univ., Uppsala, 1994. {15, 81}

108. S. Janson, J. Montelius, *The Design of the AKL/PS 0.0 Prototype Implementation of the Andorra Kernel Language*, ESPRIT Deliverable, EP 2471 (PEPMA), Swedish Institute of Computer Science, 1992. {15, 81}

109. N. Jones & H. Søndergaard, *A Semantics-Based Framework for the Abstract Interpretation of Prolog*, report 86/14, University of Copenhagen, 1986. {64}

110. R. Jones, R. Lins, *Garbage Collection: Algorithms for Automatic Dynamic Memory Management*, John Wiley & Sons, 1996. {21}

111. P. Kacsuk, A Highly Parallel Prolog Interpreter Based on the Generalized Data Flow Model, *Proc. Intl. Logic Programming Conf.*, Uppsala University, Uppsala, 1984. {18}

112. P. Kacsuk, Generalized Data Flow Model for Programming Multiple Microprocessor Systems, *Proc. of the 3rd Symp. on Microcomp. and Microproc. Appl.*, 1983. {18}

113. P. Kacsuk, *Execution Models of Prolog for Parallel Computers*, Pitman, London, 1990. {18}

114. K. Kennedy, *Automatic Translation of Fortran Programs to Vector Form*, Tech. Rep. 467-029-4, Rice Univ., 1980. {18}

115. F. Kluźniak, *SPILL: A Specification Language Base on Logic Programming*, LOGPRO Research Rep. LITH-IDA-R-91-28, Dept. of Comp. and Inf. Sci., Linköping Univ., Linköping, 1991. {19, 124}

116. R. A. Kowalski, Predicate Logic as a Computer Language, *Information Processing 74*, pp. 569–574, North-Holland, Amsterdam, 1974. {3, 14}

117. R. A. Kowalski, *Logic for Problem Solving*, North-Holland, Amsterdam, 1979. {3}

118. C. P. Kruskal, A. Weiss, Allocating Independent Subtasks on Parallel Processors, *IEEE Trans. Software Engineering*, 11(10):1001–1016, 1985. {33}

119. D. Kuck, R. Kuhn, B. Leasure, M. Wolfe, The Structure of an Advanced Retargetable Vectorizer, *Tutorial on Supercomputers: Designs and Applications*, IEEE Computer Society Press, 1984. {18}

120. S. Le Houitouze, A New Data Structure for Implementing Extensions to Prolog, *Proc. Programming Language Implementation and Logic Programming*, LNCS 456, Springer-Verlag, 1990. {16, 21, 149}

121. H. Lieberman, C. Hewitt, A Real-Time Garbage Collector Based on the Lifetimes of Objects, *Communications of the ACM*, 26(6):419–429, June 1983. {11, 20, 22, 134, 156}

122. Y.-J. Lin, V. Kumar, A Parallel Execution Scheme for Exploiting AND-Parallelism of Logic Programs, *Proc. Intl. Conf. Parallel Processing*, 1986 {14}

123. Y.-J. Lin, V. Kumar, C. Leung, An Intelligent Backtracking Algorithm for Parallel Execution of Logic Programs, *Proc. Intl. Conf. Logic Programming*, MIT Press, 1986. {14}

124. Y.-J. Lin, V. Kumar, AND-Parallel Execution of Logic Programs on a Shared Memory Multiprocessor: A Summary of Results, *Proc. Intl. Conf. Symp. Logic Programming*, MIT Press, 1988. {14}

125. T. Lindgren, *The Compilation and Execution of Recursion-Parallel Prolog on Shared-Memory Multiprocessors*, PhL thesis, Uppsala Theses in Computer Science 18, 1993. {52, 64}

126. T. Lindgren, Compiling for Nested Recursion-Parallelism, *Parallelism and Implementation Technology for (Constraint) Logic Programming Languages* workshop at JICSLP'96, 1996. {8, 80, 82}

127. J. W. Lloyd, *Foundations of Logic Programming* (2nd ed.), Springer-Verlag, 1987. {3}

128. J. W. Lloyd, R. W. Topor, Making Prolog more Expressive, *Journal of Logic Programming*, 1(3):225–240. {19, 124}

129. E. Lusk, E. R. Butler, T. Diss, R. Olson, R. Overbeek, R. Stevens, D. H. D. Warren, A. Calderwood, P. Szeredi, S. Haridi, P. Brand, M. Carlsson, A. Ciepielewski, B. Hausman, The Aurora OR-Parallel Prolog System, *Proc. Intl. Conf. Fifth Generation Computer Systems*, 1988. {22}

