36

digital

PARIS RESEARCH LABORATORY

1983–1993: The Wonder Years of Sequential Prolog Implementation

December 1993

Peter Van Roy

36

1983–1993: The Wonder Years of Sequential Prolog Implementation

Peter Van Roy

December 1993

Publication Notes

This report is an edited version of an article to appear in the *Journal of Logic Programming*, tenth anniversary issue, 1994, Elsevier North-Holland.

Contact address of author:

Peter Van Roy
vanroy@prl.dec.com

Digital Equipment Corporation
Paris Research Laboratory
85 Avenue Victor Hugo
92500 Rueil-Malmaison, France

Abstract

This report surveys the major developments in sequential Prolog implementation during the period 1983–1993. In this decade, implementation technology has matured to such a degree that Prolog has left the university and become useful in industry. The survey is divided into four parts. The first part gives an overview of the important technical developments starting with the Warren Abstract Machine (WAM). The second part presents the history and the contributions of the major software and hardware systems. The third part charts the evolution of Prolog performance since Warren's DEC-10 compiler. The fourth part extrapolates current trends regarding the evolution of sequential logic languages, their implementation, and their role in the marketplace.

Résumé

Ce rapport passe en revue les developpements majeurs d'implantation de Prolog séquentiel pendant les années 1983–1993. Dans cette période, la technologie d'implantation a mûri considérablement. Prolog n'est plus seulement utilisé dans les universités mais est devenu utile pour l'industrie. La revue est divisée en quatre parties. La première donne un résumé des développements techniques importants à partir de la machine abstraite de Warren, la WAM ("Warren Abstract Machine"). La seconde partie présente l'histoire et les contributions des logiciels et matériels. La troisième partie montre l'évolution des performances des systèmes Prolog depuis le compilateur DEC-10 de Warren. La quatrième partie extrapole, à partir des tendances actuelles, l'évolution des langages logiques séquentiels, leur implantation, et leur impact économique.

Keywords

Implementation, Prolog, logic programming, WAM, Warren Abstract Machine, abstract interpretation, compiler, survey, history.

Acknowledgements

The author thanks the many developers and pioneers in the logic programming community. In particular, the following friends and colleagues helped tremendously by recollecting past events and providing critical comments: Abder Aggoun for ECRC and CHIP, Dave Bowen for Quintus Prolog, Mats Carlsson for SICS, SICStus Prolog, and its native code timing measurements, Koen De Bosschere, Saumya Debray for SB-Prolog and QD-Janus, Bart Demoen and André Mariën for Leuven and BIM Prolog, Marc Gillet for IBM Prolog and its timing measurements, Manuel Hermenegildo for MA^3 and PLAI, Bruce Holmer, Tim Lindholm for Quintus Prolog, Peter Ludemann, Micha Meier for ECRC, SEPIA, the KCM, and ECLiPSe, Richard Meyer for putting up with this paper *black hole*, Lee Naish and Jeff Schultz for MU-Prolog and NU-Prolog and their timing measurements, Hiroshi Nakashima, Katsuto Nakajima, Takashi Chikayama, and Kouichi Kumon for ICOT, the PSI machines and their timing measurements, Ulrich Neumerkel for the VAM and BinProlog, Jacques Noyé for ECRC and the KCM, Fernando Pereira for Edinburgh, DEC-10 Prolog, C-Prolog, and Quintus Prolog, Christian Pichler for IF/Prolog and SNI-Prolog, Andreas Podelski, Olivier Ridoux for MALI and λProlog, Konstantinos Sagonas for SB-Prolog, XSB and its timing measurements, Bart Sano for the VLSI-BAM, Kazuo Seo for Pegasus and the Pegasus-II timing measurements, Zoltan Somogyi for NU-Prolog and databases, Vason Srini for the VLSI-PLM, Péter Szeredi for MProlog and its timing measurements, Paul Tarau for BinProlog, Andrew Taylor, Evan Tick for ICOT, David H.D. Warren for Edinburgh, DEC-10 Prolog, and Quintus Prolog, David Scott Warren for Stony Brook, SB-Prolog, the SLG-WAM, and XSB, S. Bharadwaj Yadavalli, and finally, the referees for a host of good suggestions. I also thank Hervé Touati and Chris Weikart for their excellent job of proofreading. Finally, a special thank you for TEXnician *par excellence* Hassan Aït-Kaci.

Contents

De Prolog van Tachtig was zonder twijfel prachtig,
maar de Prolog van Thans maakt ook geen kwade kans.
– Dr. D. von Tischtiegel, *Ongerijmde Rijmen*.

1 Introduction

This report is a personal view of the progress made in sequential Prolog implementation from 1983 to 1993, supplemented with learning of the wise [10]. 1983 was a serendipitous year in two ways, one important and one personal. In this year David H. D. Warren published his seminal technical report [163] on the New Prolog Engine, which was later christened the WAM (for Warren Abstract Machine).[1] This year also marks the beginning of my research career in logic programming.

The title reflects my view that the period 1983–1993 represents the "coming of age" of sequential Prolog implementation. In 1983, most Prolog programmers (except for a lucky few at Edinburgh and elsewhere) were still using interpreters. In 1993 there are many high quality compilers, and the fastest of these are approaching or exceeding the speed of imperative languages. Prolog has found a stable niche in the marketplace. Commercial systems are of high quality with a full set of desirable features and enough large industrial applications exist to prove the usefulness of the language [102, 103].

1.1 The Influence of the WAM

The development of the WAM in 1983 marked the beginning of a veritable "gold rush" for Prolog developers, all eager for that magical moment when their very own system would be up and running.

David Warren presented the WAM in a memorable talk at U.C. Berkeley in October 1983. This talk was full of mystery, and I remember being amazed at how *append/3* was compiled into WAM instructions. The sense of mystery was enhanced by the strange names of the instructions: put, get, unify, variable, value, execute, proceed, try, retry, and trust.

