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.

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 [151] 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 [156] for an introduction to Prolog, and Kowalski's book [95] for an introduction to programming in logic. Finally, the textbook by Lloyd [103] is recommended for an introduction to the theory behind logic programming.

Prolog has its origin in Robinson's [130] Resolution Principle for automated theorem proving from the early 1960's. Colmerauer, Kanoui, Pasero and Roussel [46] developed the first Prolog interpreter in 1972. Kowalski [94] showed how logic can be used for programming using declarative and procedural readings of clauses.

Warren [178, 182] 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" [179] 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 [173, 175] Aquarius Prolog uses a WAM-inspired abstract machine (Berkley Abstract Machine). Taylor's [157] 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 [174] "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 [115] 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 [143, 144], FCP [145, 191], Parlog [43, 44], (F)GHC [168, 169], Strand [61] and Janus [138].

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 [161, 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* [111, 109] (this can be viewed as an implementation technique for the Reform inference system [165]). 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 [32], 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 [27].

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 [102].

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, 23] and Voronkov [176, 177] 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

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

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

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 quan-

tifications 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 [54, 172]. This insight has led to the invention of generational garbage collection [98] 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 [172, 114]. 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 [52] have improved the scheme further using ideas from Bekkers, Ridoux and Ungaro [25].

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 [125] where execution of pure OR-parallelism is discussed. The possibility of AND-parallel execution was noted by Kowalski [94] 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 [47] 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 [99, 100, 101] 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 [153] 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 [41].

DeGroot [51] 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 [76] refined DeGroot's scheme and added a backtracking semantics. Hermenegildo [75, 79, 80, 118] defined and implemented the first independent AND-parallel system: &-Prolog. &-Prolog has played an important role for AND-parallel implementations. System such as ACE [65, 66] and DDAS [148, 146, 147] are largely based on &-Prolog.

Chang, Despain and DeGroot [37, 38] were first to propose that scheduling of parallel execution could be done through compile time analysis. Borgwardt [30, 31] proposed a stack-based implementation of the execution model developed by Chang et al. [38].

Hermenegildo and Rossi [80] 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 [189] 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 [119, 124] 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 [70, 181] where deterministic goals are executed in parallel, and non-deterministic goals are suspended. The Basic Andorra Model has been implemented in Andorra-I [134, 135, 136, 190].

In the Extended Andorra Model [71, 133], 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 [86, 87, 113].

Gupta, Santos Costa, Yang, and Hermenegildo [64] combine independent AND-parallelism, deterministic dependent AND-parallelism and OR-parallelism in IDIOM. This built on ideas from AO-WAM [67], which is an extension of RAP-WAM [76] 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 [187] and Wrench's APPNet [188].

Shen has developed a scheme for general dependent AND-parallelism called DDAS [146, 147, 148, 149]. 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 [150] mentions work in progress.

Another approach to parallelism is to introduce explicit concurrency. This was done in IC-Prolog by Clark and McCabe [41] and in Relational Language by Clark and Gregory [42]. These ideas were adopted by others and resulted in the concurrent logic languages: Concurrent Prolog [143], FCP [145], Parlog [43, 49], Strand [61], Guarded Horn Clauses (GHC) [166, 167], and Flat GHC [40, 170]. These languages are also referred to as de-evolutioned by Tick [162] 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 [77] recently proposed an elegant scheme where attributed variables [120, 97] 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 [152] 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 [164] 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 [165], based on these ideas.

Inspired by Tärnlund's ideas Millroth developed *Reform Compilation* [108, 109, 110, 111] 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 [184] 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 [73] 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 [142, 141] 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 [78] and Debray and Jain [50] discuss how goals can be spawned more efficiently using program transformation techniques. These have some similarities to the transformations described by Tärnlund [164] 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 [68, 126, 127, 128, 129] 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 [89, 90, 91]. This in turn was inspired by other data flow models for parallel execution of Prolog proposed by Moto-oka and Fuchi [116] and Umeyama and Tamura [171]. These execution models were motivated from the research on data flow parallelism which emerged in the early 1980's [12, 53].

