UPPSALA THESES IN COMPUTING SCIENCE
26

Compilation Techniques for Prolog

Thomas Lindgren

Computing Science Department
Uppsala University

Thesis for the Degree of
Doctor of Philosophy

UPPSALA 1996

Compilation Techniques for Prolog

Thomas Lindgren

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 26
ISSN 0283-359X
ISBN 91-506-1181-X

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

Abstract

Lindgren, T. 1996: Compilation Techniques for Prolog, *Uppsala Theses in Computing Science 26*. 36pp. Uppsala. ISSN 0283-359X, ISBN 91-506-1181-X.

Current Prolog compilers are largely limited to optimizing a single predicate at a time. We propose two methods to express the global control of Prolog programs. The first method transforms a Prolog program into a continuation-passing style, where all operations have explicit success and failure continuations. The second method directly constructs a control flow graph from the Prolog program. This is done by first performing an analysis that determines the potential targets of success or failure for every predicate, then by using this information to build the control flow graph. We develop an optimization, factoring, that reduces the size of the analysis solution substantially.

Uninitialized output variables are an important feature of all high-performance implementations of logic programming language today. We show that a polyvariant output variable analysis is more precise and robust than the previously proposed monovariant analyses, while remaining fast and practical.

Garbage collection is a potential bottleneck in a high-performance Prolog implementation. We show that generational copying garbage collection can be adapted to Prolog, and that such a collector is considerably faster than currently used Prolog garbage collectors. A number of problems specific to Prolog and its implementation techniques are presented and solutions provided.

Finally, we propose Reform Prolog, a recursion-parallel dialect of Prolog. We show that recursion-parallelism can be efficiently exploited on today's shared memory multiprocessors by means of a restricted execution model, a streamlined runtime system and sophisticated compilation techniques. Our experiments indicate that Reform Prolog runs programs with low parallelization overheads (compared to standard sequential Prolog implementations), yet attains high speedups even on large multiprocessors.

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

ISSN 0283-359X
ISBN 91-506-1181-X
Printed by Graphic Systems AB, Stockholm 1996.

Acknowledgments

First of all, thanks to Håkan Millroth, my advisor. His guidance and mentoring have been invaluable.

Johan Bevemyr has been a close friend and colleague ever since our undergraduate days. Without his hard work on Reform Prolog and his knack for low-level implementation, things would have been much harder, not to mention less fun.

Per Mildner has been a great help and a conscientous critic of my later work. I have been fortunate in having ready access to his experience.

Thanks to my external examiner, Peter Van Roy, and to my readers, Jonas Barklund, Mats Carlsson, Seif Haridi and Sven-Olof Nyström. I appreciate your efforts.

The computing science department has provided a stimulating setting for research. In particular, Patric Hedlin, who helped with hacking Reform Prolog; Magnus Nordin, who provided perspectives on static analysis, computer graphics and animation and many other things; and Mattias Waldau, who deserves a big thanks for early inspiration.

I have been fortunate in discussing the ideas of this thesis with many of the premier people in logic programming. In particular, I would like to thank Saumya Debray, Ulrich Neumerkel, Paul Tarau and David H.D. Warren for discussions over the years.

John Lloyd and Tony Bowers were helpful and hospitable during a visit at Bristol; Malcolm Brown kindly helped me and Johan at the Edinburgh Parallel Computing Centre. Thanks to all of you.

Finally, many thanks to my friends and family for making these years enjoyable.

Prior publication

The papers in this thesis have been published previously. They have not been edited, except to adjust typography, remove typographical errors and update references.

A. T. Lindgren, A continuation-passing style for Prolog, in *Proc. Intl. Logic Programming Symposium 1994*, MIT Press, 1994.

B. T. Lindgren, Control flow analysis of Prolog, in *Proc. Intl. Logic Programming Symposium 1995*, MIT Press, 1995.

C. T. Lindgren, Polyvariant detection of uninitialized arguments of Prolog predicates, Journal of Logic Programming. Vol 28(3) Sept. 1996, pp. 217-229.

D. J. Bevemyr, T. Lindgren, Simple and efficient copying garbage collection for Prolog, in *Programming Language Implementation and Logic Programming 1994*, LNCS 844, Springer Verlag, 1994.

E. J. Bevemyr, T. Lindgren, H. Millroth, Exploiting recursion-parallelism in Prolog, in *PARLE-93*, eds. A. Bode, M. Reeve, G. Wolf, LNCS 694, Springer Verlag, 1993.

F. J. Bevemyr, T. Lindgren, H. Millroth, Reform Prolog: The language and its implementation, in *Logic Programming: Proceedings of the Tenth International Conference*, MIT Press, 1993.

G. T. Lindgren, J. Bevemyr, H. Millroth, Compiler Optimizations in Reform Prolog: Experiments on the KSR-1 Multiprocessor, in *Proc. EURO-PAR'95*, LNCS 966, Springer Verlag, 1995.

Summary

In this chapter, we briefly introduce the logic programming language Prolog, the compilation of Prolog and parallel execution of Prolog programs. We then summarize the rest of the chapters of this thesis, present the related work in the field, and summarize our contributions.

1.1 INTRODUCTION

Prolog

Prolog is a programming language based on a subset of predicate logic where programs consist of Horn clauses. Robinson [92] showed that resolution could be used to efficiently prove theorems for such theories; Colmerauer [30] and Kowalski [67] subsequently noted that such clauses could be viewed as *programs* in addition to logical theories.

For example, consider a predicate specifying list concatenation such that the third argument is the concatenation of the first two arguments. We can write this as two Horn clauses, as shown below.

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

Concatenate/3 is a simple inductive definition. If the first argument is the empty list, then the concatenation of the first two arguments is simply the second argument. If the first argument is a list [X | Xs], with head X and tail Xs, and the second argument is Ys, then the third argument is the concatenation if Zs is the concatenation of Xs and Ys, and the third argument is [X | Zs].

Consider the query:

```
?- concatenate([1,2,3],[4,5],A).
```

It is easy to prove that A = [1,2,3,4,5] is a solution.

However, logic programming differs from functional programming in that predicates can be used in several modes. Consider the following query.

```
?- concatenate(A,B,[1,2]).
```

This query has several solutions. The Prolog system will attempt to find one of them, and then produce more solutions on demand. Prolog employs an incomplete search and control strategy, which may fail to terminate even if solutions can be found in finite time. The use of such a strategy separates Prolog from pure theorem proving.

Many good books on the basics of Prolog and logic programming have been published, such as Sterling and Shapiro's textbook [114] on Prolog programming, or Lloyd's text on logic programming theory [74]. The novice reader is directed to those publications; we will in the rest of this thesis assume a working knowledge of the concepts and programming methods of Prolog and logic programming, such as resolution, unification, backtracking, cut, clause indexing, higher order predicates, and so on.

Compilation of sequential Prolog

A Prolog compiler maps resolution and unification into executable code. Since the distance between the two is considerable, the first implementations were interpreters. However, Warren [131] developed a compiled implementation of Prolog in the late 1970's. In 1983, he proposed his 'new engine', or Warren's abstract machine (WAM), as it became known [132].

The WAM inspired considerable research in Prolog implementation. The main goals of WAM were to optimize control by clause indexing and environment trimming, and to compile unifications into efficient low-level code. A previous design [131] also provided some of these concepts in a different setting. Several commercial and academic designs have been built on the WAM, including Quintus Prolog, SICStus Prolog and Eclipse.

Warren's design has been the starting point of two major directions of research. The first of these is to improve on Warren's engine by local optimizations which target each clause or predicate in isolation [19]. An example of this is when clauses fail early. In this case, backtracking can be optimized [20, 78]. A second example is that unifications can be compiled into very efficient code [124, 76, 77]. As a third example, clause indexing can be improved by sophisticated source-to-source transformation or code generation [34, 22].

The drawback of local techniques is that they cannot take advantage of the calling context. For example, it may be the case that a predicate is always

called in a single mode. Taking advantage of this information can significantly improve clause indexing and the compilation of primitive operations, by reducing both code size and execution time.

The second approach is to perform a global dataflow analysis of the program, and then annotate each predicate and/or goal with the facts holding when the predicate (goal) is reached. Among the dataflow facts detected by current compilers are the following.

- The mode of a variable.
- The possible bindings of a variable.
- Whether a variable needs to be dereferenced or trailed.
- Whether variables are aliased.

The analysis information can be used to improve clause indexing (e.g., by deleting type tests that are statically determined) or by improving unifications (e.g., if the involved variables are bound or free, or may be bound efficiently) [127, 118]. Compilation based on global analysis yields good or excellent speedups over systems lacking analysis [128, 119]. See also Van Roy's excellent survey of the implementation work for Prolog and logic programming [129].

Compilation of parallel Prolog

As multiprocessor computers become increasingly common, another means of reducing execution times presents itself: distributing the computation on several processors and running them in parallel.

Exploiting parallelism in Prolog is conceptually easy enough: perform multiple resolution steps in parallel. If a goal is resolved with several clauses at once, we say that the system extracts *or-parallelism*. If several goals are resolved simultaneously, the system extracts *and-parallelism*. When the resolved goals share free variables, the parallelism is classified as *dependent and-parallelism*. When goals do not share free variables, the parallelism is said to be *independent and-parallelism*.