130. E.P. Markatos & T.J. LeBlanc, Using Processor Affinity in Loop Scheduling on Shared-Memory Multiprocessors, Tech. Rep. 410, University of Rochester, March 1992. {33}

131. J. McCarthy, Recursive Functions of Symbolic Expressions and their Computation by Machine, *Communications of the ACM*, 3(4):184–195, 1960. {20}

132. J.M. Mellor-Crummey & M.L. Scott, Algorithms for Scalable Synchronization on Shared-Memory Multiprocessors, *ACM Transactions on Computer Systems*, 9(1):21–65, 1991. {30, 153}

133. M. Meier, Recursion vs. Iteration in Prolog, *Proc. Intl. Conf. Logic Programming*, MIT Press, 1991. {19, 124}

134. M. Metcalf, J. Reid, *Fortran 90 Explained*, Oxford Univ. Press, 1990. {105}

135. H. Millroth, *Reforming the Compilation of Logic Programs*, PhD thesis, Comp. Sci. Dept., Uppsala Univ., Uppsala, 1991. {16, 75, 124}

136. H. Millroth, Reforming Compilation of Logic Programs, *Proc. Intl. Conf. Logic Programming*, MIT Press, 1991. {6, 16, 62, 88, 98}

137. H. Millroth, Reform Compilation for Non-Linear Recursion, *Proc. Intl. Conf. Logic Programming and Automated Reasoning*, LNCS 624, Springer-Verlag, 1992. {16}

138. H. Millroth, SLDR-Resolution: Parallelizing Structural Recursion in Logic Programs, *Journal of Logic Programming*, 25(2):93–117, 1995. {6, 16, 26, 46, 62}

139. M. Minsky, *A LISP Garbage Collector Algorithm using Serial Secondary Storage*, A.I. Memo 58, Massachusetts Institute of Technology Project MAC, Cambridge, Massachusetts, 1963. {20}

140. J. Montelius, K. Ali, An And/Or-Parallel implementation of AKL, *New Generation Computing*, 13(4):31–52, 1995. {15, 81}

141. D. Moon, Garbage Collection in a Large Lisp System, *ACM Symp. Lisp and Functional Programming*, 1984. {12}

142. T. Moto-oka, *et al.* [sic], Challenge for Knowledge Information Processing Systems, *Proc. Intl. Conf. on Fifth Generation Computer Systems*, 1981. {4}

143. T. Moto-oka, K. Fuchi, The Architectures in the Fifth Generation Computers, *Proc. IFIP'83*, 1983. {18}

144. F. Morris, A Time- and Space- Efficient Compaction Algorithm, *Communications of the ACM*, 12(9):662–665, August 1978. {20, 128}

145. K. Muthukumar, M. V. Hermenegildo, Compile-Time Derivation of Variable Dependency using Abstract Interpretation, *Journal of Logic Programming*, 13(1):315–347, 1992. {14}

146. L. Naish, Parallelizing NU-Prolog, *Logic Programming: Proc. of the Fifth Intl. Conf. and Symp.*, MIT Press, 1988. {15, 27, 30, 31, 47, 63, 89}

147. U. Neumerkel, Extensible Unification by Metastructures, *Proc. of META'90 workshop*, 1990. {16}

148. M. Nilsson, H. Tanaka, A Flat GHC Implementation for Supercomputers, *Logic Programming: Fifth Intl. Conf. Symp.*, MIT Press, 1988. {18}

149. M. Nilsson, H. Tanaka, Massively Parallel Implementation of Flat GHC on the Connection Machine, *Proc. Intl. Conf. Fifth Generation Computer Systems*, Ohmsha, 1992. {18}

150. W. J. Older, J. A. Rummell, An Incremental Garbage Collector for WAM-Based Prolog, *Proc. Joint Intl. Conf. Symp. Logic Programming*, MIT Press, Cambridge, Mass., 1992. {20, 129}

151. D. Palmer, L. Naish, NUA-Prolog: An Extension to the WAM for Parallel Andorra, *Proc. 8th Intl. Conf. Logic Programming*, MIT Press, 1991. {15, 56}

152. D. Palmer, J. Prins, S. Westfold, Work-efficient Nested Data Parallelism, *Frontiers of Massively Parallel Computing*, 1994. {75}

153. D. A. Patterson & J. L. Hennessy, *Computer Organization & Design: The Hardware/Software Interface*, Morgan Kaufmann Publ., 1993. {72}