Nilsson and Tanaka [121, 122] 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 [19, 20] 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 [91], Barklund and Millroth in Nova Prolog [22], and by Fagin [59]. The idea of having explicitly parallel data structures have previously been used in other languages such as APL [85], CM Lisp [82, 155], $\star$Lisp [160], NESL [29] 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 [96], Kennedy [92], Burke and Cytron [33] 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 [151] 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 [176, 177] 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 [93], 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 [140] and {log} [55].

Barklund & Hill [21] have studied how to incorporate restricted quantifications and arrays in Gödel [81], while Apt [10] has studied how bounded quantifications and arrays could be used also in constraint based languages.

Lloyd & Topor [104] 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 [137] 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 [107] 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 [154] (and some earlier LISP dialects), as well as Standard ML [72], 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 [158, 159], and also by an exposition by Gries [63].

Hermenegildo and Carro [78] 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 [26, 28, 62] and has also been applied in logic programming [18, 183].

Garbage Collection of Prolog

Sequential Collection It has been observed in other languages that most data tend to be short lived [54, 172]. This insight led to the invention of generational garbage collection [7, 98] where young objects are garbage collected more often than old. Copying collectors [60, 112] and generational copying collectors have been considered unsuitable for Prolog until recently. Prolog implementations such as SICStus Prolog [35] use a mark-sweep algorithm [106] 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 [45, 139] algorithm for marking and on Morris' algorithm [45, 117] for compacting.

Touati and Hama [163] 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 [24] and later by Older and Rummell [123].

Bekkers, Ridoux and Ungaro [25] 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 [36, 97], 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 [39] 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 [132] 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. [52] 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 [45], Appel [8], Jones and Lins [88], and Wilson [186] 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 [25].

Parallel Collection Most parallel Prolog implementations do not include a garbage collector. OR-parallel systems such as Muse [4] and Aurora [34, 105] use more or less sequential mark-sweep collectors [2, 56, 57]. The Aurora collector [185] designed by Weemeeuw and Demoen is slightly more complicated since Aurora uses the SRI-model [180] 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 [48] 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 [84] describe a parallel stop-and-copy collector based on Cheney's [39] 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 [69] 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 [74] developed a concurrent lock-free version of Halstead's algorithm.

Lieberman and Hewitt [98] 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' [83] developed a concurrent copying garbage collector for shared memory with two processes; collector and mutator. Ellis,

Li and Appel [58] propose a similar design with several mutators and one collector. Röjemo [131] extended the collector by Ellis et al. for the $\langle v, \mathrm{G}\rangle$-machine [13] (a parallel version of the G-machine [14]).

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}

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}

6. A. W. Appel, A Runtime System, *Lisp and Symbolic Computation*, 3(4):343–380, 1990. {21}

7. A. W. Appel, Simple Generational Garbage Collection and Fast Allocation, *Software—Practice and Experience*, 19(2):171–183, 1989. {20, 21}

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}

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}

11. H. Arro, J. Barklund, J. Bevemyr, Parallel Bounded Quantifications — Preliminary Results, *ACM SIGPLAN Notices*, 28(5):117–124, 1993. {20}

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 $\langle v, G \rangle$-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}

18. J. Barklund, *Tabulation of Functions in Definite Clause Programs*, UPMAIL Tech. Rep. 82, Comp. Sci. Dept., Uppsala Univ.,1994. {20}

19. J. Barklund, N. Hagner, M. Wafin, KL1 in Condition Graphs on a Connection Machine, *Proc. Intl. Conf. Fifth Generation Computer Systems*, Ohmsha, 1988. {18}

20. J. Barklund, N. Hagner, M. Wafin, *Connection Graphs*, UPMAIL Tech. Rep. 48, Comp. Sci. Dept., Uppsala Univ., Uppsala, 1988. {18}

21. 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}

22. J. Barklund, H. Millroth, *Nova Prolog*, UPMAIL Tech. Rep. 52, Comp. Sci. Dept., Uppsala Univ., 1988. {18}

23. J. Barklund, H. Millroth, Providing Iteration and Concurrency in Logic Programs Through Bounded Quantifications, *Intl. Conf. on Fifth Generation Computer Systems*, 1992. {9}

24. J. Barklund, H. Millroth, Garbage Cut for Garbage Collection of Iterative Prolog Programs, *3rd Symp. on Logic Programming*, IEEE, 1986. {20}