The WAM is simple on the outside (a small, clean instruction set) and complex on the inside (the instructions do complex things). This simultaneously helped and hindered implementation technology. Because the WAM is complex on the inside, for a long time many people used it "as is" and were content with its level of performance. Because the WAM is simple on the outside, it was a perfect environment for extensions. After a few years, people were extending the WAM left and right (see Section 2.3). Papers on yet another WAM extension for a new logic language were (and are) very common.

The quickest way to get an implementation of a new logic language is to write an interpreter in

[1] The name *WAM* is due to the logic programming group at Argonne National Laboratory.

Prolog. In the past, the quickest way to get an *efficient* implementation was usually to extend the WAM. Nowadays, it is often better to compile the language into an existing implementation. For example, the QD-Janus system [39] is a sequential implementation of Janus (a flat committed-choice language) on top of SICStus Prolog (see Section 3.1.9). Performance is reasonable partly because SICStus provides efficient support for coroutining.

If the language is sufficiently different from Prolog, then it is better to design a new abstract machine. For example, the λProlog language [100] was implemented with MALI [20]. λProlog generalizes Prolog with predicate and function variables and typed λ-terms, while keeping the familiar operational and least fixpoint semantics. MALI is a general-purpose memory management library that has been optimized for logic programming systems.

1.2 Organization of the Survey

The survey is divided into four parts. The first part (Section 2) gives an overview from the viewpoint of implementation technology. The second part (Section 3) gives an overview from the viewpoint of the systems (both software and hardware) that were responsible for particular developments. The vantage points of the two parts are complementary, and there is some overlap in the developments that are discussed. The third part (Section 4) summarizes the evolution of Prolog performance from the perspective of the Warren benchmarks. The fourth part (Section 5) extrapolates current implementation trends into the future. Finally, Section 6 recapitulates the main developments and concludes the survey.

A large number of Prolog systems have been developed. The subset included in this survey covers systems that are popular (*e.g.*, SICStus Prolog), are good examples of a particular class of systems (*e.g.*, CHIP for constraint languages), or are especially innovative (*e.g.*, Parma). They all have implementations on Unix workstations. I have done my best to contact everyone who has made a significant contribution. There are Prologs that exist only on other platforms, *e.g.*, on PCs (Arity, LPA, Delphia) and on Lisp machines (LMI, Symbolics). There is relatively little publicly available information about these systems, and therefore I do not cover them in this report.

2 The Technological View

This section gives an overview of Prolog implementation technology. Section 2.1 gives a brief history of the pre-WAM days (before 1983) and presents the main principle of Prolog compilation. Section 2.2 presents and justifies the WAM as Warren originally defined it. Section 2.3 explores a few of the myriad systems it has engendered. Section 2.4 highlights recent developments that break through its performance barrier. Section 2.5 presents some promising execution models different from the WAM.

Prolog systems can be divided into two categories: *structure-sharing* or *structure-copying*. The idea of structure sharing is due to Boyer and Moore [19]. Structure copying was first described by Bruynooghe [21, 22]. The distinction is based on how compound terms are represented. In a structure-sharing representation, all compound terms are represented as a pair of pointers (called a *molecule*): one pointer to an array containing the values of the term's variables, and another pointer to a representation of the term's nonvariable part (the *skeleton*). In a structure-copying representation, all compound terms are represented as record structures with one word identifying the main functor followed by an array of words giving its arguments. It is faster to create terms in a structure-sharing representation. It is faster to unify terms in a structure-copying representation. Memory usage of both techniques is similar in practice. Early systems were mostly structure-sharing. Modern systems are mostly structure-copying. The latter includes WAM-based systems and all systems discussed in this survey, except when explicitly stated otherwise.

2.1 Before the Golden Age

The insight that deduction could be used as computation was developed in the 1960's through the work of Cordell Green and others. Attempts to make this insight practical failed until the conception of the Prolog language by Alain Colmerauer and Robert Kowalski in the early 1970's. It is hard to imagine the leap of faith this required back then: to consider a logical description of a problem as a program that could be executed efficiently. The early history is presented in [32], and interested readers should look there for more detail.

The work on Prolog was preceded by the Absys system. Absys (from *Aberdeen System*) was designed and implemented at the University of Aberdeen in 1967. This system was an implementation of pure Prolog [46]. For reasons that are unclear but that are probably cultural, Absys did not become widespread.

Several systems were developed by Colmerauer's group. The first system was an interpreter written in Algol-W by Philippe Roussel in 1972. This interpreter served to give users enough programming experience so that a refined second system could be built. The second system was a structure-sharing interpreter written in Fortran in 1973 by Gérard Battani, Henri Meloni, and René Bazzoli, under the supervision of Roussel and Colmerauer. This system's operational semantics and its built-ins are essentially the same as in modern Prolog systems, except for the *setof/3* and *bagof/3* built-ins which were introduced by David Warren in 1980 [162]. The system had reasonable performance and was very influential in convincing people that programming in logic was a viable idea.

In particular, David Warren from the University of Edinburgh was convinced. He wrote the Warplan program during his two month stay in Marseilles in 1974 [30]. Warplan is a general problem solver that searches for a plan (a list of actions) that transforms an initial state to a goal state.

2.1.1 The First Compiler: DEC-10 Prolog

Back in Edinburgh and thinking about a dissertation topic, Warren was intrigued by the idea of building a compiler for Prolog. An added push for this idea was the fact that the parser for the interpreter was written in Prolog itself, and hence was very slow. It took about a second to parse each clause and users were beginning to complain.

By 1977 Warren had developed DEC-10 Prolog, the first Prolog compiler [159]. This landmark system was built with the help of Fernando Pereira and Luis Pereira.[2] It is structure-sharing and supports mode declarations. It was competitive in performance to Lisp systems of the day and was for many years the highest performance Prolog system. Its syntax and semantics became the de facto standard, the "Edinburgh standard". The 1980 version of this system had a heap garbage collector and last call optimization (see Section 2.2.4) [160]. It was the first system to have either. An attempt to commercialize this system failed because of the demise of the DEC-10/20 machines and because of bureaucratic problems with the British government, which controlled the rights of all software developed with public funds.