Or-parallel systems have been successfully built for some years [2, 75] and have been applied to search problems. In recent years, or-parallel constraint solving has emerged as a potential application area [24, 122].

For and-parallel systems, the main problem is the management of parallel backtracking. An and-parallel system must handle process failure, which may require notifying consumers of the bindings the failing process has produced, as well as telling some other process to perform backtracking.

A secondary problem is to ensure that sequential and parallel execution produce identical answers (i.e., yield the same solutions and side-effects in the same order).

There are several solutions to these problems. The first, and perhaps simplest, is to define a new language, possibly better suited to concurrent execution. This approach was taken for languages such as Concurrent Prolog [103], Parlog [27] and GHC [125], and was termed stream and-parallelism: processes are goals, and communicate by means of shared variables that implement, e.g., streams of messages. Tick [121] gives a good overview of the state of the art in this field. An example of a recent implementation of such a language is the Monaco compiler [120].

A second approach, compatible with Prolog, is to extract independent and-parallelism [40, 56]. A failed process needs in this case only notify its siblings of the failure, which is substantially simpler than full dependence checking. A number of systems have been built to exploit independent and-parallelism [57, 52]. Subsequently, the notion of independence has been refined so that processes may share variables but still are independent as long as they do not affect each other's control [58].

The third approach (again used with Prolog) is to only bind variables when the binding process is deterministic. No dependence checking is then needed, because the producer of the binding cannot backtrack to produce another binding [135, 133, 84]. This is also called *determinism-driven and-parallelism*, and has been the basis of a number of research systems.

A fourth approach is to record the dependences between processes dynamically, so that backtracking can be done properly. This approach is taken by Conery and Kibler [31], and subsequently by Shen [104, 105, 106]. Dependence recording and dependence checking delegated entirely to runtime is too inefficient. However, Shen [104] has shown that by carefully selecting the dependences to record at compile-time, and otherwise run only independent goals in parallel, one can construct a system competitive with less general dependent and-parallel systems such as Andorra-I [136].

A fifth, and fairly recent, approach is to exploit *data-parallelism*, where a single program operates simultaneously on a (preferrably large) collection of data. The *single-program, multiple-data* (SPMD) paradigm, where each process is a parallel loop iteration, exploits data-parallelism and is arguably the most popular programming model for conventional multiprocessors. In logic programming, data-parallelism has been used in the contexts of or-parallelism [109, 122] and and-parallelism [8, 81, 9, 130].

The sixth and final approach combines the previous methods. A program may lack parallelism of a given form, which will yield poor performance on systems that extract only that form of parallelism. Some researchers

have set the goal to extract *maximal parallelism* from Prolog programs, by simultaneously exploiting multiple forms of parallelism [49, 52, 51, 50, 96].

1.2 SUMMARY

This section summarizes the papers in this thesis and discusses the scientific contributions, and my personal contributions.

In Prolog programs, the clause and goal selection rules are implicit. Paper A shows how to translate Prolog into a form where control is explicit, which simplifies the underlying implementation. Paper B shows how to derive a control flow graph from a Prolog program. Debray [36] has shown that the control flow graph is a useful tool for an optimizing compiler. Paper C proposes a robust static analysis to detect uninitialized arguments to predicates; this information can be used for important compiler optimizations. Paper D shows how to adapt generational copying garbage collection to Prolog, and demonstrates that this has advantages over previous mark-sweep algorithms. Papers E, F and G develop the design and implementation issues of Reform Prolog, a data-parallel dialect of Prolog.

Continuation-passing style for Prolog

A fundamental problem in executing Prolog is how to efficiently map Prolog's control (including recursion and backtracking) onto conventional hardware.

The traditional solution to this problem is to translate the Prolog code into abstract machine instructions that manage the control structures of Prolog, typically using a variant of WAM. The disadvantage with this approach is that the WAM is quite complex [132, 1] and thus difficult to implement, optimize and reason about.

While most Prolog implementations use a Prolog-to-WAM compiler, an alternative approach, *binarization*, has been proposed by Tarau [115, 116, 117] and Demoen and Mariën [42, 41]. The idea is to manage control by transforming general Prolog programs into a subset of Prolog that can be implemented more easily. When binarization is applied globally (i.e., for the entire program), the result is that every clause has a single goal. This removes the need for a goal selection rule in the process of resolving a clause. The advantage is that the underlying abstract machine is considerably simpler than WAM while being competitive in performance [116].

Binarization does not compile the clause selection (search) rule, however, but relies on WAM's scheme for backtracking. Paper A shows how to express failure control on the source level, just as binarization expresses success control. The idea is to rewrite the program so that every predicate consists

of a single, binary clause. Resolution of such a clause is trivial: replace the head of the clause with the body. All static control decisions of the program have thus been compiled away.

We develop a translation of Prolog into *binary continuation style* (BCS): each predicate takes two extra arguments, being the success and failure continuations of the predicate. First rewrite all predicates into *indexed single-clausal form*, where each predicate collapses its clauses into a single clause with explicit indexing operations (using, e.g., if-then-else and disjunctions). Each new predicate is then rewritten into a collection of BCS predicates and a number of metacall clauses that handle failure and success control when predicates return.

The generated programs have the following properties.

- Primitive operations P have explicit success and failure continuations.
- Nondeterministic choices are coded by (a) saving the current state, (b) building a continuation that restores the saved state, and (c) passing this as a failure continuation.
- All user-defined 'entry point' predicates are defined by a single, binary clause. They take two extra arguments: a success continuation and a failure continuation. These extra arguments can be viewed as 'labels with arguments' in the sense of the original continuation-passing style [113].
- The metacall clauses can be viewed as implementing two jump tables, or as code pointers on the implementation level.

The paper concludes with a brief discussion on some possible optimizations. First, the representation of environments and choicepoints is explicit, and thus open to compiler optimization. For example, a compiler may arrange so that environments share data, or so that environments and choicepoints share data. Second, the compiler can share code between predicates, in a spectrum of options ranging from WAM to binarization. Third, since state save/restore operations are explicit, the compiler can eliminate them or move them around.

Scientific contributions The paper extends binarization to encompass failure control. The new intermediate format, binary continuation style, can be used by an optimizing compiler.

Control flow analysis

Paper B discusses how to construct a control flow graph (CFG) for a Prolog program. The difficulty lies in properly accounting for tail recursion, backtracking and cuts.

The main problem is to determine where control passes when a predicate succeeds or fails. The solution chosen in the paper is to form a collection of set constraints. The analyzer first annotates the program with labels, indicating targets of success or failure. It then walks all the predicates of the program and formulates local constraints, e.g., on the form "if predicate p fails, it may backtrack into predicate q".

We solve these constraints by transforming them into a directed graph and reducing the graph with Tarjan's strongly connected components algorithm [100] into a directed acyclic graph. The reduced graph is then traversed to find the labels reachable by each node.

Given the control flow solution, we show how to construct a control flow graph (CFG) for the program. Our measurements indicate that success control is unproblematic, while failure control is less straightforward. However, in some smaller benchmarks, all backtracking could be statically resolved as jumps.

The size of the control flow solution (a set of labels per variable) may be impractical for large, nondeterministic programs. We show that the reduced graph has considerable structure which can be exploited to share the solution between variables, which we call *factoring*. For 11 out of 26 benchmark programs, the factored solution is half the size of the original solution. For a further seven programs, the original solution is 50% larger than the factored solution.

Scientific contributions The paper is the first to show how to construct a global control flow graph (encompassing the entire program) from a Prolog program, using a simple and efficient algorithm based on set constraints. It also shows how to reduce the size of the control flow solution by factoring the solution and sharing it between predicates.

Measurements on a set of standard benchmarks indicate that success control is typically simple, while failure control is more complex and sometimes very complex.

Polyvariant detection of uninitialized arguments

In practice, many Prolog predicates have arguments that are always unbound and unaliased. These *uninitialized variables* serve as output arguments, locations where the predicate returns information. This is good

news for a Prolog compiler. An uninitialized variable need not be trailed or dereferenced when it is bound, nor need we initialize its memory location beforehand. Optimizing uninitialized variables can lead to considerable execution time savings, and is used by all high-performance logic programming implementations today [128, 118, 110, 48].

While some systems require the user to declare output arguments, Prolog compilers traditionally derive such information by global analysis. A simple and effective approach is taken by Van Roy [128] and Getzinger [46]. However, their analysis has the drawback of being *monovariant*: all calls to a predicate in the entire program are used to summarize which arguments are uninitialized. The drawback is that a seldom-visited call site may be summarized with a frequently-visited call site and ruin valuable information at that site, which leads to less aggressive optimization. In essence, the monovariant analysis is brittle.

Paper C proposes the use of a *polyvariant* algorithm instead: multiple versions of a predicate may be generated, depending on the available information. The approach taken in the paper is to generate a new predicate version for every combination of initialized/uninitialized arguments.

Some questions are raised by this approach. First, the polyvariant algorithm may generate a number of predicate versions exponential in the number of arguments. Is the analysis practical? The paper shows that over a large number of benchmarks, ranging from tens to several thousands of lines of code, the number of versions per predicate is moderate: the number of predicates increases by approximately 20% over the original when program entrypoints are declared.