25. 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}

26. R. E. Bellman, *Dynamic Programming*, Princeton Univ. Press, Princeton, N.J., 1957. {20}

27. J. Bevemyr, T. Lindgren, H. Millroth, Reform Prolog: The Language and its Implementation, *Logic Programming: Proc. Tenth Intl. Conf.*, MIT Press, 1993. {8}

28. R. S. Bird, Tabulation Techniques for Recursive Programs, *ACM Computing Surveys*, 12(4):403–417, 1980. {20}

29. G. Blelloch, Programming Parallel Algorithms, *Communications of the ACM*, 39(3), 1996. {18}

30. P. Borgwardt, Parallel Prolog Using Stack Segments on Shared-Memory Multiprocessors, *IEEE Symp. Logic Programming*, 1984. {14}

31. P. Borgwardt, D. Rea, Distributed Semi-Intelligent Backtracking for a Stack Based AND-Parallel Prolog, *Proc. Symp. Logic Programming*, IEEE Computer Society, 1986. {14}

32. H. Burkhardt, S. Frank, B. Knobe, J. Rothnie, *Overview of the KSR1 Computer System*, Tech. Rep. KSR-TR-9202001, Kendall Square Research, 1992. {7}

33. M. Burke, R. Cytron, Interprocedural Dependence Analysis and Parallelization, *Proc. SIGPLAN'86 Symp. Compiler Construction*, 1986. {18}

34. M. Carlsson, *Design and Implementation of an Or-Parallel Prolog Engine*, PhD thesis, SICS-RITA/02, 1990. {22}

35. M. Carlsson, *SICStus Prolog Internals Manual*, Internal Report, Swedish Institute of Computer Science, 1989. {20}

36. M. Carlsson, *Variable Shunting for the WAM*, Tech. Rep. R91-07, Swedish Institute of Computer Science, 1989. {21}

37. 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}

38. 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}

39. C. J. Cheney, A Nonrecursive List Compacting Algorithm, *Communications of the ACM*, 13(11):677–678, November 1970. {21, 22}

40. T. Chikayama, Y. Kimura, Multiple Reference Management in Flat GHC, *Logic Programming—Proc. Fourth Intl. Conf.*, MIT Press, 1987. {16}

41. 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}

42. K. L. Clark, S. Gregory, A Relational Language for Parallel Programming, *Proc. ACM Symp. Functional Programming and Computer Architecture*, 1981. {16}

43. K. L. Clark, S. Gregory, *PARLOG: A Parallel Logic Programming Language*, Rep. DOC 83/5, Dept. Computing, Imperial College, London, 1983. {5, 16}

44. K. L. Clark, S. Gregory, Notes on the Implementation of PARLOG, *Journal of Logic Programming*, 2(1):17–42, 1985. {5}

45. J. Cohen, Garbage Collection of Linked Data Structures, *Computing Surveys*, 13(3):341–367, September 1981. {20, 21}

46. 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}

47. J. S. Conery, D. F. Kibler, Parallel Interpretation of Logic Programs, *Proc. ACM Symp. Functional Programming and Computer Architecture*, 1981. {14}

48. J. Crammond, A Garbage Collection Algorithm for Shared Memory Parallel Processors, *Intl. Journal of Parallel Programming*, 17(6):497–522, 1988. {22}

49. J. Crammond, The Abstract Machine and Implementation of Parallel Parlog, *New Generation Computing*, 10(4):385–422, Springer-Verlag, 1992. {16}

50. S. K. Debray, M. Jain, A Simple Program Transformation for Parallelism, *Intl. Symp. Logic Programming*, MIT Press, 1994. {17}

51. D. DeGroot, Restricted AND-parallelism, *Proc. Intl. Conf. Fifth Generation Computer Systems*, North-Holland, Amsterdam, 1984. {14}

52. 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}

53. J. B. Dennis, Data Flow Supercomputers, *IEEE Computer*, 13(11):48–56, 1980. {18}

54. L. P. Deutsch, D. G. Bobrow, An Efficient Incremental Automatic Garbage Collector, *Communications of the ACM*, 19(9):522–526, 1976. {11, 20}

55. 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}

56. M. Doprochevsky, *Garbage Collection in the OR-Parallel Logic Programming System ElipSys*, ECRC Tech. Rep. DPS-85, 1991. {22}

57. 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}

58. 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}

59. B. Fagin, *Data-Parallel Logic Programming Systems*, Tech. Rep., Thayer School of Engineering, Dartmouth Collage, New Hampshire, 1990. {18}

60. R. Fenichel, J. Yochelson, A LISP Garbage-collector for Virtual- memory Computer Systems, *Communications of the ACM*, 12(11):611–612, 1969.