2.1.2 The Simplification Principle

The main principle in compiling Prolog is to simplify each occurrence of one of its basic operations (namely, unification and backtracking). This principle underlies every Prolog compiler. Compiling Prolog is feasible because this simplification is so often possible. For example, unification is often used purely as a parameter passing mechanism. Most such cases are easily detected and compiled into efficient code.

It is remarkable that the simplification principle has continued to hold to the present day. It is valid for WAM-based systems, native code systems, and systems that do global analysis. In the WAM the simplification is done statically (at compile-time) and locally [79]. The simplification can also be done dynamically (with run-time tests) and globally. An example of dynamic simplification is clause selection (see Section 2.4.3). Examples of global simplification are global analysis (see Sections 2.4.5 and 2.4.6) and the two-stream unification algorithm (see Section 2.4.2). The latter compiles the unification of a complete term as a whole, instead of compiling each functor separately like the WAM.

2.1.3 Bridging the Gap Between DEC-10 Prolog and the WAM

An important early system is the C-Prolog interpreter, which was developed at Edinburgh in 1982 by Fernando Pereira, Luis Damas, and Lawrence Byrd. It is based on EMAS Prolog, a system completed in 1980 by Luis Damas. C-Prolog was one of the best interpreters, and is still a very usable system. It did much to create a Prolog programming community and to establish the Edinburgh standard. It is cheap, robust, portable (it is written in C), and fast enough for real programs.

There were several compiled systems that bridged the gap between the DEC-10 compiler (1977–1980) and the WAM (1983) [17, 28]. They include Prolog-X and NIP (New Implementation of Prolog). David Bowen, Lawrence Byrd, William Clocksin, and Fernando Pereira at Edinburgh were the main contributors in this work. These systems miss some of the WAM's good optimizations: separate choice points and environments, argument passing in registers instead of on the stack, and clause selection (indexing). David Warren left Edinburgh for SRI in 1981.

[2] They are not related.

Prolog		Imperative language
set of clauses	-----	program
predicate; set of clauses with same name and arity	-----	procedure definition; nondeterministic case statement
clause; axiom	-----	one branch of a nondeterministic case statement; if statement; series of procedure calls
goal invocation	-----	procedure call
unification	-----	parameter passing; assignment; dynamic memory allocation
backtracking	-----	conditional branching; iteration; continuation passing
logical variable	-----	pointer manipulation
recursion	-----	iteration

Figure 1: The Correspondence Between Logical and Imperative Concepts

According to Warren, the WAM design was an outcome of his own explorations and was not influenced by this work.

2.2 The Warren Abstract Machine (WAM)

By 1983 Warren had developed the WAM, a structure-copying execution model for Prolog that has become the de facto standard implementation technique [163]. The WAM defines a high-level instruction set that maps closely to Prolog source code. This section concisely explains the original WAM. In particular, the many optimizations of the WAM are given a uniform justification. This section assumes a basic knowledge of how Prolog executes [85, 115, 130] and of how imperative languages are compiled [3].

For several years, Warren's report was the sole source of information on the WAM, and its terse style gave the WAM an aura of inscrutability. Many people learned the WAM by osmosis, gradually absorbing its meaning. Nowadays, there are texts that give lucid explanations of the WAM and WAM-like systems [4, 85].

There are two main approaches to efficient Prolog implementation: emulated code and native code. Emulated code compiles to an abstract machine and is interpreted at run-time. Native code compiles to the target machine and is executed directly. Native code tends to be faster and emulated code tends to be more compact. With care, both approaches are equally portable (see Section 5.1). The original WAM is designed with an emulated implementation in mind. For example, its unification instructions are more suited to emulated code (see Section 3.1.4). The two-stream unification algorithm of Section 2.4.2 is more suited to native code.

2.2.1 The Relationship of the WAM to Prolog and Imperative Languages

The execution of Prolog is a natural generalization of the execution of imperative languages (see Figure 1). It can be summarized as:

Prolog = imperative language
+ unification
+ backtracking

As in imperative languages, control flow is left to right within a clause. The goals in a clause body are called like procedures. A goal corresponds to a predicate. When a goal is called, the clauses in the predicate's definition are chosen in textual order from top to bottom. Backtracking is chronological, *i.e.*, control goes back to the most recently made choice and tries the next clause. Hence, Prolog is a somewhat limited realization of logic programming, but in practice its trade-offs are good enough for a logical and efficient programming style to be possible [113].

The WAM mirrors Prolog closely, both in how the program executes and in how the program is compiled:

WAM = sequential control (call/return/jump instructions)
+ unification (get/put/unify instructions)
+ backtracking (try/retry/trust instructions)
+ optimizations (to use as little memory as possible)

The WAM has a stack-based structure, of which a subset is similar to imperative language execution models. It has call and return instructions and local frame (environment) management instructions. It is extended with instructions to perform unification and backtracking. These form the core of the WAM. Around this core, the WAM has added optimizations intended to reduce memory usage.

Prolog as executed by the WAM defines a close mapping between the terminology of logic and that of an imperative language (see Figure 1). Predicates correspond to procedures. Procedures always have a case statement as the first part of their definition. Clauses correspond to the branches of this case statement. Variables are scoped locally to a clause.[3] Goals in a clause correspond to calls. Unification corresponds to parameter passing and assignment. Other features do not map directly: backtracking, the single-assignment nature, and the modification of control flow with the cut operation. Cut is a language feature that increases the determinism of a program by removing choice points.

The WAM is a good intermediate language in the sense that writing a Prolog-to-WAM compiler and a WAM emulator are both straightforward tasks. A compiler and emulator can be built without a deep understanding of the internals of Prolog or the WAM.

[3]Global variables and self-modifying code are possible with the *assert/1* and *retract/1* built-ins. These built-ins are potentially nonlogical and certainly inefficient, and hence should be infrequent.