154. G. H. Pollard, *Parallel Execution of Horn Clause Programs*, PhD thesis, Imperial College, London, 1981. {14}

155. C. D. Polychronopoulos, D. J. Kuck, Guided Self-Scheduling: A Practical Scheduling Scheme for Parallel Supercomputers, *IEEE Transactions on Computers*, 36(12):1425–1439, 1987. {33}

156. E. Pontelli, G. Gupta, Data Parallel Logic Programming in &ACE, *Proc. of the IEEE Intl. Symp. on Parallel and Distributed Processing*, IEEE, 1995. {17, 81}

157. E. Pontelli, G. Gupta, Determinacy Driven Optimization of And-Parallel Prolog Implementations, *Intl. Conf. Logic Programming*, MIT Press, 1995. {17, 81}

158. E. Pontelli, G. Gupta, *Nested Parallel Call Optimisation*, Tech. Rep., New Mexico State University, {17, 81}

159. E. Pontelli, G. Gupta, D. Tang, M. Carro, M. V. Hermenegildo, Improving the Efficiency of Nondeterministic Independent And-parallel Systems, *Journal of Computer Languages*, 22(2-3), 1996. {17, 81}

160. J. A. Robinson, A Machine-oriented Logic Based on the Resolution Principle, *Journal of the ACM*, 12(1):23–41, 1965. {3}

161. N. Röjemo, A Generational Garbage Collector for a Parallel Graph Reducer, *Intl. Workshop on Memory Management*, LNCS 637, Springer-Verlag, 1992

162. D. Sahlin, *Making Garbage Collection Independent of the Amount of Garbage*, Tech. Rep. R87008, Swedish Institute of Computer Science, 1987. {21, 130, 143}

163. V. Santos Costa, *Compile-Time Analysis for the Parallel Execution of Logic Programs in Andorra-I*, PhD thesis, University of Bristol, 1993. {15}

164. V. Santos Costa, D. H. D. Warren, R. Yang, Andorra-I: A Parallel Prolog System that Transparently Exploits both And- and Or-Parallelism, *Third ACM SIGPLAN Symp. on Principles & Practices of Parallel Programming*, 1991. {15, 43, 56, 57}

165. V. Santos Costa, D. H. D. Warren, R. Yang, The Andorra-I Preprocessor: Supporting Full Prolog on the Basic Andorra Model, *Logic Programming: Proc. of the Eighth Intl. Conf.*, MIT Press, 1991. {15}

166. V. Santos Costa, D. H. D. Warren, R. Yang, The Andorra-I Engine: a Parallel Implementation of the Basic Andorra Model, *Logic Programming: Proc. of the Eighth Intl. Conf.*, MIT Press, 1991. {15}

167. T. Sato, H. Tamaki, First Order Compiler: a Deterministic Logic Program Synthesis Algorithm, *Journal of Symbolic Computation*, 8(6):605–627, 1989. {19, 124}

168. V. A. Saraswat, K. Kahn, J. Levy, Janus: A Step Towards Distributed Constraint Programming, *North American Conf. Logic Programming*, MIT Press, 1990. {5}

169. H. Schorr , W. M. Waite, An Efficient Machine-Independent Procedure for Garbage Collection in Various List Structures, *Communications of the ACM*, 10(8):501–506, 1967. {20, 128, 145}

170. J. T. Schwartz, R. B. K. Devar, E. Dubinski, E. Schonberg, *Programming with Sets: an Introduction to SETL*, Springer-Verlag, 1986. {19, 124}

171. D. C. Sehr, L. V. Kale, D. A. Padua, Loop Transformations for Prolog Programs, LNCS 768, Springer-Verlag. {17}

172. D. C. Sehr, *Automatic Parallelization of Prolog Programs*, PhD thesis, Univ. of Illinois at Urbana Champaign, 1992. {17}

173. E. Y. Shapiro, *A Subset of Concurrent Prolog and its Interpreter*, ICOT Tech. Rep. TR-003, Institute for New Generation Computing Technology, Tokyo, 1983. {5, 16}

174. E. Y. Shapiro, The Family of Concurrent Logic Programming Languages, *ACM Computing Surveys*, 21(3):413–510, 1989. {5}

175. E. Y. Shapiro, A Subset of Concurrent Prolog and Its Interpreter, *Concurrent Prolog: Collected Papers*, vol. 1, MIT Press, 1987. {5, 16}