61. I. Foster, S. Taylor, Strand: A Practical Parallel Programming Language, *North American Conf. Logic Programming*, MIT Press, 1989. {5, 16}

62. D. P. Friedman, D. S. Wise, M. Wand, Recursive Programming through Table Look-Up, *Symp. on Symbolic and Algebraic Computation*, ACM, 1976. {20}

63. D. Gries, *The Science of Programming*, Springer-Verlag, 1981. {20}

64. 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}

65. 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}

66. 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}

67. G. Gupta, B. Jayaraman, Combined And-Or Parallelism on Shared Memory Multiprocessors, *North American Conf. Logic Programming*, MIT Press, 1989. {15}

68. G. Gupta, E. Pontelli, Last Alternative Optimization, *8th IEEE Symp. on Parallel and Distributed Processing*, IEEE Computer Society, 1996. {17}

69. R. H. Halstead, Implementation of Multilisp: Lisp on a Multiprocessor, *ACM Symp. LISP and Functional Programming*, 1984. {22}

70. S. Haridi, A Logic Programming Language Based on the Andorra Model, *New Generation Computing*, 7(2-3):109–125, 1990. {15}

71. S. Haridi, S. Janson, Kernel Andorra Prolog and its Computation Model, *Intl. Conf. Logic Programming*, MIT Press, 1990. {15}

72. R. Harper, D. MacQueen, R. Milner, *Standard ML*, Technical Report ECS-LFCS-86-2, Dept. Computer Science, Edinburgh Univ., 1986. {20}

73. W. L. Harrison III, The Interprocedural Analysis and Parallelization of Scheme Programs, *Lisp and Symbolic Computation*, 2(3-4):176–396, 1989. {17}

74. 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}

75. 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}

76. 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}

77. 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}

78. M. V. Hermenegildo, M. Carro, Relating Data-Parallelism and (And–) Parallelism in Logic Programs, *Journal of Computer Languages*, 22(2-3), 1996. {17, 20}

79. M. V. Hermenegildo, K. J. Greene, &-Prolog and its Performance: Exploiting Independent And-Parallelism, *Proc. Intl. Conf. Logic Programming*, MIT Press, 1990. {14}

80. M. V. Hermenegildo, F. Rossi, Non-Strict Independent And-Parallelism, *Proc. Intl. Conf. Logic Programming*, MIT Press, 1990. {14, 15}

81. P. M. Hill, J. W. Lloyd, *The Gödel Programming Language*, MIT Press, 1993. {19}

82. W. D. Hillis, G. L. Steele Jr., Data Parallel Algorithms, *Communications of the ACM*, 29(12):1170–1183, 1986. {18}

83. 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}

84. 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}

85. K. E. Iverson, *A Programming Language*, Wiley, 1962. {18}

86. S. Janson, *AKL A Multiparadigm Programming Language*, PhD thesis, Comp. Sci. Dept., Uppsala Univ., Uppsala, 1994. {15}

87. 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}

88. R. Jones, R. Lins, *Garbage Collection: Algorithms for Automatic Dynamic Memory Management*, John Wiley & Sons, 1996. {21}

89. P. Kacsuk, A Highly Parallel Prolog Interpreter Based on the Generalized Data Flow Model, *Proc. Intl. Logic Programming Conf.*, Uppsala University, Uppsala, 1984. {18}

90. P. Kacsuk, Generalized Data Flow Model for Programming Multiple Microprocessor Systems, *Proc. of the 3rd Symp. on Microcomp. and Microproc. Appl.*, 1983. {18}

91. P. Kacsuk, *Execution Models of Prolog for Parallel Computers*, Pitman, London, 1990. {18}

92. K. Kennedy, *Automatic Translation of Fortran Programs to Vector Form*, Tech. Rep. 467-029-4, Rice Univ., 1980. {18}

93. 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}

94. R. A. Kowalski, Predicate Logic as a Computer Language, *Information Processing 74*, pp. 569–574, North-Holland, Amsterdam, 1974. {3, 14}