P	Program counter
CP	Continuation Pointer (top of return stack)
E	current Environment pointer (in local stack)
B	most recent Backtrack point (in local stack)
A	top of local stack
TR	top of TRail
H	top of Heap
HB	Heap Backtrack point (in heap)
S	Structure pointer (in heap)
Mode	Mode flag (read or write)
A1, A2, ...	Argument registers
X1, X2, ...	temporary variables

Table 1: The Internal State of the WAM

2.2.2 Data Structures and Memory Organization

Prolog is a dynamically typed language, *i.e.*, variables may contain objects of any type at run-time. Hence, it must be possible to determine the type of an object at run-time by inspection.[4] In the WAM, terms are represented as tagged words: a word contains a tag field and a value field. The tag field contains the type of the term (atom, number, list, or structure). See [52] for an exhaustive presentation of alternative tagging schemes. The value field is used for different purposes depending on the type: it contains the value of integers, the address of unbound variables and compound terms (lists and structures), and it ensures that each atom has a value different from all other atoms. Unbound variables are implemented as self-referential pointers, *i.e.*, they point to themselves. When two variables are unified, one of them is modified to point to the other.[5] Therefore it may be necessary to follow a chain of pointers to access a variable's value. This is called *dereferencing* the variable.

Table 1 shows how the internal state of the WAM is stored in registers. The purpose of most registers is straightforward. The HB register caches the value of H stored in the most recent choice point. The S register is used during unification of compound terms (with arguments): it points to an argument being unified. All arguments can be accessed one by one by successively incrementing S. Some instructions have different behaviors during read and write mode unification; the mode flag is used to distinguish between them (see Section 2.2.3). In the original WAM, the mode flag is implicit (it is encoded in the program counter).

The external state (stored in memory) is divided into six logical areas (see Figure 2): two stacks for the data objects, one stack (the PDL) to support unification, one stack (the trail) to support the interaction of unification and backtracking, one area as code space, and one area as a symbol table.

[4] Unless the type can be determined at compile-time.

[5] More precisely, variable-variable unification can be implemented with a Union-Find algorithm [91]. With this algorithm, unifying n variables requires $O(n\alpha(n))$ time, where $\alpha(n)$ is the inverse Ackermann function.

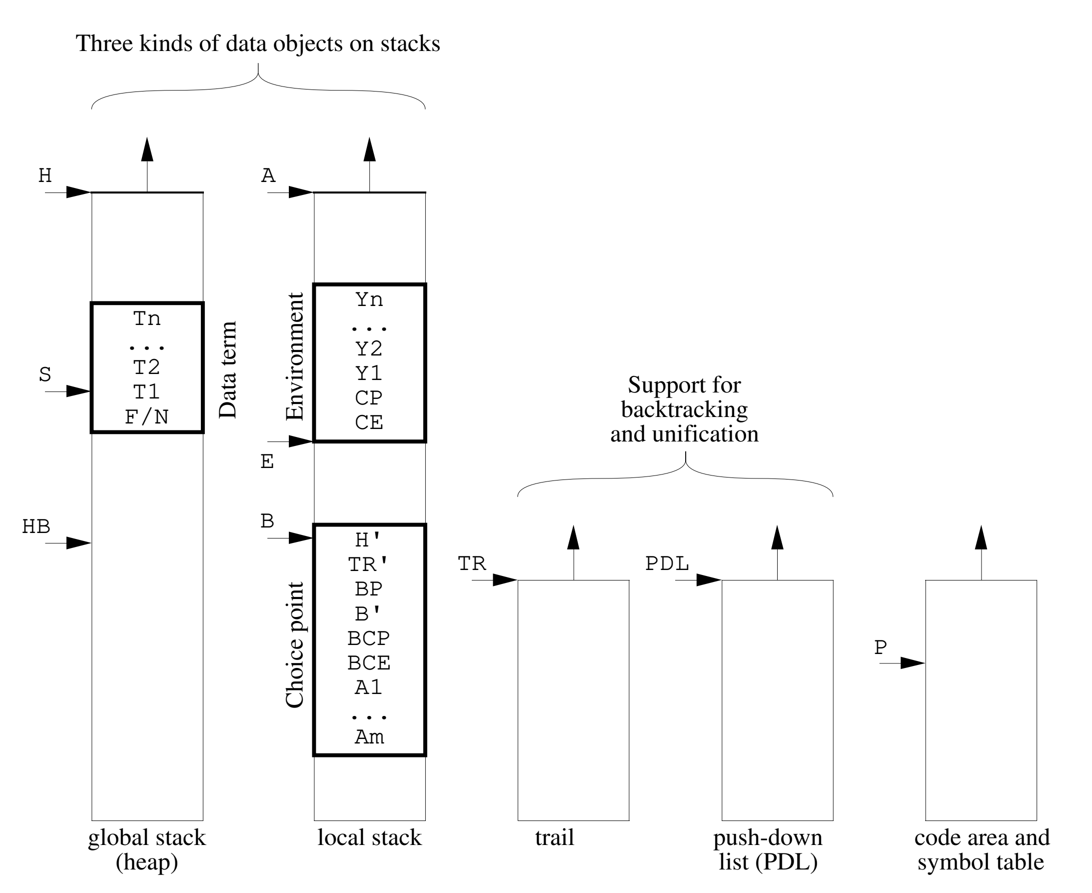

Figure 2: The External State of the WAM

- **The global stack** or **heap**. This stack holds lists and structures, the compound data terms of Prolog.

- **The local stack**. This stack holds environments and choice points. Environments (also known as local frames or activation records) contain variables local to a clause. Choice points encapsulate execution state for backtracking, *i.e.*, they are continuations. A variant model, the split-stack, uses separate stacks for environments and choice points. There is no significant performance difference between the split-stack and the merged-stack models. The merged-stack model uses more memory if choice points are created.

- **The trail**. This stack is used to save locations of bound variables that have to be unbound on backtracking. Saving the addresses of variables is called *trailing*, and restoring them to being unbound is called *detrailing*. Not all variables that are bound have to be trailed. A variable must only be trailed if it continues to exist on backtracking, *i.e.*, if its location on the global or local stack is older than the top of this stack stored in the most recent choice point. This is called the *trail condition*. Performing it is called the *trail check*.