176. K. Shen, Exploiting Dependent And-Parallelism in Prolog: the Dynamic Dependent And-Parallel Scheme (DDAS), *Proc. Joint Intl. Symp. Logic Programming*, MIT Press, 1992. {14, 15, 81, 82}

177. K. Shen, *Studies of And-Or Parallelism*, PhD thesis, Cambridge University, revised June 1992. {14, 15, 81}

178. K. Shen, Implementing Dynamic Dependent And-parallelism, *Proc. 10th Intl. Conf. Logic Programming*, MIT Press, 1993. {14, 15, 81}

179. K. Shen, Initial Results of the Parallel Implementation of DASWAM, *Proc. Joint Intl. Conf. Symp. Logic Programming*, MIT Press, 1996. {15}

180. K. Shen, personal communication, 1996. {15}

181. J. P. Singh, J. L. Hennessy, An Empirical Investigation of the Effectiveness and Limitations of Automatic Parallelization, *Proc. Intl. Symp. on Shared Memory Multiprocessing*, 1991 {2, 18}

182. D. A. Smith, MultiLog: Data Or-parallel Logic Programming, *Intl. Conf. Logic Programming*, MIT Press, 1993. {16}

183. Z. Somogyi, K. Ramamohanarao, J. Vaghani, A Stream AND-parallel Execution Algorithm with Backtracking, *Proc. Fifth Intl. Conf. Symp. Logic Programming*, MIT Press, 1988. {14}

184. G. L. Steele Jr., *Common LISP: The Language*, Digital Press, 1984. {20, 124}

185. G. L. Steele Jr., W. D. Hillis, *Connection Machine Lisp: Fine-Grained Parallel Symbolic Processing*, Tech. Rep. PL86–2, Thinking Machines Corporation, 1986. {18}

186. L. Sterling, E. Shapiro, *The Art of Prolog*, MIT Press, 1986 {3}

187. P. Tang & P.-C. Yew, Processor Self-Scheduling for Multiple Nested Parallel Loops, *Proc. 1986 Intl. Conf. Parallel Processing*, August 1986. {33, 51, 90}

188. A. Taylor, *High Performance Prolog Implementation*, PhD thesis, Basser Department of Computer Science, Sydney University, 1991. {4}

189. R. D. Tennent, Quantification in Algol-like Languages, *Information Processing Letters*, 25:133–137, 1987. {20, 100, 125}

190. R. D. Tennent, *Semantics of Programming Languages*, Prentice-Hall International, 1991. {20, 100, 125}

191. Thinking Machines Corporation, *Connection Machine: ⋆Lisp Dictionary*, Cambridge, Massachusetts, 1991. {18}

192. E. Tick, Memory- and Buffer-Referencing Characteristics of a WAM-based Prolog, *Journal Logic Programming*, 11(1-2):133–162, 1991. {71}

193. E. Tick, *Parallel Logic Programming*, MIT Press, 1991. {5}

194. E. Tick, The Deevolution of Concurrent Logic Programming Languages, *Journal of Logic Programming*, 23(2):89–124, 1995. {16}

195. P. Tinker, G. Lindstrom, A Performance-Oriented Design for OR-Parallel Logic Programming, *Proc. Intl. Conf. Logic Programming*, MIT Press, 1987. {147}

196. H. Touati, T. Hama, A Light-Weight Prolog Garbage Collector, *Proc. Intl. Conf. on Fifth Generation Computing Systems*, 1988. {20, 129}

197. D. A. Turner, Miranda: a Non-Strict Functional Language with Polymorphic Types, *Functional Programming Languages and Computer Architecture*, LNCS 201, Springer-Verlag, 1985. {123}

198. S.-Å. Tärnlund, Horn Clause Computability, *BIT*, 17(2):215–226, 1977. {99}

199. S.-Å. Tärnlund, *Logic information processing*, TRITA-IBADB 1034, Department of Information Processing and Computer Science, Royal Institute of Technology and University of Stockholm, 1975. {16, 17}

200. S.-Å. Tärnlund, Reform, unpublished manuscript, Computing Science Department, Uppsala University, 1991. {6, 16, 26, 46}

201. K. Ueda, *Guarded Horn clauses*, ICOT Tech. Rep. TR-103, Institute for New Generation Computing Technology, Tokyo, 1985. {16}

202. K. Ueda, *Guarded Horn clauses*, PhD thesis, Faculty of Engineering, Univ. of Tokyo, 1986. {16}

203. K. Ueda, Guarded Horn clauses, *Concurrent Prolog: Collected Papers*, vol. 1, MIT Press, 1987. {5}

204. K. Ueda, M. Morita, Moded Flat GHC and Its Message-Oriented Implementation Technique, *New Generation Computing*, 13(1):3–43, 1994. {5}