95. R. A. Kowalski, *Logic for Problem Solving*, North-Holland, Amsterdam, 1979. {3}

96. 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}

97. 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}

98. 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}

99. Y.-J. Lin, V. Kumar, A Parallel Execution Scheme for Exploiting AND-Parallelism of Logic Programs, *Proc. Intl. Conf. Parallel Processing*, 1986 {14}

100. 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}

101. 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}

102. T. Lindgren, Compiling for Nested Recursion-Parallelism, *Parallelism and Implementation Technology for (Constraint) Logic Programming Languages* workshop at JICSLP'96, 1996. {8}

103. J. W. Lloyd, *Foundations of Logic Programming* (2nd ed.), Springer-Verlag, 1987. {3}

104. J. W. Lloyd, R. W. Topor, Making Prolog more Expressive, *Journal of Logic Programming*, 1(3):225–240. {19}

105. 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}

106. J. McCarthy, Recursive Functions of Symbolic Expressions and their Computation by Machine, *Communications of the ACM*, 3(4):184–195, 1960. {20}

107. M. Meier, Recursion vs. Iteration in Prolog, *Proc. Intl. Conf. Logic Programming*, MIT Press, 1991. {19}

108. H. Millroth, *Reforming the Compilation of Logic Programs*, PhD thesis, Comp. Sci. Dept., Uppsala Univ., Uppsala, 1991. {16}

109. H. Millroth, Reforming Compilation of Logic Programs, *Proc. Intl. Conf. Logic Programming*, MIT Press, 1991. {6, 16}

110. H. Millroth, Reform Compilation for Non-Linear Recursion, *Proc. Intl. Conf. Logic Programming and Automated Reasoning*, LNCS 624, Springer-Verlag, 1992. {16}

111. H. Millroth, SLDR-Resolution: Parallelizing Structural Recursion in Logic Programs, *Journal of Logic Programming*, 25(2):93–117, 1995. {6, 16}

112. 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}

113. J. Montelius, K. Ali, An And/Or-Parallel implementation of AKL, *New Generation Computing*, 13(4):31–52, 1995. {15}

114. D. Moon, Garbage Collection in a Large Lisp System, *ACM Symp. Lisp and Functional Programming*, 1984. {12}

115. T. Moto-oka, *et al.* [sic], Challenge for Knowledge Information Processing Systems, *Proc. Intl. Conf. on Fifth Generation Computer Systems*, 1981. {4}

116. T. Moto-oka, K. Fuchi, The Architectures in the Fifth Generation Computers, *Proc. IFIP'83*, 1983. {18}

117. F. Morris, A Time- and Space- Efficient Compaction Algorithm, *Communications of the ACM*, 12(9):662–665, August 1978. {20}

118. K. Muthukumar, M. V. Hermenegildo, Compile-Time Derivation of Variable Dependency using Abstract Interpretation, *Journal of Logic Programming*, 13(1):315–347, 1992. {14}

119. L. Naish, Parallelizing NU-Prolog, *Logic Programming: Proc. of the Fifth Intl. Conf. and Symp.*, MIT Press, 1988. {15}

120. U. Neumerkel, Extensible Unification by Metastructures, *Proc. of META'90 workshop*, 1990. {16}

121. M. Nilsson, H. Tanaka, A Flat GHC Implementation for Supercomputers, *Logic Programming: Fifth Intl. Conf. Symp.*, MIT Press, 1988. {18}

122. M. Nilsson, H. Tanaka, Massively Parallel Implementation of Flat GHC on the Connection Machine, *Proc. Intl. Conf. Fifth Generation Computer Systems*, Ohmsha, 1992. {18}

123. 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}

124. D. Palmer, L. Naish, NUA-Prolog: An Extension to the WAM for Parallel Andorra, *Proc. 8th Intl. Conf. Logic Programming*, MIT Press, 1991. {15}

125. G. H. Pollard, *Parallel Execution of Horn Clause Programs*, PhD thesis, Imperial College, London, 1981. {14}

126. E. Pontelli, G. Gupta, Data Parallel Logic Programming in &ACE, *Proc. of the IEEE Intl. Symp. on Parallel and Distributed Processing*, IEEE, 1995. {17}