- **The push-down list (PDL)**. This stack is used as a scratch-pad during the unification of nested compound terms. Often the PDL does not exist as a separate stack, *e.g.*, the local stack is used instead.
- **The code area**. This area holds the compiled code of a program. It is not recovered on backtracking.
- **The symbol table**. This area is not mentioned in the original article on the WAM. It holds various kinds of information about the symbols (atoms and structure names) used in the program. It is not recovered on backtracking. It contains the mapping between the internal representation of symbols and their print names, information about operator declarations, and various system-dependent information related to the state of the system and the external world. Because creating a new entry is relatively expensive, symbol table memory is most often not recovered on backtracking. It may be garbage collected. Systems that manipulate arbitrary numbers of new atoms (*e.g.*, systems with a database interface) must have garbage collection.

It is possible to vary the organization of the memory areas somewhat without changing anything substantial about the execution. For example, some systems have a single data area (sometimes called the *firm heap*) that combines the code area and symbol table.

2.2.3 The Instruction Set

The WAM instruction set, along with a brief description of what each instruction does, is summarized in Table 2. Unification of a variable with a data term known at compile-time is decomposed into instructions to handle the functor and arguments separately (see Figures 3 and 4). There are no **unify_list** and **unify_structure** instructions; they are left out because they can be implemented using the existing instructions. The **switch_on_constant** and **switch_on_structure** instructions fall through if A_1 is not in the hash table. The original WAM report does not talk about the **cut** operation, which removes all choice points created since entering the current predicate. Implementations of cut are presented in [4, 85]. A variable stored in the current environment (pointed to by E) is denoted by Y_i. A variable stored in a register is denoted by X_i or A_i. A register used to pass arguments is denoted by A_i. A register used only internally to a clause is denoted by X_i. The notation V_i is shorthand for X_i or Y_i. The notation R_i is shorthand for X_i or A_i.

A useful optimization is the variable/value annotation. Instructions annotated with “variable” assume that their argument has not yet been initialized, *i.e.*, it is the first occurrence of the variable in the clause. In this case, the unification operation is simplified. For example, the **get_variable** X_2, A_1 instruction unifies X_2 with A_1. Since X_2 has not yet been initialized, the unification reduces to a move. Instructions annotated with “value” assume that their argument has been initialized (*i.e.*, all later occurrences of the variable). In this case, full unification is done.

Figures 3 and 4 give the Prolog source code and the compiled WAM code for the predicate *append/3*. The mapping between Prolog and WAM instructions is straightforward (see Sec-

Loading argument registers (just before a call)	
put_variable V_n, R_i	Create a new variable, put in V_n and R_i.
put_value V_n, R_i	Move V_n to R_i.
put_constant C, R_i	Move the constant C to R_i.
put_nil R_i	Move the constant nil to R_i.
put_structure F/N, R_i	Create the functor F/N, put in R_i.
put_list R_i	Create a list pointer, put in R_i.
Unifying with registers (head unification)	
get_variable V_n, R_i	Move R_i to V_n.
get_value V_n, R_i	Unify V_n with R_i.
get_constant C, R_i	Unify the constant C with R_i.
get_nil R_i	Unify the constant nil with R_i.
get_structure F/N, R_i	Unify the functor F/N with R_i.
get_list R_i	Unify a list pointer with R_i.
Unifying with structure arguments (head unification)	
unify_variable V_n	Move next structure argument to V_n.
unify_value V_n	Unify V_n with next structure argument.
unify_constant C	Unify the constant C with next structure argument.
unify_nil	Unify the constant nil with next structure argument.
unify_void N	Skip next N structure arguments.
Managing unsafe variables (an optimization; see Section 2.2.4)	
put_unsafe_value V_n, R_i	Move V_n to R_i and globalize.
unify_local_value V_n	Unify V_n with next structure argument and globalize.
Procedural control	
call P, N	Call predicate P, trim environment size to N.
execute P	Jump to predicate P.
proceed	Return.
allocate	Create an environment.
deallocate	Remove an environment.
Selecting a clause (conditional branching)	
switch_on_term V, C, L, S	Four-way jump on type of A_1.
switch_on_constant N, T	Hashed jump (size N table at T) on constant in A_1.
switch_on_structure N, T	Hashed jump (size N table at T) on structure in A_1.
Backtracking (choice point management)	
try_me_else L	Create choice point to L, then fall through.
retry_me_else L	Change retry address to L, then fall through.
trust_me_else *fail*	Remove top-most choice point, then fall through.
try L	Create choice point, then jump to L.
retry L	Change retry address, then jump to L.
trust L	Remove top-most choice point, then jump to L.

Table 2: The Complete WAM Instruction Set

```
append([], L, L).
append([X|L1], L2, [X|L3]) :- append(L1, L2, L3).
```

Figure 3: The Prolog Code for *append/3*

append/3:	**switch_on_term** V_1,C_1,C_2,*fail*	Jump if variable, constant, list, structure.
V_1:	**try_me_else** V_2	Create choice point if A_1 is variable.
C_1:	**get_nil** A_1	Unify A_1 with nil.
	get_value A_2, A_3	Unify A_2 and A_3.
	proceed	Return to caller.
V_2:	**trust_me_else** *fail*	Remove choice point.
C_2:	**get_list** A_1	Start unification of list in A_1.
	unify_variable X_4	Unify head: move head into X_4.
	unify_variable A_1	Unify tail: move tail into A_1.
	get_list A_3	Start unification of list in A_3.
	unify_value X_4	Unify head: unify head with X_4.
	unify_variable A_3	Unify tail: move tail into A_3.
	execute *append/3*	Jump to beginning (last call optimization).

Figure 4: The WAM Code for *append/3*

tion 2.2.5). The **switch** instruction jumps to the correct clause or set of clauses depending on the type of the first argument. This implements first-argument selection (indexing). The choice point (**try**) instructions link a set of clauses together. The **get** instructions unify with the head arguments. The **unify** instructions unify with the arguments of structures.

The same instruction sequence is used to take apart an existing structure (read mode) or to build a new structure (write mode). The **get** instructions set the mode flag, which determine whether execution proceeds in read mode or write mode. For example, if **get_list** A_i sees an unbound variable argument, it sets the flag to write mode. If it sees a list argument, it sets the flag to read mode. If it sees any other type, it fails, *i.e.*, it backtracks by restoring state from the most recent choice point. The **unify** instructions have different behavior in read and write mode. The **get** instructions initialize the S register and the **unify** instructions increment the S register.