Second, does the polyvariant analysis yield better results than the monovariant analysis? Our measurements show that for fifteen programs, the monovariant and polyvariant algorithms yield the same results, while for sixteen programs, 2-26% of the predicate arguments are conditionally uninitialized (i.e., uninitialized at some but not all call sites). In three programs, two of which are 2000 lines of code or more, the monovariant analysis does not find *any* uninitialized arguments, while the polyvariant algorithm still detects 17-26% uninitialized arguments. The monovariant algorithm naturally enough never provided more precision than the polyvariant algorithm. Generally, the gap between the monovariant and polyvariant algorithm increased when we looked at only the large programs.

Finally, can the analysis be completed in reasonable time? In general, the polyvariant algorithm required less than a second to run on a 55-MHz workstation with SICStus native code. Even the largest program required only 4.5 seconds. Hence, such an analysis does not appear to be a bottleneck in a compiler.

Scientific contributions The paper shows that a polyvariant algorithm for uninitialized arguments is fast, practical (does not result in code explosion), and robust (always as precise as a monovariant algorithm and retains precision even in difficult cases).

Generational garbage collection

Paper D consider the problem of garbage collection for Prolog. In particular, the paper shows that copying and generational copying garbage collection can be implemented straightforwardly and efficiently on standard hardware.

The basic idea of the paper is to adapt an efficient collector, such as Cheney's algorithm [23]. The paper solves a number of problems that appear when adapting generational copying collection to Prolog.

Interior pointers are common in Prolog. An interior pointer is one that points directly to a field in a structure, without referring to the head of the structure. This implies that only parts of a structure may be live when a collection occurs, but also that there is no way to tell directly whether a pointer is interior or not.

We say that the *live data* at a given point is the data reachable by traversal starting from a set of root registers. A copying collector migrates the live data from an old area, called from-space, into a new area, called to-space.

Ideally, the algorithm should only migrate live data into to-space. If a structure has live subfields but is itself dead, then only the live subfields should be migrated. On the other hand, if the entire structure is live, then the entire structure must be copied as a unit. This is a conundrum for a straightforward copying algorithm, since it may encounter pointers to the subfields before discovering any pointers to the structure itself. Simply copying the subfield makes subsequent copying of the entire structure very difficult: the already copied fields must somehow be moved back inside the structure and pointers to the fields must be adjusted.

Paper D solves the problem of interior pointers by using a mark-copy algorithm. First, the live data in from-space are marked by traversing the data reachable from a set of roots. Then, the marked data are copied into to-space.

The next problem with copying collection is that the ordering of heap data is not retained after a collection. Consider the Prolog heap as a stack of segments delimited by choicepoints. On backtracking, a Prolog implementation can reclaim the topmost segment, since all data therein belong to a failed branch of the proof tree. This has led to the general use of mark-sweep algorithms that carefully preserve the segments. However, Cheney-style

copying [23] does not preserve the segment ordering of data. Instead, it traverses the live data in breadth-first order.

There are three problems with not preserving the ordering of data on the heap. First, memory can not always be reclaimed on backtracking. Second, Prolog implementations record conditional bindings on a trail stack, and can avoid recording a binding ("trailing the binding") in reasonably common situations. However, for our algorithm, bindings in the copied area must always be trailed (rather than occasionally). Third, variables may have been compared by special Prolog primitives, e.g., `@</2`. This is generally implemented by address comparison, and requires the relative ordering of variables to be maintained.

The proposed 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. Measurements show that the copying algorithm in practice recovers as much memory on backtracking as a conventional ("perfect") mark-sweep algorithm on a range of realistic benchmarks. Furthermore, the extra amount of trailing is shown to be negligible in practice: never more than one-quarter of a percent of the total number of trailings.

The proposed solution to the problem of retaining the order of compared variables is to migrate unbound variables to a dedicated stack at the time of comparison. That stack is then collected using a mark-sweep algorithm to retain the ordering between variables, while copying is used for the rest of the heap.

Finally, the paper shows how to extend the copying algorithm to generational collection. Generational garbage collection [71, 3] relies on the observation that newly created objects tend to be short-lived. Thus, garbage collection should concentrate on recently created data. The heap is split into two or more generations, and the most recent generation is collected most frequently. When the youngest generation fills up, a collection spanning more generations is done, and the survivors move to the oldest of these generations. The main difficulty is to record pointers from older to newer generations, which is needed to identify all live data. This is also called a *write-barrier*, since writing the old generation requires some checking and possibly recording the written pointer.

The crucial insight is that pointers from the old generation to the new generation (in a two-generation system) can be found by scanning the trail stack. By changing the trailing mechanism so that all bindings in the old generation are trailed, we get a write-barrier almost for free. The only extra cost is some unnecessary trailings in some situations. This cost is again negligible for the measured benchmarks.

Scientific contributions The paper shows how conventional garbage collection techniques can be adapted to Prolog by solving a number of problems specific to Prolog: interior pointers, heap ordering of data and write-barriers for generational garbage collection.

Measurements indicate that a number of potential problems with mark-copy and generational mark-copy are unproblematic in practice, that mark-copy beats mark-sweep, and that generational mark-copy beats mark-copy.

Author's contributions The construction of the garbage collection algorithms and experimental evaluation in the paper were done in close collaboration with Johan Bevemyr. Bevemyr wrote the actual collector code, collected the benchmarking data and integrated the three collectors with Reform Prolog.

Reform Prolog

Papers E and F introduced Reform Prolog, a language built on Millroth's principle of recursion parallelism [79]. Paper G evaluates compiler optimizations that reduce the number of synchronizations and locking operations needed during parallel execution.

Consider the list recursive predicate p.

$$\begin{array}{lcl} p([],\ldots) & \leftarrow & \Upsilon \\ p([X|Xs],\ldots) & \leftarrow & \Phi \wedge p(Xs,\ldots) \wedge \Psi \end{array}$$

Millroth [79, 81] showed that such predicates could be compiled into efficient loop code, using *Reform compilation*.

1. Compute the length of the input list as n and build the n instances of Φ and Ψ.
2. For i ranging from 1 to n, execute Φ_i.
3. Execute the remaining recursive call to p.
4. For i ranging from n to 1, execute Ψ_i.

Millroth also showed that nonlinear recursion could be compiled into loop code [80]; this is beyond the scope of our work.

The Reform Prolog project was started in September 1990 to study the parallelization of Prolog programs by means of Reform compilation. Originally,

the project members were Håkan Millroth, Margus Veanes and the author. Johan Bevemyr joined the group in late 1990. It was decided that dependent and-parallelism, where goals communicate by shared logic variables, was the most interesting venue.

An early prototype design, reminiscent of Shen's DDAS [104] in that it handled 'roll-backs' between processes (i.e., resetting the readers of a variable binding if the producer of the variable binding failed and removed the binding), ran on a distributed memory machine at Edinburgh's Parallel Computing Centre in September 1991. Our experiences with this system were mainly negative: tracking dependences between processes was quite expensive, and the combination of distributed memory (with explicit process communication) and pointer datastructures required considerable marshalling and unmarshalling when communication occurred. In 1992, Bevemyr and the author formulated a second design with the intent to optimize or remove, rather than tolerate, the difficult features of implementing a parallel Prolog system. Shen [106] had shown that automatic parallelization of sequential logic programs did not yield the huge speedups required to efficiently use massively parallel machines; thus, we instead put the responsibility of extracting parallelism on the programmer, and defined a simple language and execution model that could be implemented very efficiently.

Rather than trying to extract maximal parallelism from an arbitrary Prolog program, Reform Prolog parallelized a restricted and well-defined class of programs. Rather than accepting backtracking between processes, Reform Prolog enforced binding determinism and process determinism by a mixture of compile time and runtime means. Rather than using a sophisticated runtime system to control the parallel computation and exploit local opportunities for parallel execution, Reform Prolog only extracted parallelism from user-declared recursion-parallel predicates. Finally, rather than aiming for large speedups relative to a parallel system running on one processor, the Reform Prolog system was designed for high *absolute* speedups, with respect to a sequential system. This was done by extending a sequential WAM and confining overheads due to parallelism to process interactions (locking and suspension). It would then be straightforward to use available sequential compiler technology on the sequential portions of the code, should there be a need. Some encouraging initial experiments with the new system were run and reported at a JICSLP'92 workshop.

Execution model The Reform Prolog execution model gives programmers a model for parallel execution and parallel performance. It furthermore specifies how time-dependent operations, such as var/1 or indexing, are managed.

1. Binding a variable X that may be tested by time-dependent operations can only be done when the binding goal is leftmost in the resolvent. Testing the value of X must delay until X is bound or the loop iteration is leftmost.

2. Bindings to variables shared between loop iterations may never be conditional [84].

3. When a loop iteration is finished, it is always deterministic.

4. A (parallel) loop iteration can not start a parallel loop.

Further details, e.g., on handling predicates with side-effects, can be found in Refs. [13, 72].

By making the first restriction, we ensure that parallel and sequential execution yield the same answer. By making the second and third restrictions, we can guarantee that no costly 'rollback-on-failure' operations need to be supported. (See for instance Shen's DDAS for a design where such rollbacks are supported [104, 106].) The fourth restriction means the runtime system can be kept simple and efficient. All processors share a single parallel loop, which makes scheduling and leftmostness checking straightforward.