127. E. Pontelli, G. Gupta, Determinacy Driven Optimization of And-Parallel Prolog Implementations, *Intl. Conf. Logic Programming*, MIT Press, 1995. {17}

128. E. Pontelli, G. Gupta, *Nested Parallel Call Optimisation*, Tech. Rep., New Mexico State University, {17}

129. 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}

130. J. A. Robinson, A Machine-oriented Logic Based on the Resolution Principle, *Journal of the ACM*, 12(1):23–41, 1965. {3}

131. N. Röjemo, A Generational Garbage Collector for a Parallel Graph Reducer, *Intl. Workshop on Memory Management*, LNCS 637, Springer-Verlag, 1992

132. D. Sahlin, *Making Garbage Collection Independent of the Amount of Garbage*, Tech. Rep. R87008, Swedish Institute of Computer Science, 1987. {21}

133. V. Santos Costa, *Compile-Time Analysis for the Parallel Execution of Logic Programs in Andorra-I*, PhD thesis, University of Bristol, 1993. {15}

134. 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}

135. 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}

136. 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}

137. T. Sato, H. Tamaki, First Order Compiler: a Deterministic Logic Program Synthesis Algorithm, *Journal of Symbolic Computation*, 8(6):605–627, 1989. {19}

138. V. A. Saraswat, K. Kahn, J. Levy, Janus: A Step Towards Distributed Constraint Programming, *North American Conf. Logic Programming*, MIT Press, 1990. {5}

139. 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}

140. J. T. Schwartz, R. B. K. Devar, E. Dubinski, E. Schonberg, *Programming with Sets: an Introduction to SETL*, Springer-Verlag, 1986. {19}

141. D. C. Sehr, L. V. Kale, D. A. Padua, Loop Transformations for Prolog Programs, LNCS 768, Springer-Verlag. {17}

142. D. C. Sehr, *Automatic Parallelization of Prolog Programs*, PhD thesis, Univ. of Illinois at Urbana Champaign, 1992. {17}

143. 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}

144. E. Y. Shapiro, The Family of Concurrent Logic Programming Languages, *ACM Computing Surveys*, 21(3):413–510, 1989. {5}

145. E. Y. Shapiro, A Subset of Concurrent Prolog and Its Interpreter, *Concurrent Prolog: Collected Papers*, vol. 1, MIT Press, 1987. {5, 16}

146. 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}

147. K. Shen, *Studies of And-Or Parallelism*, PhD thesis, Cambridge University, revised June 1992. {14, 15}

148. K. Shen, Implementing Dynamic Dependent And-parallelism, *Proc. 10th Intl. Conf. Logic Programming*, MIT Press, 1993. {14, 15}

149. K. Shen, Initial Results of the Parallel Implementation of DASWAM, *Proc. Joint Intl. Conf. Symp. Logic Programming*, MIT Press, 1996. {15}

150. K. Shen, personal communication, 1996. {15}

151. 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}

152. D. A. Smith, MultiLog: Data Or-parallel Logic Programming, *Intl. Conf. Logic Programming*, MIT Press, 1993. {16}

153. 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}

154. G. L. Steele Jr., *Common LISP: The Language*, Digital Press, 1984. {20}

155. G. L. Steele Jr., W. D. Hillis, *Connection Machine Lisp: Fine-Grained Parallel Symbolic Processing*, Tech. Rep. PL86–2, Thinking Machines Corporation, 1986. {18}

156. L. Sterling, E. Shapiro, *The Art of Prolog*, MIT Press, 1986 {3}

157. A. Taylor, *High Performance Prolog Implementation*, PhD thesis, Basser Department of Computer Science, Sydney University, 1991. {4}

158. R. D. Tennent, Quantification in Algol-like Languages, *Information Processing Letters*, 25:133–137, 1987. {20}

159. R. D. Tennent, *Semantics of Programming Languages*, Prentice-Hall International, 1991. {20}

160. Thinking Machines Corporation, *Connection Machine: ⋆Lisp Dictionary*, Cambridge, Massachusetts, 1991. {18}

161. E. Tick, *Parallel Logic Programming*, MIT Press, 1991. {5}

162. E. Tick, The Deevolution of Concurrent Logic Programming Languages, *Journal of Logic Programming*, 23(2):89–124, 1995. {16}