Choice point handling (backtracking) is done by the **try** instructions. The **try_me_else** L instruction creates a choice point, *i.e.*, it saves all the machine registers on the local stack. It is compiled just before the code for the first clause in a predicate. It causes a jump to label L on backtracking. The **try** L instruction is identical to **try_me_else** L, except that the destinations are switched: **try** immediately jumps to L. The **retry_me_else** L instruction modifies a choice point that already exists by changing the address that it jumps to on backtracking. It is compiled with clauses after the first but not including the last. This means that a predicate with n clauses is compiled with $n-2$ **retry_me_else** instructions. The **trust_me_else** *fail* instruction removes the top-most choice point from the stack. It is compiled with the last clause in a predicate.

2.2.4 Optimizations to Minimize Memory Usage

The core of the WAM is straightforward. What makes it subtle are the added optimizations. Because of these optimizations the WAM is extremely memory efficient. For programs with sufficient backtracking, a garbage collector is not necessary. The optimizations are explained in terms of the following classification of memory, from least to most costly to allocate, deallocate, and reuse.

- **Registers** (arguments, temporary variables). These are available at any time without overhead.
- **Short-lived memory** (environments on the local stack). This memory is recovered on forward execution, backtracking, and garbage collection.
- **Long-lived memory** (choice points on the local stack, data terms on the heap). This memory is recovered only on backtracking and garbage collection.
- **Permanent memory** (the code area and symbol table). This memory is recovered only by garbage collection.

With this classification, the optimizations can be explained as follows.

- **Prefer registers to memory**. There are three optimizations in this category.
  - **Argument passing**. All procedure arguments are passed in registers. This is important because Prolog is procedure-call intensive. For example, the most efficient way to iterate is through recursion. Backtracking can express iteration as well, but less efficiently.
  - **The return address**. Inside a procedure, the return address of the immediate caller is stored in the CP register. This optimization is closely related to the *leaf routine* calling protocol done in imperative language compilers.
  - **Temporary variables**. Temporary variables are variables whose lifetimes do not cross a call. That is, they are not used both before and after a call. Therefore they may be kept in registers. This definition of temporary variables simplifies and slightly generalizes Warren's original definition.
- **Prefer short-lived memory to long-lived memory**. There are three optimizations in this category.
  - **Permanent variables**. Permanent variables are variables that need to survive a call. They may not be kept in registers, but must be stored in memory. They are given a slot in the environment. This makes it easy to deallocate their memory if they are no longer needed after exiting the predicate (see unsafe variables, below).
  - **Environment trimming (last call optimization)**. Environments are stored on the local stack and recovered on forward execution just before the last call in a

procedure body. This optimization is known as the *tail recursion optimization* or more accurately, the *last call optimization*. This is based on the observation that an environment's space does not need to exist after the last call, since no further computation is done in the environment. The space can be recovered before entering the last call instead of after it returns. Because execution will never return to the procedure, the last call may be converted into a jump. For recursive predicates, this converts recursion into iteration, since the jump is to the first instruction of the predicate. The WAM generalizes the last call optimization to be done gradually during execution of a clause: the environment size is reduced ("trimmed") after each call in the clause body, so that only the information needed for the rest of the clause is stored. Trimming increases the amount of memory that is recovered by garbage collection.

  – **Unsafe variables**. A variable whose lifetime crosses a call must be allocated an unbound variable cell in memory (*i.e.*, in an environment or on the heap). If it is sure that the unbound variable will be bound before exiting the clause, then the space for the cell will not be referenced after exiting the clause. In that case the cell may be allocated in the environment and recovered with environment trimming. In the other case one is not sure that the unbound variable will be bound. This leads to the following space-time trade-off. The fastest alternative is to always create the variable on the heap. The most memory-efficient alternative is to create the variable on the environment and just before trimming the environment, to move the variable to the heap if it is unbound. The WAM has chosen the second alternative, and the variable being tested is referred to as an "unsafe variable".

- **Prefer long-lived memory to permanent memory**. Data objects (variables and compound terms) disappear on backtracking, and hence all allocated memory for them may be put on the heap. In a typical implementation this is not quite true. The symbol table and code area are not put on the heap, because their modifications (*i.e.*, newly interned atoms and asserted clauses) are permanent.

Measurements have been done of the unsafe variable trade-off for Quintus Prolog (see Section 3.1.4) and the VAM (see Section 2.5.1) [76]. Tim Lindholm measured the increase of peak heap usage for Quintus on a set of programs including Chat-80 [161] and the Quintus test suite and compiler. He found that the first alternative increases peak heap usage by 50 to 100% for Quintus (see Section 3.1.4). Because this leads to increased garbage collection and stack shifting, Lindholm concluded that unsafe variables are useful.

Andreas Krall measured the increase of peak heap usage on a series of small and medium-size programs for the VAM, which stores all unbound variables on the heap. He measured increases of from 4% to 26%, with an average of 15%. Because unsafe variables impose a run-time overhead (two comparisons instead of one for the trail check and run-time tests for globalizing variables), Krall concluded that unsafe variables are not useful.

The VAM and Quintus execution models are significantly different, so the VAM and Quintus measurements cannot be compared directly. My own view is that unsafe variables are useful since the run-time overhead is small and the reduction of heap usage is significant.

2.2.5 How to Compile Prolog to the WAM

Compiling Prolog to the WAM is straightforward because there is a close mapping between lexical tokens in the Prolog source code and WAM instructions. Figure 5 gives a scheme for compiling Prolog to the WAM. For simplicity, the figure omits register allocation and peephole optimization. This compilation scheme generates suboptimal code. One can improve it by generating **switch** instructions to avoid choice point creation in some cases. For more information on WAM compilation see [116, 148].

The clauses of predicate *p/3* are compiled into blocks of code that are linked together with **try** instructions. Each block consists of a sequence of **get** instructions to do the unification of the head arguments, followed by a sequence of **put** instructions to set up the arguments for each goal in the body, and a **call** instruction to execute the goal. The block is surrounded by **allocate** and **deallocate** instructions to create an environment for permanent variables. The last call is converted into a jump (an **execute** instruction) because of the last call optimization (see Section 2.2.4).