Compiler optimizations Reform Prolog puts the responsibility of keeping track of time-dependent and shared variables on the compiler. The Reform Prolog compiler must emit explicit locking operations when shared or time-dependent variables are bound, and preserves sequential semantics by emitting suspension instructions when time-dependent variables may be bound. This means the compiler can also *avoid* such instructions when it can determine that they are not needed. In fact, the compiler emits ordinary WAM code when the involved data are known to be local to the process.

In order to conservatively approximate which variables are shared or time-dependent, the compiler employs a static analysis. The static analysis determines modes, aliasing and linearity of variables. For parallel code, the analyzer also tracks whether variables *may be* robust or fragile: by tagging variables shared between loop iterations as robust when the loop is begun, and by tagging robust variables as fragile when they are tested. Finally, the analyzer also keeps track of whether loop iterations are deterministic or not, at every goal and at the end of an iteration.

Code generation using this information is described in greater detail in the author's licentiate thesis [72]. Briefly, the compiler uses knowledge of locality, modes and determinism to optimize operations. Many primitive operations permit synchronization and locking operations to be omitted when data are known to be instantiated or local.

Experimental results For a 24-processor Sequent Symmetry, we found parallelization overheads on the order of 2-12% as compared to a sequential Prolog system. Competing systems such as Andorra-I and NUA-Prolog show parallelization overheads of 35-40% and 50%, respectively (for 10 and 11 processors, respectively). For a 48-processor KSR-1, we found parallelization overheads of 0-17%, with larger benchmarks on the order of 2-6% .

Finally, two compiler optimizations were evaluated: removing suspensions and removing locking instructions. We found that optimizing suspensions was vital to the complex benchmarks, since they otherwise have to suspend almost immediately (although an interesting exception to this rule was found). Optimizing lockings was less successful. Even though the compiler reduced the number of executed lockings to 52% of the unoptimized number, there was no change in execution times. We attribute this to the infrequency of locking operations, and to the execution model of Reform Prolog.

Scientific contributions Papers E, F and G contribute a number of insights to the design and implementation of and-parallel logic programming systems.

- Data-parallelism is a rich source of parallel work even for logic programs, and recursion-parallelism is an appropriate tool to exploit such work.
- Reform Prolog confirms Naish's insight that binding determinism [84] is a crucial implementation concept for dependent and-parallel systems.
- Static analysis can accurately separate shared data from local data, which reduces the need for locking instructions (e.g., reduce the dynamic number of lockings to 52% of the unoptimized total).
- Static analysis can accurately identify the data accesses that sequentialize the parallel workers, and the compiler can avoid unnecessary sequentialization using this information.

Reform Prolog showed that a simple data-parallel Prolog implementation could attain significantly greater speedup and efficiency than previous comparable parallel logic programming systems.

Author's contributions Reform Prolog's execution model was designed in close collaboration with Johan Bevemyr and Håkan Millroth. The Reform Prolog compiler was written by me, while the execution engine was written by Bevemyr. The compiler optimizations were designed in collaboration with with Johan Bevemyr and implemented by me.

Reflections and later work Our results inspired a number of subsequent papers [59, 90, 89] which incorporate some of the Reform Prolog techniques into control-parallel systems, in particular the 'fast unfolding' of Reform compilation. Examples of such systems are &-Prolog [57] and ACE [52]. Results have generally been encouraging.

Recently, Bevemyr [14] and the author [73] have proposed methods to extend the 'flat' Reform Prolog system into efficiently executing arbitrarily nested recursion-parallel loops.

1.3 RELATED WORK

Paper A

There has been considerable interest in transformations of Prolog into Scheme and partial continuation passing styles and the relation of Prolog to intermediate representations, such as Warren's abstract machine [132]. We review earlier research in this section.

Warren's abstract machine It is interesting to compare our intermediate format with Warren's abstract machine, WAM [132].

- Our notation extends the scope of compilation (the 'chunks' or 'basic blocks') by considering actions taken on failure.
- Indexing can be expressed as a source-to-source transformation and also extended to utilize other mutually exclusive tests to restrict the number of candidate clauses.
- We can trade code size against possible unnecessary data allocation by intelligent generation of continuation clauses. Our formulation admits both the approach used in WAM, that of simple binarization, as well as a spectrum of trade offs in between.
- Continuation representation is explicit, thus allowing trimming of choice points as well as environments. Our notation represents choicepoints and environments as terms, which means a compiler can change the parameter passing conventions to suit particular situations, alter the representation of choicepoints and enviroments, and so on. In the WAM, choicepoints and environments have a fixed structure which is inaccessible to the compiler.
- Internal registers are saved separately from argument registers, which implements shallow backtracking by default rather than as a special optimization. We can also admit shallow backtracking in several levels by nested saving of internal registers.

On a more fundamental level, the WAM hides much of the control flow of the computation from the compiler writer. This simplifies compilation but excludes some optimizations. Our representation directly admits a translation into control flow graphs, since control flow is explicit.

Translations in Prolog Binarization, as proposed by Tarau [115] and Demoen and Mariën [42, 41] ignores the failure continuation that relates the different clauses of a predicate. The effects of this is that cuts are handled ad hoc, that control constructs other than conjunction must be handled by rewriting the program so that only conjunctions remain, and that an implicit control stack remains (the stack of choicepoints). Our binary continuation style avoids these problems and also allows us to directly describe indexing as a source-to-source transformation, which binarization delegates to the abstract machine.

Nilsson [87] shows how to derive the WAM choicepoint instructions by partial evaluation of a meta interpreter. Our work is distinct from the control aspects of the WAM and adds translations of other control constructs such as cut.

Brisset and Ridoux [17] propose a CPS for λProlog. As that language has λ-abstractions, their translation is similar to Scheme or Lisp translations discussed below, but does not explicitly manage substitutions. We have found that explicit operations that save and restore state can be useful, since they then can be moved and removed by program transformations. Since they generate higher-order terms in their translation, a subsequent pass of closure conversion [4] converts the term to a first-order representation. Our algorithm, in contrast, directly generates first-order terms as continuations.

Neumerkel [86, 85] has proposed Continuation Prolog, which allows compilers to manipulate continuations and remove some auxilliary output variables. The new program is then translated to binary or standard Prolog. The transformation is manual in nature, but could possibly be automated.

Optimization of Prolog using control flow graphs Debray [36] proposes several optimizations based on Prolog predicates translated into control flow graphs with success and failure edges. These edges do not correspond to the actual control of the program; for instance, if B_1 continues with B_2 on success, and B_2 on failure jumps to B_3, a success edge is added from B_1 to B_3 even though B_1 will never directly jump to B_3. This framework is then used to formulate several optimizations, such as moving tag tests and dereferencing operations out of loops. We believe that, mutatis mutandis, these optimizations could be carried out on our BCS programs as well.

Translations into Scheme and Lisp Several translations of Prolog into Lisp have been proposed. We take the work of Kahn and Carlsson as representa-

tive of this approach. Kahn and Carlsson [64, 18] propose the use of *upward failure* or *downward success* continuations. The former relies on using lazy streams to produce solutions, while the latter performs a procedure return on failure. We note that both methods to some extent bury the symmetry of success and failure continuations, and that neither method eliminates all control stacks. Kahn and Carlsson [65] then employ partial evaluation to produce quite good Lisp code for naive reverse with modes.

Research at Indiana University showed that Horn clauses could be translated into Scheme by fairly straightforward means [45, 66] and that some care was needed to extend Prolog with continuations [55]. In particular, a generalized trail was required to suspend and resume a proof tree branch. Such operations are beyond the scope of this paper; all our failure continuations obey a stack-like discipline.

Paper B

Debray [36] and Sehr [101, 102] use control flow graphs to optimize sequential programs and extract parallelism, respectively. Both authors considered only intraprocedural control flow. In our formulation, procedure calls disappear in an interprocedural sea of assignments, continuation creation and primitive operations.

Debray and Proebsting [39] have recently shown that control flow analyses can be transformed into parsing problems, and use LR(0) and LR(1)-items to perform the analysis. They consider languages with tail recursion, but lacking backtracking.

Shivers [107] and Jones, Gomard and Sestoft [63] proposed control flow analysis for the purpose of recovering the control flow graph of higher-order functional programs. We have studied control flow analysis for a language with less general and somewhat different control structures, and have shown that the solution can be found quickly and represented compactly.

Paper C

Beer [11] was the first to exploit uninitialized variables. This was done by a runtime approach, where registers and heap cells were tagged as uninitialized when created and modified during execution when unifications and similar operations occurred. He found that a large portion of the dynamically occurring variables actually were uninitialized. On a set of benchmarks, he found that dereferencing and trailing could be substantially reduced.

Van Roy [127] defined a static analysis that (among other things) derived uninitialized arguments. Our monovariant transformation in essence mimics

the 'uninitializedness' subset of Van Roy's analyzer and achieves approximately the same precision [128]. Our work thus shows that this part of Van Roy's analysis can be factored out of the rest of his analysis without losing precision, and that polyvariance further improves the results while avoiding code explosion. Van Roy also developed an algorithm that could return approximately 1/3 of the outputs in registers.