205. K. Ueda, K. Furukawa, Transformation Rules for GHC Programs, *Proc. Intl. Conf. on fifth Generation Computer Systems*, ICOT, 1988. {16}

206. S. Umeyama, K. Tamura, A Parallel Execution Model of Logic Programs, *Proc. Intl. Symp. Comp. Arch.*, 1983. {18}

207. D. M. Ungar, Generation Scavenging: A Non-Disruptive High Performance Storage Reclamation Algorithm, *ACM SIGPLAN Notices*, 19(5):157–167, 1984. {11, 12, 20}

208. P. L. Van Roy, *Can Logic Programming Execute as Fast as Imperative Programming?*, PhD thesis, UCB/CSD 90/600, Computer Science Division (EECS), University of California, Berkeley, 1990. {4}

209. P. L. Van Roy, 1983–1993: The Wonder Years of Sequential Prolog Implementation, *Journal of Logic Programming*, 19,20:385–441, 1994. {4}

210. P. L. Van Roy, A. Despain, The Benefits of Global Dataflow Analysis for an Optimizing Prolog Compiler, *Proc. NACLP'90*, 1990. {4, 64}

211. A. Voronkov, Logic Programming with Bounded Quantifiers, *Logic Programming*, LNAI 592, Springer-Verlag, 1992. {9, 18, 85, 100, 103, 123}

212. A. Voronkov, *Logic Programming with Bounded Quantifiers*, Tech. Rep. ECRC-92-29, ECRC, Munich, 1992. {9, 18, 85, 100, 103, 123}

213. D. H. D. Warren, *Implementing Prolog—Compiling Predicate Logic Programs*, DAI Tech. Rep. 39-40, Edinburgh University, 1977. {3}

214. D. H. D. Warren, *An Abstract Prolog Instruction Set*, Report 309, SRI International, Menlo Park, California, USA, 1983. {3, 50, 65, 71, 77, 87, 100, 107, 127, 130, 142}

215. D. H. D. Warren, The SRI-model for Or-Parallel Execution of Prolog, *1987 IEEE Intl. Symp. Logic Programming*, IEEE Press, 1987. {22}

216. D. H. D. Warren, The Andorra Model, presented at Gigalips workshop, University of Manchester, 1988. {15}

217. D. H. D. Warren, F. C. N. Pereira, L. M. Pereira, Prolog—The Language and its Implementation Compared With LISP, *SIGPLAN Notices*, 12(8), 1977. {3}

218. D. S. Warren, Memoing for Logic Programs, *Communications of the ACM*, 35(3):93–111, 1992. {20, 125}

219. R. A. Warren, M. V. Hermenegildo, Experiments with Prolog: An Overview, ACA/PP SRS Tech. Note 43, Computer Technology Corporation, Austin, 1987. {17}

220. P. Weemeeuw, B. Demoen, Garbage Collection in Aurora: An overview, *Proc. Intl. Workshop on Memory Management 92*, LNCS 637, Springer-Verlag, Berlin, 1992. {22}

221. P. R. Wilson, Uniprocessor Garbage Collection Techniques, *Proc. Intl. Workshop on Memory Management 92*, LNCS 637, Springer-Verlag, Berlin, 1992. {21}

222. M. J. Wise, *Prolog Multiprocessors*, Prentice-Hall, 1986. {15}

223. K. L. Wrench, *A Distributed AND-Or Parallel Prolog Network*, Tech. Rep. 212, University of Cambridge Computer Laboratory, 1990. {15}

224. R. Yang, *A Parallel Logic Programming Language and Its Implementation*, PhD thesis, Department of Electrical Engineering, Keio University, Yokohama, 1986. {15}

225. R. Yang, T. Beaumont, I. Dutra, V. Santos Costa, D. H. D. Warren, Performance of the Compiler-Based Andorra-I System, *Logic Programming: Proc. Tenth Intl. Conf. Logic Programming*, MIT Press, 1993. {15}

226. E. Yardeni, S. Kliger, E. Shapiro, The Languages FCP(:) and FCP(:,?), *New Generation Computing*, 7(2):98–107, 1990. {5}

Computing Science Department
Uppsala University
Box 311, S–751 05 Uppsala, Sweden

Uppsala Theses in Computer Science 25
ISSN 0283–359X
ISBN 91–506–1177–1