2.3 WAM Extensions for Other Logic Languages

Many WAM variants have been developed for new logic languages, new computation models, and parallel systems. This section presents three significant examples:

- The CHIP constraint system, which interfaces the WAM with three constraint solvers.
- The clp(FD) constraint system, which implements a *glass box* approach that allows constraint solvers to be written at the user level.
- The SLG-WAM, which extends the WAM with memoization.

2.3.1 CHIP

CHIP (Constraint Handling In Prolog) [2] is a constraint logic language developed at ECRC (see Section 3.1.7 for more information on ECRC). The system has been commercialized by Cosytec to solve industrial optimization problems. CHIP is the first compiled constraint language. In addition to equality over Prolog terms, CHIP adds three other computation domains: finite domains, boolean terms, and linear inequalities over rationals. The CHIP compiler is built on top of the SEPIA WAM-based Prolog compiler. The system contains a tight interface between the WAM kernel and the constraint solvers. The system extends the WAM to the C-WAM (C for Constraint). The C-WAM is quite complex: it has new data structures and over one hundred new instructions. Many instructions exist to solve commonly-occurring constraints quickly.

Measurements of early versions of CHIP showed that a large amount of trailing was being done, to the point that many programs quickly ran out of memory. This happened because the trailing

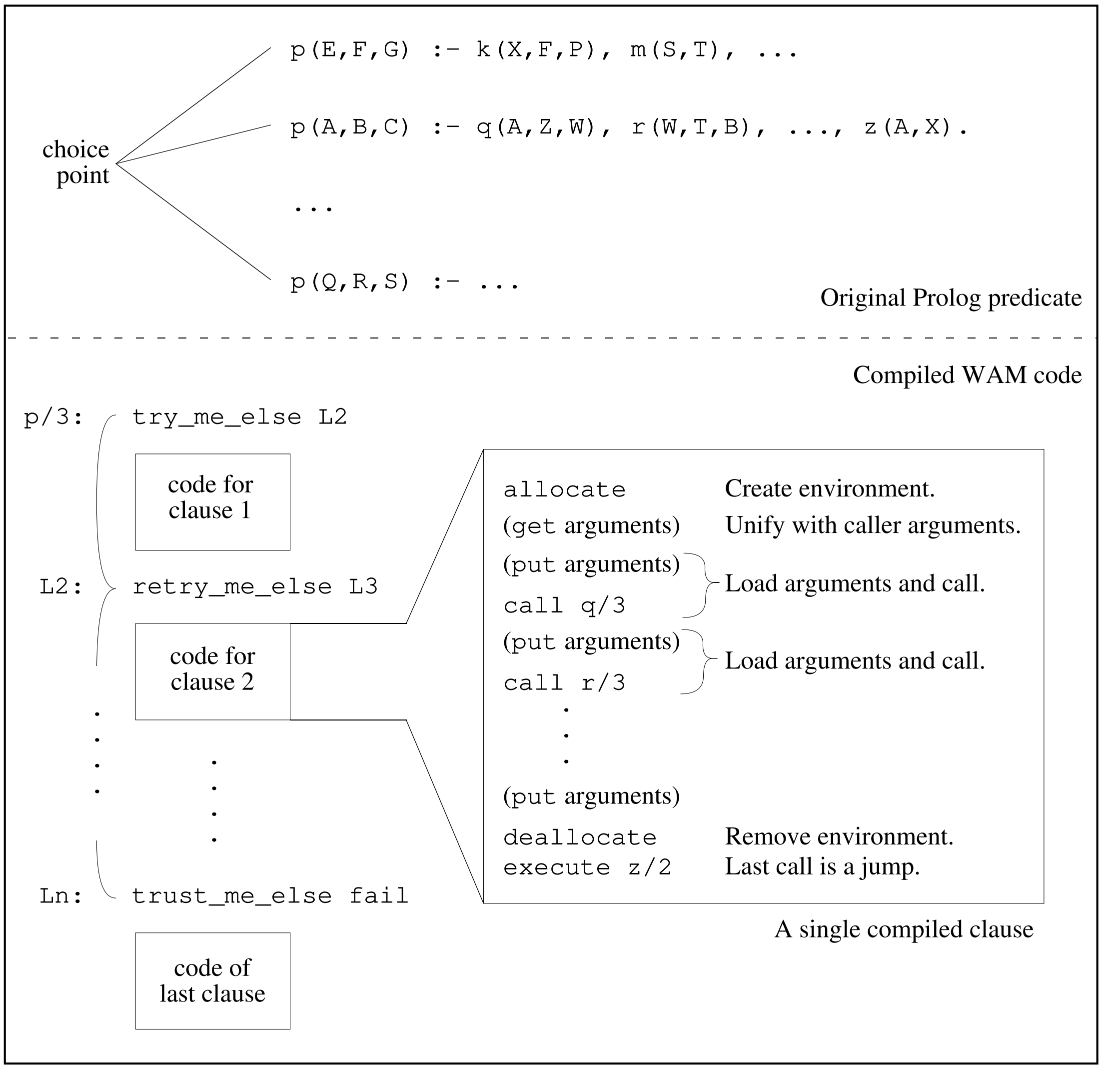

Figure 5: How to Compile Prolog to the WAM

was done with the WAM's trail condition (see Section 2.2.2). This condition is appropriate for equality constraints, which are implemented by unification in the WAM. For more complex constraints, the condition is wasteful because a variable's value is often modified several times between two choice points. The CHIP system reduces memory usage by introducing a different trail condition called "time-stamping" [1]. Each data term is marked with an identifier of the choice point segment the term belongs to (see Section 2.3.1). Trailing is only necessary if the current choice point segment is different from the segment stored in the term. Time-stamping is an essential technique for any practical constraint solver.

2.3.2 clp(FD)

The clp(FD) system [29, 40] is a finite domain solver integrated into a WAM emulator. It was built by Daniel Diaz and Philippe Codognet at INRIA (Rocquencourt, France). It uses a *glass box* approach. Instead of considering a constraint solver as a black box (in the manner of CHIP), a set of primitive operations is added that allows the constraint solver to be programmed in the user language. The resulting system outperforms the hard-wired finite domain solver of CHIP.