Getzinger improved on Van Roy's analysis and explored some alternatives, but remained within the monovariant framework [46, 47].

Taylor [118] subsequently incorporated uninitialized variables into his Parma compiler, and reported substantial performance gains. No indication of the number of uninitialized arguments derived was given.

The recently developed strongly-typed, strongly-moded logic programming language Mercury [110] restricts programs so that outputs are always uninitialized. A recent release performs a mode analysis similar to the one proposed in Paper C.

Bigot, Gudeman and Debray [15] have developed an alternative to Van Roy's algorithm, which they use to decide which output arguments should be returned in registers, and which should be returned in memory. It may be interesting to consider this for our benchmark set, and to possibly use multiple versions of a predicate for different call sites. The Bigot-Gudeman-Debray algorithm uses a single version per predicate.

Gudeman, De Bosschere, Debray and Kannan [16] have defined call forwarding as a way to hoist type tests out of loops or in general to elide type tests when the call site can statically decide tests in the callee. As shown in the nreverse example, our polyvariant transformation occasionally generates crude 'call forwarding' by breaking out calls with uninitialized arguments. This suggests that we could possibly share code between predicate versions. Van Roy's compiler [127] merges multiple calls that have produced the same intermediate code, which can be seen as the reverse of call forwarding.

We are only aware of two implementations of multiple specialization for Prolog programs: Sahlin's partial evaluator Mixtus [95] can generate multiple versions of a predicate, and Puebla and Hermenegildo [91] have recently used multiple specialization to improve the parallelization of &-Prolog.

We finally note that the proposed transformation can be seen as an abstract interpretation [33] followed by a program transformation based on the derived results.

Paper D

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. [5] as typical. This algorithm works in four steps and is based on the Deutsch-Schorr-Waite [99, 29] algorithm for marking and on Morris' algorithm [82, 29] for compaction (a more extensive summary is given in Paper D). Our copying collector uses a similar mark-phase, but copies data rather than compacting them.

Touati and Hama [123] 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 topmost 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 [10] and later by Older and Rummell [88].

We show in Paper D 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, 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.

Bekkers, Ridoux and Ungaro [12] observe that it is possible to reclaim garbage collected data on backtracking if copying collection starts at the oldest choice point (bottom-to-top). 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 [70, 94] is used to avoid duplication of variables inside structures. This may introduce new variable chains, as shown in Paper D. 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.

Subsequently, Demoen, Engels and Tarau [43] proposed and implemented an extension of the Bekkers-Ridoux-Ungaro bottom-up copying algorithm, combined with our mark-copy algorithm. They note that instant reclaiming of segments can be improved by moving data between segments. If some datum resides in segment i but is reachable only from a younger segment $i+n$, it can be migrated to segment $i+n$ and potentially be reclaimed earlier than would otherwise be possible. This relies on trimming choicepoints to delete dead fields. Furthermore, they show that top-down copying can lead to more major collections than bottom-up copying, since it can leave garbage in the old generation. We note that if the old generation is mostly filled with garbage, then a major collection will be quick, since it touches only live data. They find that the loss of instant reclaiming due to not retaining the heap ordering at a collection is "not prohibitive", which appears to confirm our results. Demoen et al do not directly compare their algorithm's efficiency with ours. However, their mark-copy algorithm appears to be approximately as efficient as ours. They do not present measurements for generational mark-copy.

Sahlin [93] has developed a method that makes the execution time of the Appleby et al. [5] 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. 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, in addition to the passes of 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.

Papers E,F and G

And-parallelism A parallel logic programming system exploiting dependent and-parallelism as well as or-parallelism was designed by Conery and Kibler [31], though with impractical overheads.

De Groot [40] and Hermenegildo [56] restricted parallel execution to independent and-parallelism, by performing runtime tests to determine whether a given conjunction is to be run in parallel or sequentially. When goals do not share variables, they can be executed in parallel. Backtracking can then be managed locally, possibly followed by killing all sibling goals when one goal in a parallel conjunction failed. Systems that rely solely on independent and-parallelism have been investigated by Hermenegildo [56, 57, 58] and others. In general, the performance is quite good when independent and-parallelism can be exploited. The overhead for parallel execution is mainly

related to process management and runtime tests for independence. An optimizing compiler can reduce the overhead of independence tests [83, 21].

Subsequent work combined independent and-parallelism with or-parallel [52] and dependent and-parallel [49] execution to extend the scope for parallel execution. The execution engines for the combination proposals are far more complex than a sequential Prolog engine, which is unfortunate if the goal is high absolute speedups with respect to a sequential system.

Yang [135] defined syntactic conditions for programs for parallel execution and found that deterministic computations were well-suited for parallel execution, since conflicting bindings meant that part of the computation failed rather than backtracked. Warren [133] formulated the related *Andorra principle*, where deterministic goals are executed in parallel and nondeterministic goals are suspended. Determinism is detected at runtime. Yang's static method was thus replaced with a dynamic method of detecting and exploiting parallelism. This approach is also called *determinism-driven parallel execution*. An implementation of the Andorra principle, Andorra-I, is described in several papers [96, 97, 98, 136].

Shen [106, 104] developed a method that allows *general* dependent and-parallelism, by maintaining enough information to restart processes that have consumed bindings that are then invalid, and by restricting access to shared variables. The resulting engine executes at about 25% the speed of SICStus Prolog, a standard Prolog implementation, and provides almost transparent parallelization of Prolog programs.

A different approach to binding conflicts was proposed by Naish [84] with PNU-Prolog. By requiring programs to be *binding determinate*, i.e., not undo any bindings during parallel execution, the binding conflict problem was once again reduced to global failure. Binding determinism was a major influence on the Reform Prolog execution model.

Clark and McCabe introduced explicitly concurrent language constructs in IC-Prolog [25] and further developed by Clark and Gregory in their work on the Relational Language [26]. From this school of thought sprang three *concurrent logic languages*, Concurrent Prolog [103], Parlog [27] and Guarded Horn Clauses (GHC) [125]. A restriction of the latter was chosen as the base language of the Japanese Fifth Generation Project. They all had in common that nondeterminism was strictly curtailed by suspending until only one alternative was possible or committing arbitrarily to one alternative when several were available. A consequence was that binding conflicts caused global failure rather than backtracking. (Certain proposed metaprimitives allow programmers to encapsulate a computation to recover from failure.) The Andorra Kernel Language [53] was proposed to join concurrent logic programming with Prolog's don't-know nondeterminism by splitting nondeterministic states into several deterministic computations.

A different approach is to provide the programmer with powerful linguistic constructs to specify large, regular parallel computations. The data-parallel paradigm, where the user specifies a single thread of control that operates simultaneously on many data objects, is suitable for this purpose.

Barklund and Millroth [8] constructed Nova Prolog, a data-parallel logic language, which later was generalized into *bounded quantifications* [9]. A bounded quantification expresses some action to be taken over a finite set of elements, which often allows data-parallel execution [7]. Voronkov [130] independently laid theoretical foundations for bounded quantifications. Using bounded quantifications allowed a concise data parallel approach to logic programming. Finally, Sehr [101, 102] has independently proposed extraction of data-parallelism in the context of Prolog; see also Section 1.3.

Loop parallelization Extracting parallelism from loops is the most popular method to parallelize imperative programs. There are essentially two methods: vectorization and concurrentization [134]. Vectorization translates program statements into vector instructions for supercomputers, and we will not consider it further. Concurrentization entails adding synchronization primitives to the loop body to ensure correct execution. Synchronization requirements are derived from *dependence analysis*, which determines what loads and stores can interfere. Recently, array dataflow analysis [44] has been developed to deal more precisely with this problem.

Automatic loop-level parallelization of imperative programs has met some difficulties. For instance, Banerjee et al [6] note:

> "Dependence analysis in the presence of pointers has been found to be a particularly difficult problem. ... Much work has been done on this problem, though in general it remains unsolved."

Reform Prolog also exploits loop level parallelism, but parallelizes programs with arbitrary pointer structures – the clean semantics and execution model of the language makes the required analyses simpler than in the imperative case. Singh and Hennessy [108] note that parallelization technology is too limited. In particular, loop parallelism by itself is insufficient to extract enough parallelism from the examined programs. We believe this is in part due to the inability to parallelize loops with procedure calls. Reform Prolog allows procedure calls in parallel procedures and checks statically that they conform to the execution model.

Harrison [54] developed a system that parallelizes recursive calls in Scheme. In particular, the implementation of recursion-parallelism is similar to ours, though presented in a more general context. Harrison performs side-effect and dependence analysis to restructure and parallelize Scheme programs.

The main differences of our system are (i) that Prolog relies on 'side-effects' in the form of single-assignment to a much higher degree than Scheme, which necessitates a different treatment from that described by Harrison, and (ii) that Harrison considers only **doall** loops and recurrences, which do not require the explicit insertion of synchronization instructions. The Reform compiler handles general **doacross** loops. Harrison performs a thorough job of restructuring the computation, which could be a useful future extension to our system.