163. H. Touati, T. Hama, A Light-Weight Prolog Garbage Collector, *Proc. Intl. Conf. on Fifth Generation Computing Systems*, 1988. {20}

164. 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}

165. S.-Å. Tärnlund, Reform, unpublished manuscript, Computing Science Department, Uppsala University, 1991. {6, 16}

166. K. Ueda, *Guarded Horn clauses*, ICOT Tech. Rep. TR-103, Institute for New Generation Computing Technology, Tokyo, 1985. {16}

167. K. Ueda, *Guarded Horn clauses*, PhD thesis, Faculty of Engineering, Univ. of Tokyo, 1986. {16}

168. K. Ueda, Guarded Horn clauses, *Concurrent Prolog: Collected Papers*, vol. 1, MIT Press, 1987. {5}

169. K. Ueda, M. Morita, Moded Flat GHC and Its Message-Oriented Implementation Technique, *New Generation Computing*, 13(1):3–43, 1994. {5}

170. K. Ueda, K. Furukawa, Transformation Rules for GHC Programs, *Proc. Intl. Conf. on fifth Generation Computer Systems*, ICOT, 1988. {16}

171. S. Umeyama, K. Tamura, A Parallel Execution Model of Logic Programs, *Proc. Intl. Symp. Comp. Arch.*, 1983. {18}

172. D. M. Ungar, Generation Scavenging: A Non-Disruptive High Performance Storage Reclamation Algorithm, *ACM SIGPLAN Notices*, 19(5):157–167, 1984. {11, 12, 20}

173. 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}

174. P. L. Van Roy, 1983–1993: The Wonder Years of Sequential Prolog Implementation, *Journal of Logic Programming*, 19,20:385–441, 1994. {4}

175. P. L. Van Roy, A. Despain, The Benefits of Global Dataflow Analysis for an Optimizing Prolog Compiler, *Proc. NACLP'90*, 1990. {4}

176. A. Voronkov, Logic Programming with Bounded Quantifiers, *Logic Programming*, LNAI 592, Springer-Verlag, 1992. {9, 18}

177. A. Voronkov, *Logic Programming with Bounded Quantifiers*, Tech. Rep. ECRC-92-29, ECRC, Munich, 1992. {9, 18}

178. D. H. D. Warren, *Implementing Prolog—Compiling Predicate Logic Programs*, DAI Tech. Rep. 39-40, Edinburgh University, 1977. {3}

179. D. H. D. Warren, *An Abstract Prolog Instruction Set*, Report 309, SRI International, Menlo Park, California, USA, 1983. {3}

180. D. H. D. Warren, The SRI-model for Or-Parallel Execution of Prolog, *1987 IEEE Intl. Symp. Logic Programming*, IEEE Press, 1987. {22}

181. D. H. D. Warren, The Andorra Model, presented at Gigalips workshop, University of Manchester, 1988. {15}

182. 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}

183. D. S. Warren, Memoing for Logic Programs, *Communications of the ACM*, 35(3):93–111, 1992. {20}

184. R. A. Warren, M. V. Hermenegildo, Experiments with Prolog: An Overview, ACA/PP SRS Tech. Note 43, Computer Technology Corporation, Austin, 1987. {17}

185. P. Weemeeuw, B. Demoen, Garbage Collection in Aurora: An overview, *Proc. Intl. Workshop on Memory Management 92*, LNCS 637, Springer-Verlag, Berlin, 1992. {22}

186. P. R. Wilson, Uniprocessor Garbage Collection Techniques, *Proc. Intl. Workshop on Memory Management 92*, LNCS 637, Springer-Verlag, Berlin, 1992. {21}

187. M. J. Wise, *Prolog Multiprocessors*, Prentice-Hall, 1986. {15}

188. K. L. Wrench, *A Distributed AND-Or Parallel Prolog Network*, Tech. Rep. 212, University of Cambridge Computer Laboratory, 1990. {15}

189. R. Yang, *A Parallel Logic Programming Language and Its Implementation*, PhD thesis, Department of Electrical Engineering, Keio University, Yokohama, 1986. {15}

190. 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}

191. 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