Larus and Hilfinger describe an advanced parallelizing compiler for Lisp, Curare [69], that performs alias analysis prior to restructuring [68]. By computing the program dependence graph of the program, they can extract parallelism from the program. The alias analysis then serves as a data dependence analysis. Reform Prolog also computes an alias analysis, but extracts parallelism less freely. Furthermore, Reform Prolog reasons about locality of data, which Curare apparently does not. Locality information turns out to be very useful when processes are medium or coarse grained. Interestingly, Curare uses a *destination-passing* technique to improve parallelism, reminiscent of logic variables. The destination-passing style means a function takes an extra argument where the result of the function is stored.

Sehr [101, 102] has proposed a parallel Prolog system, built on the same principles as Reform Prolog. His system exploits or-parallelism and (essentially) recursion-parallelism, but is restricted to and-parallelizing only 'inner loop' predicates. In contrast, our compiler considers the entire program and can insert synchronization and locking instructions in predicates called from the parallel loop, not only the parallel loop predicate itself. Sehr's system was, as far as we know, never implemented.

Static analysis The dataflow analyzer of Reform Prolog was built by extending a framework proposed by Debray [37]. The abstract domain used is an extension of that of Debray and Warren [38, 35]. While their domain tracked may-aliases and modes, Reform Prolog tracks modes and list types (including some difference lists), may- and must-aliases, linearity, locality and determinism.

Aquarius Prolog [127] used a simple mode analysis to improve sequential code. While Aquarius Prolog's domain does not fully track aliases, and so is somewhat weaker than Reform Prolog's domain, we have found that on a number of programs, the list types of Reform Prolog are the sole practical difference in precision. Taylor [118, 119] proposed a somewhat stronger domain, based on depth-k tracking of term structure, modes, constant types and recursive list types, but did not publish any precision results. Getzinger [47] found that a domain similar to Taylor's provided the best cost-benefit ratio for sequential compilation, out of a large collection of domains.

The treatment of aliases in Reform Prolog is similar to that of Chang's SDDA [21], in that aliases are treated as equivalence classes of possibly or certainly aliased variables. Subsequently, more powerful tracking of aliases has been proposed, e.g., by Muthukumar and Hermenegildo [83], Sundararajan [111], Jacobs and Langen [60] and the PROP domain [32, 126]. These proposals have been used for accurate groundness information, however, while Reform Prolog uses aliases to keep track of freeness.

The linearity domain of Reform Prolog keeps track of whether a term contains repeated occurrences of variables [62]. The approach taken by Reform Prolog is less sophisticated than other proposed domains [62, 112, 28, 61], but appears to work well in practice. This is probably because many Prolog programs do not exhibit complex sharing.

1.4 CONCLUSION

The contributions of this thesis span a wide range of topics, centered around compiled execution of Prolog.

- We have developed a continuation-passing style that fully compiles the control of Prolog.
- We have developed a simple and efficient method for constructing the control flow graph of Prolog programs.
- We have developed a simple, efficient and robust technique to detect uninitialized arguments to Prolog predicates.
- We have adapted copying and generational copying garbage collection techniques to Prolog and demonstrated that these techniques are more efficient than traditional collectors.
- We have developed an efficient execution model for dependent and-parallel and recursion-parallel execution of Prolog programs.
- We have demonstrated that a sophisticated compiler can substantially reduce suspension and locking overhead in a dependent and-parallel Prolog implementation.

Bibliography

1. H. Aït-Kaci, *The WAM: A (Real) Tutorial*, MIT Press, 1991. {5}

2. K.A.M. Ali, R. Karlsson, The Muse or-parallel Prolog model and its performance, in *Proceedings of the North American Conference on Logic Programming*, MIT Press, 1990. {3}

3. A.W. Appel, Simple generational garbage collection and fast allocation, *Software—Practice and Experience* 19(2):171–183, 1989. {10}

4. A.W. Appel, *Compiling With Continuations*, Cambridge University Press, 1992. {16}

5. K. Appleby, M. Carlsson, S. Haridi, and D. Sahlin, Garbage Collection for Prolog Based on WAM, *Communications of the ACM*, 31(6):719–741, June 1988. {19, 20}

6. U. Banerjee, R. Eigenmann, A. Nicolau, D.A. Padua, Automatic program parallelization, Proceedings of the IEEE, vol. 81, no. 2, February 1993 {22}

7. J. Barklund, J. Bevemyr, Executing bounded quantifications on shared memory multiprocessors, in *Programming Language Implementation and Logic Programming 1993*, LNCS 714, Springer Verlag, 1993. {22}

8. J. Barklund, H. Millroth, *Nova Prolog*, UPMAIL Technical Report 52, Computing Science Department, Uppsala University, 1988. {4, 22}

9. J. Barklund, H. Millroth, Providing iteration and concurrency in logic programs through bounded quantifications, in *Proc. International Conference on Fifth Generation Systems*, Ohmsha, 1992. {4, 22}

10. J. Barklund, H. Millroth, Garbage cut for garbage collection of iterative Prolog programs, *3rd Symposium on Logic Programming*, Salt Lake City, September 1986, IEEE. {19}

11. J. Beer, The occur-check problem revisited, Journal of Logic Programming Vol. 5, pp. 243-261, North-Holland, 1988. {17}

12. Y. Bekkers, O. Ridoux and L. Ungaro, Dynamic Memory Management for Sequential Logic Programming Languages, *Proceedings of the International Workshop on Memory Management 92*, LNCS 637, Springer-Verlag, Berlin, 1992. {19}

13. J. Bevemyr, *A Recursion Parallel Prolog Engine*, Licentiate of Philosophy Thesis, Uppsala Theses in Computer Science 16/93, 1993. {13}

14. J. Bevemyr, A scheme for executing nested recursion parallelism, in JICSLP'96 Post-Conference Workshop on Implementation Techniques, September 1996. {15}

15. P. Bigot, D. Gudeman, S.K. Debray, Output value placement in moded logic programs, in *Logic Programming: Proceedings of the Eleventh International Conference*, MIT Press, 1994. {18}

16. K. De Bosschere, S.K. Debray, D. Gudeman, S. Kannan, Call forwarding: A simple interprocedural optimization technique for dynamically typed languages, in *Proc. Principles of Programming Languages*, ACM Press, 1994. {18}

17. P. Brisset, O. Ridoux, Continuations in λProlog, in *Proceedings of the Tenth International Conference on Logic Programming*, ed. D.S. Warren, MIT Press, 1993. {16}

18. M. Carlsson, On Implementing Prolog in Functional Programming, NGC 2 (1984), pp. 347-359. {17}

19. M. Carlsson, *Design and Implementation of an Or-Parallel Prolog Engine*, Ph.D. Thesis, SICS-RITA/02, 1990. {2}

20. M. Carlsson, On the efficiency of optimizing shallow backtracking in compiled Prolog, in *Proc. Sixth International Conference on Logic Programming*, MIT Press, 1989. {2}

21. J.-H. Chang, *High performance execution of Prolog programs based on a static dependency analysis*, Ph.D. Thesis, UCB/CSD 86/263, Univ. Calif. Berkeley, 1986. {21, 24}

22. T. Chen, I.V. Ramakrishnan, R. Ramesh, Multistage indexing algorithms, in *Proc. Joint International Conference & Symposium on Logic Programming'92*, MIT Press, 1992. {2}

23. C.J. Cheney, A nonrecursive list compacting algorithm, *Communications of the ACM*, 13(11):677–678, November 1970. {9, 10}

24. D.A. Clark, C.J. Rawlings, J. Shirazi, L-L. Li, K. Schuerman, M. Reeve, A. Véron, Solving large combinatorial problems in molecular biology using the ElipSys parallel constraint logic programming system, in Computer Journal 36(4). {3}

25. K.L. Clark, F. McCabe, The control facilities of IC-Prolog, in *Expert Systems in the Micro-Electronic World* (ed. D. Michie), Edinburgh University Press, 1979. {21}

26. K.L. Clark, S. Gregory, A relational language for parallel programming, in *Proceedings ACM Symposium on Functional Programming and Computer Architecture*, 1981. {21}

27. K.L. Clark, S. Gregory, PARLOG: A parallel logic programming language, report DOC 83/5, Department of Computing, Imperial College, London, 1983. {4, 21}

28. M. Codish, A. Mulkers, M. Bruynooghe, M. Garcia de la Banda, M. Hermenegildo, Improving abstract interpretations by combining domains, in *Proceedings of the 1993 Symposium on Partial Evaluation and Program Manipulation*, ACM Press, 1993. {24}

29. J. Cohen, Garbage Collection of Linked Data Structure, *Computing Surveys*, 13(3):341–367, September, 1981. {19}

30. 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. {1}

31. J.S. Conery, D.F. Kibler, Parallel interpretation of logic programs, in *Proceedings ACM Symposium on Functional Programming and Computer Architecture*, 1981. {4, 20}

32. A. Cortesi, G. Filé, W. Winsborough, Prop revisited: propositional formulas as abstract domain for groundness analysis, in *Proc. Sixth Annual IEEE Symposium on Logic in Computer Science*, IEEE Press, 1991. {24}

33. P. Cousot, R. Cousot, Abstract interpretation and application to logic programs, Journal of Logic Programming, 1992:13:103-179. {18}

34. S. Dawson, C.R. Ramakrishnan, I.V. Ramakrishnan, K. Sagonas, S. Skiena, T. Swift, D.S. Warren, Unification factoring for efficient execution of logic programs, in *Proc. Principles of Programming Languages*, MIT Press, 1996. {2}

35. S.K. Debray, Static inference of modes and data dependencies in logic programs, ACM Transactions on Programming Languages and Systems, Vol. 11, No. 3, July 1989, pp. 418-450. {23}

36. S.K.Debray, A simple code improvement scheme for Prolog, Journal of Logic Programming, 1992:13:57-88. {5, 16, 17}

37. S.K. Debray, Efficient dataflow analysis of logic programs, Journal of the ACM, Vol 39, No. 4, October 1992. {23}

38. S.K. Debray, D.S. Warren, Automatic mode inference for logic programs, Journal of Logic Programming, Vol. 5, No. 3, 1988. {23}

39. S.K. Debray, T. Proebsting, Inter-procedural control flow analysis of first order programs with tail call optimization, Draft, University of Arizona, May 1996. {17}

40. D. De Groot, Restricted AND-parallelism, in *Proceedings of the International Conference on Fifth Generation Systems*, North-Holland, Amsterdam, 1984. {4, 20}

41. B. Demoen, A. Mariën, Implementation of Prolog as Binary Definite Programs, Proceedings of the Second Russian Conference on Logic Programming, Springer Verlag. {5, 16}

42. B. Demoen, On the Transformation of a Prolog Program to a More Efficient Binary Program, Proceedings of the LOPSTR'92 workshop, Manchester, July 1992. {5, 16}

43. B. Demoen, G. Engels, P. Tarau, Segment order preserving copying garbage collection for WAM based Prolog, in ACM Symposium on Applied Computing, ACM Press, 1995. {20}

44. P. Feautrier, Dataflow analysis of array and scalar references, International Journal of Parallel Programming, Vol. 20, No. 1, February 1991, Plenum Press {22}

45. M. Felleisen, *Transliterating Prolog into Scheme*, Computer Science Department Technical Report 182, Indiana University, 1985. {17}

46. T. Getzinger, Abstract Interpretation for the Compile-Time Analysis of Logic Programs, Ph.D. Thesis, Technical Report ACAL-TR-93-09, University of South California, September 1993. {8, 18}

47. T. Getzinger, The Costs and Benefits of Abstract Interpretation-Driven Prolog Optimization, in *Proc. First International Static Analysis Symposium*, LNCS 864, Springer Verlag, 1994. {18, 23}

48. D. Gudeman, K. De Bosschere, S.K. Debray, `jc`: an efficient and portable sequential implementation of Janus, in *Logic Programming: Proceedings of the Joint International Conference and Symposium on Logic Programming*, MIT Press, 1992. {8}

49. G. Gupta, V. Santos Costa, R. Yang, M.V. Hermenegildo, IDIOM: Intergrating Dependent and-, Independent and- and Or-parallelism, in *Logic Programming: Proceedings of the 1991 International Symposium*, MIT Press, 1991. {5, 21}

50. G. Gupta, B. Jayaraman, Compiled and-or parallel execution of logic programs, in *Proc. North American Conference on Logic Programming '89*, MIT Press, 1989. {5}

51. G. Gupta, B. Jayaraman, Optimizing and-or parallel implementations, in *Proc. North American Conference on Logic Programming '90*, MIT Press, 1990. {5}

52. G. Gupta, M.V. Hermenegildo, ACE: And/Or-parallel copying-based execution of logic programs, in *Parallel Execution of Logic Programs*, LNCS 569, Springer Verlag, 1991. {4, 5, 15, 21}

53. S. Haridi, A logic programming language based on the Andorra model, New Generation Computing 7(1990), pp. 109-125, Springer Verlag, 1990. {21}

54. W.L. Harrison III, The interprocedural analysis and parallelization of Scheme programs, Lisp and Symbolic Computation, Vol. 2, no. 3/4, 1989 {22}

55. C. Haynes, Logic continuations, Journal of Logic Programming, 4(2):157–176, June 1987. {17}

56. M.V. Hermenegildo, An abstract machine for restricted AND-parallel execution of logic programs, in *Third International Conference on Logic Programming*, LNCS 225, Springer Verlag, 1986. {4, 20}

57. M.V. Hermenegildo, K.J. Greene, &-Prolog and its performance: exploiting independent and-parallelism, in *Proceedings of the Seventh International Conference on Logic Programming*, MIT Press, 1990. {4, 15, 20}

58. M.V. Hermenegildo, F. Rossi, Non-strict independent and-parallelism, in *Proceedings of the Seventh International Conference on Logic Programming*, MIT Press, 1990. {4, 20}

59. M. Hermenegildo, M. Carro, Relating data-parallelism and (And-)parallelism in logic programs, Computer Languages Journal, to appear. {15}

60. D. Jacobs, A. Langen, Accurate and efficient approximation of variable aliasing in logic programs, in *Proc. North American Conference on Logic Programming 1989*, MIT Press, 1989. {24}

61. G. Janssens, M. Bruynooghe, Deriving descriptions of possible values of program variables by means of abstract interpretation, Journal of Logic Programming 1992:13:205-258. {24}

62. N. Jones & H. Søndergaard, A semantics-based framework for the abstract interpretation of Prolog, report 86/14, University of Copenhagen, 1986. {24}

63. N. Jones, C. Gomard, P. Sestoft, *Partial Evaluation and Automatic Program Generation*, Prentice-Hall, 1993. {17}

64. K. Kahn, M. Carlsson, How To Implement Prolog on a Lisp Machine, in *Implementations of Prolog*, ed. J. Campbell, Ellis Horwood, 1984. {17}

65. K. Kahn, M. Carlsson, *The Compilation of Prolog Programs without the Use of a Prolog Compiler*, UPMAIL Technical Report 27, Uppsala University, 1984. {17}

66. E. Kohlbecker, *eu-Prolog*, Technical Report 155, Computer Science Department, Indiana University, 1984. {17}

67. R.A. Kowalski, Predicate logic as a computer language, in *Information Processing 74*, pp. 569-574, North-Holland, 1974. {1}

68. J.R. Larus, P.N. Hilfinger, Detecting conflicts between structure accesses, in *Proceedings of the SIGPLAN'88 Conference on Programming Language Design and Implementation*, ACM Press, 1988 {23}

69. J.R. Larus, P.N. Hilfinger, Restructuring Lisp programs for concurrent execution, in *Proceedings of the ACM/SIGPLAN PPEALS 1988 Parallel Programming: Experience with Applications, Languages and Systems*, ACM Press, 1988 {23}

70. S. Le Huitouze, A new datastructure for implementing extensions to Prolog, Proc. Programming Language Implementation and Logic Programming 1990, LNCS 456, Springer Verlag, 1990. {19}

71. 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. {10}

72. T. Lindgren, *The compilation and execution of recursion-parallel Prolog on shared-memory multiprocessors*, Licentiate of Philosophy Thesis, Uppsala Theses in Computer Science 18/93, November 1993. {13}

73. T. Lindgren, Compiling for nested recursion-parallelism, in JICSLP'96 post-conference workshop on implementation, September 1996. {15}

74. J. Lloyd, *Foundations of Logic Programming* (2nd ed.), Springer Verlag, {2}

75. E. Lusk, D.H.D. Warren, S. Haridi, P. Brand, R. Butler, A. Calderwood, M. Carlsson, A. Ciepielewski, T. Disz, B. Hausman, R. Olson, R. Overbeek, R. Stevens, P. Szeredi, The Aurora or-parallel system, New Generation Computing, vol 7(2-3), 1990. {3}

76. A. Mariën, B. Demoen, A new scheme for unification in WAM, in *International Logic Programming Symposium 1991*, MIT Press, 1991. {2}

77. M. Meier, Compilation of compound terms in Prolog, technical report ECRC-95-12, ECRC, July 1990. {2}

78. M. Meier, Shallow backtracking in Prolog programs, technical report ECRC-95-11, ECRC, February 1987. {2}

79. H. Millroth, *Reforming the compilation of logic programs*, Ph.D. Thesis, Uppsala Theses in Computer Science 10, 1991. {11}

80. H. Millroth, Reform compilation for non-linear recursion, *Proceedings of the International Conference on Logic Programming and Automated Reasoning*, LNCS 624, Springer Verlag, 1992. {11}

81. H. Millroth, SLDR-resolution: parallelizing structural recursion in logic programs, Journal of Logic Programming, Vol 25(2), Nov. 1995, pp. 93-117. {4, 11}

82. F. Morris, A Time- and Space- Efficient Compaction Algorithm, Communications of the ACM, 12(9):662–665, August 1978. {19}

83. K. Muthukumar, M.V. Hermenegildo, Compile-time derivation of variable dependency using abstract interpretation, Journal of Logic Programming, 1992:13:315-347. {21, 24}

84. L. Naish, Parallelizing NU-Prolog, *Logic Programming: Proceedings of the Fifth International Conference and Symposium*, MIT Press, 1988. {4, 13, 14, 21}

85. U. Neumerkel, A transformation based on the equality between terms, in *Logic Programming Synthesis and Transformation, LOPSTR'93*, Springer Verlag, 1993. {16}

86. U. Neumerkel, Continuation Prolog: A new intermediary language for WAM and BinWAM code generation, in Post-ILPS'95 Workshop on Implementation of Logic Programming Languages. {16}

87. U. Nilsson, Towards a methodology for the design of abstract machines for logic programming languages, Journal of Logic Programming, 1993:16:163-189. {16}

88. W.J. Older and J.A. Rummell, An Incremental Garbage Collector for WAM-Based Prolog, *Proceedings of the Joint International Conference and Symposium on Logic Programming*, MIT Press, 1992. {19}

89. E. Pontelli, G. Gupta, Data Parallel Logic Programming in &ACE, Proc. of the IEEE International Symposium on Parallel and Distributed Processing, IEEE Press, 1995. {15}

90. E. Pontelli, G. Gupta, Determinacy Driven Optimization of And-Parallel Prolog Implementations, *Int. Conf. Logic Programming*, MIT Press, 1995. {15}

91. G. Puebla, M. Hermenegildo, Implementation of multiple specialization in logic programs, in *Proc. Partial Evaluation and Program Manipulation*, ACM Press, 1995. {18}

92. J. A. Robinson, A machine-oriented logic based on the resolution principle, Journal of the ACM, 12(1):23-41, 1965. {1}

93. D. Sahlin, Making garbage collection independent of the amount of garbage, Research Report R87008, Swedish Institute of Computer Science, 1987. {20}

94. D. Sahlin, M. Carlsson, Variable shunting for the WAM, SICS research report R91:07, SICS, 1991. {19}

95. D. Sahlin, The Mixtus approach to automatic partial evaluation of full Prolog, in *Proc. North American Conference on Logic Programming'90*, MIT Press, 1990. {18}

96. V. Santos Costa, D.H.D. Warren, R. Yang, Andorra-I: A parallel Prolog system that transparently exploits both and- and or-parallelism, in *Third ACM SIGPLAN Symposium on Principles & Practices of Parallel Programming*, ACM Press, 1991. {5, 21}

97. V. Santos Costa, D.H.D. Warren, R. Yang, The Andorra-I preprocessor: supporting full Prolog on the basic Andorra model, in *Logic Programming: Proceedings of the Eighth International Conference*, MIT Press, 1991. {21}

98. V. Santos Costa, D.H.D. Warren, R. Yang, The Andorra-I engine: a parallel implementation of the basic Andorra model, in *Logic Programming: Proceedings of the Eighth International Conference*, MIT Press, 1991. {21}

99. H. Schorr and W.M. Waite, An Efficient Machine-Independent Procedure for Garbage Collection in Various List Structures, Communications of the ACM, 10(8):501–506, August 1967. {19}

100. R. Sedgewick, *Algorithms (2nd edition)*, Addison-Wesley, 1989. {7}

101. D.C. Sehr, L.V. Kale, D.A. Padua, Loop transformations for Prolog programs, LNCS 768, Springer Verlag, 1993. {17, 22, 23}

102. D.C. Sehr, *Automatic Parallelization of Prolog Programs*, Ph.D. Thesis, Univ. of Illinois at Urbana Champaign, 1992. {17, 22, 23}

103. E.Y. Shapiro, *A subset of Concurrent Prolog and its interpreter*, ICOT Technical Report TR-003, Institute for New Generation Computing Technology, Tokyo, 1983. {4, 21}

104. K. Shen, Exploiting dependent and-parallelism in Prolog: the dynamic dependent and-parallel scheme (DDAS), in *Proceedings of the Joint International Symposium on Logic Programming*, MIT Press, 1992. {4, 12, 13, 21}

105. K. Shen, Initial results of the parallel implementation of DASWAM, in *Logic Programming: Proceedings from the 1996 Joint International Conference and Symposium*, MIT Press, 1996 {4}

106. K. Shen, *Studies of And-Or Parallelism*, Ph.D. Thesis, Cambridge University, revised June 1992. {4, 12, 13, 21}

107. O. Shivers, Control flow analysis in Scheme, in *Proc. SIGPLAN'88 Conference on Programming Language Design and Implementation*, ACM Press, 1988. {17}

108. J.P. Singh, J.L. Hennessy, An empirical investigation of the effectiveness and limitations of automatic parallelization, in *Proceedings of the International Symposium on Shared Memory Multiprocessing*, Tokyo, April 1991 {22}

109. D.A. Smith, MultiLog: data or-parallel logic programming, in *Logic Programming: Proceedings of the Tenth International Conference*, MIT Press, 1993. {4}

110. Z. Somogyi, F. Henderson, T. Conway, The execution algorithm of Mercury: an efficient purely declarative logic programming language. Revised version of paper appearing in *ILPS'94 Post-Conference Workshop on Implementation Techniques for Logic Programming Languages*. {8, 18}

111. R. Sundararajan, An abstract interpretation scheme for groundness, freeness and sharing analysis of logic programs, Technical report CIS-TR-91-06. Dept. of Computer and Information Science, University of Oregon, 1991. {24}

112. H. Søndergaard, An application of abstract interpretation of logic programs: Occur check reduction, in *ESOP'86 Proceedings European Symposium on Programming*, LNCS 213, Springer Verlag, 1986. {24}

113. G.L. Steele, *Rabbit: A Compiler for Scheme*, MIT AI Memo 474, Massachusetts Institute of Technology, 1978. {6}

114. L. Sterling, E. Shapiro, *The Art of Prolog*, MIT Press, 1986 {2}

115. P. Tarau, M. Boyer, Elementary logic programs, in *Programming Language Implementation and Logic Programming 1990*, LNCS 456, Springer Verlag, 1990. {5, 16}

116. P. Tarau, Low-level issues in implementing a high-performance continuation passing Prolog engine, in *Programming Language Implementation and Logic Programming '92*, LNCS 631, Springer Verlag, 1992. {5}

117. P. Tarau, WAM-optimizations in BinProlog: towards a realistic Continuation Passing Prolog Engine, Technical Report 92-3, Université de Moncton, 1992. {5}

118. A. Taylor, *High Performance Prolog Implementation*, Ph.D. Thesis, Basser Department of Computer Science, Sydney University, 1991. {3, 8, 18, 23}

119. A. Taylor, Parma – bridging the gap between imperative and logic programming, Journal of Logic Programming, Special Issue on High-Performance Implementations. To appear. {3, 23}

120. E. Tick, C. Banerjee, Performance evaluation of Monaco compiler and runtime kernel, in *Logic Programming: Proceedings of the Tenth International Conference*, MIT Press, 1993. {4}

121. E. Tick, The deevolution of concurrent logic programming languages, Journal of Logic Programming 23(2), 89-124, 1995. {4}

122. B.-M. Tong, H.-F. Leung, Concurrent constraint logic programming on massively parallel SIMD computers, in *Proc. International Symposium on Logic Programming 1993*, MIT Press, 1993. {3, 4}

123. H. Touati, T. Hama, A light-weight prolog garbage collector, in *Proceedings of the International Conference on Fifth Generation Computing Systems*, Ohmsha, 1988. {19}

124. A.K. Turk, Compiler optimizations for the WAM, in *Third International Conference on Logic Programming*, LNCS 225, Springer Verlag, 1986. {2}

125. K. Ueda, *Guarded Horn clauses*, ICOT Technical Report TR-103, Institute for New Generation Computing Technology, Tokyo, 1985. {4, 21}

126. P. Van Hentenryck, A. Cortesi, B. Le Charlier, Evaluation of the domain `Prop`, Journal of Logic Programming Vol. 23, pp. 237-278, Elsevier, 1995. {24}

127. P.L. Van Roy, *Can Logic Programming Execute as Fast as Imperative Programming?*, Ph.D. Thesis, UCB/CSD 90/600, Computer Science Division (EECS), University of California, Berkeley, 1990. {3, 17, 18, 23}

128. P. Van Roy, A. Despain, The benefits of global dataflow analysis for an optimizing Prolog compiler, in *Proc. North American Conference on Logic Programming'90*, MIT Press, 1990. {3, 8, 18}

129. Peter Van Roy, 1983-1993: The wonder years of sequential Prolog implementation, Journal of Logic Programming, 1994:19,20:385-441 {3}

130. A. Voronkov, Logic programming with bounded quantifiers, in *Logic Programming*, LNAI 592, Springer Verlag, 1992. {4, 22}

131. D.H.D. Warren, *Implementing Prolog – compiling predicate logic*, DAI Technical Reports 39-40, Edinburgh University {2}

132. D.H.D. Warren, *An abstract Prolog instruction set*, Report 309, SRI International, 1983. {2, 5, 15}

133. D.H.D. Warren, The Andorra model, presented at Gigalips workshop, University of Manchester, 1988. {4, 21}

134. M. Wolfe, *Optimizing Supercompilers for Supercomputers*, MIT Press, 1989 {22}

135. R. Yang, *A Parallel Logic Programming Language and Its Implementation*, Ph.D. Thesis, Department of Electrical Engineering, Keio University, Yokohama, 1986. {4, 21}

136. R. Yang, T. Beaumont, I. Dutra, V. Santos Costa, D.H.D. Warren, Performance of the compiler-based Andorra-I system, in *Logic Programming: Proceedings of the Tenth International Conference on Logic Programming*, MIT Press, 1993. {4, 21}

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

Uppsala Theses in Computer Science 26
ISSN 0283-359X
ISBN 91-506-1181-X