In clp(FD), a single primitive constraint is added to the system, namely the range constraint **X in R**, where X is a domain variable and R is a range (*e.g.*, **1..10**). Instead of just using constant ranges, the idea is to introduce what are known as *indexical* ranges, *i.e.*, ranges of the form **f(Y)..g(Y)** or **h(Y)** where **f(Y)**, **g(Y)**, and **h(Y)** are functions of the domain variable Y. A set of these functions that do local propagation is built-in. For example, the system provides the constraints **X in min(Y)..max(Y)** and **X in dom(Y)** with the obvious meanings. Arithmetic constraints such as **X=Y+Z** and boolean constraints such as **X=Y and Z** can be written in terms of indexical range constraints.

Indexical range constraints are smoothly integrated into the WAM by providing support for domain variables and suspension queues for the various indexical functions [40]. The time-stamping technique of CHIP is used to reduce trailing.

2.3.3 SLG-WAM

Memoization is a technique that caches already-computed answers to a predicate. By adding memoization to Prolog's resolution mechanism, one obtains an execution model that can do both top-down and bottom-up execution. For certain algorithms, this model executes simple logical definitions with a lower order of complexity than a pure top-down execution would. For example, the recursive definition of the Fibonacci function runs in linear time rather than exponential time. More realistic examples are parsing and dynamic programming.

One realization of memoization is OLDT resolution (Ordered Linear resolution of Definite clauses with Tabulation) [131]. A recent generalization, SLG resolution [27], handles negation as well. This has been implemented in an abstract machine, the SLG-WAM (previously called the OLDT-WAM), and realized in the XSB system (see Section 3.1.8). The current implementation executes Prolog code with less than 10% overhead relative to the WAM as implemented in XSB, and is much faster than deductive database systems [132]. An important source of overhead is the complex trail: it is a linked list whose elements contain the address and old contents of a cell.

2.4 Beyond the WAM: Evolutionary Developments

The WAM was a large step towards the efficient execution of Prolog. From the viewpoint of theorem proving, Prolog is extremely fast. But there is still a large gap between the efficiency of the WAM and that of imperative language implementations. As people started using Prolog for standard programming tasks, the gap became apparent and people started to optimize their systems. This section discusses the gap and some of the clever ideas that have been developed to close it.

2.4.1 Chinks in the Armor

This section lists the limits to Prolog performance and their causes.

- WAM instructions are too coarse-grained to perform much optimization. For example, many WAM instructions perform an implicit dereference operation, even if the compiler can determine that such an operation is unnecessary in a particular case. In practice, dereference chains are short: dynamic measurements on real programs show that two thirds are of length zero (no memory reference is required), one third are of length one, and $<1\%$ are of length two or greater [145]. Despite these statistics, dereferencing is expensive. For example, Aquarius on the VLSI-BAM, a high-performance system with hardware support, spends 9% of its total execution time doing dereferencing [152].
- The majority of predicates written by human programmers are intended to give at most one solution, *i.e.*, they are deterministic. These predicates are in effect case statements, yet they are too often compiled in an inefficient manner using the full generality of backtracking (which implies saving the machine state and repeated failure and state restoration). The WAM's first-argument selection is inadequate to compile these predicates efficiently (see Section 2.4.3). Measurements of Prolog applications support this assertion:
  - Tick shows that shallow backtracking (backtracking from clause to clause within a single predicate) dominates even for well-written deterministic programs. Choice point references constitute about half (45–60%) of all data references [143].
  - Touati and Despain show that at least 40% of all choice point and fail operations can be removed through optimization [145].
- The single-assignment nature of Prolog (*i.e.*, a variable can only be assigned one value in forward execution) needs to be handled well. In a straightforward implementation it is time-consuming to modify large data structures incrementally, because the programmer may use copying of terms to represent incremental changes, and the implementation will not optimize this copying away. This problem, also known as the *copy avoidance*

problem, is a special case of the general problem of efficiently representing state modification in logic. It is impossible to use large data structures with the same efficiency as in procedural languages unless the compiler is able to introduce destructive assignment (overwriting of memory locations) in the implementation. Section 5.1 gives suggestions on how to get around this problem.

- Prolog has dynamic typing (variables may contain values of any type) and dynamic memory allocation (all data objects are allocated at run-time). Both of these cost execution time. They should be compiled statically wherever possible.

- Programming style has a large effect on a program's efficiency. Prolog programming is at a high level of abstraction, so it hides many details of the implementation from the programmer, making it difficult to improve efficiency when it is important to do so. For example, adding a single cut can make the difference between a program that runs fast and one that thrashes. This is possible even if the cut does not change the operational semantics of the program. The thrashing behavior is caused by a pile-up of choice points during deterministic (forward) execution. Because the choice points encapsulate execution states that remain accessible through potential backtracking, their memory is not recovered by garbage collection.

- The apparent need for architectural support. So-called "general-purpose" architectures are in fact optimized for imperative languages and number crunching. To run Prolog equally well, either the compiler must do more work, or conceivably the architecture should be modified. Some experiments have been done with architectures optimized for Prolog (among others, the PSI-II, KCM, and VLSI-BAM, see Section 3.2), but the true architectural needs of Prolog are a moving target. They depend on the execution model and the sophistication of the compiler. As better compilers have been developed, the perceived architectural needs of Prolog have been getting smaller and smaller. One need likely to stay for a long time is a fast memory system. Prolog's dynamic nature requires frequent pointer dereferencing. There are no compilation techniques on the horizon that are likely to reduce the resulting need for a fast memory system (see Section 5.1).

2.4.2 How to Compile Unification: The Two-Stream Algorithm

This section presents the two-stream unification algorithm, an elegant scheme for compiling unification that is more efficient than the WAM for native code implementation. Rough measurements comparing unification times of the VLSI-PLM (a microcoded WAM) and the VLSI-BAM (see Section 3.2.3) show a speedup factor of two to three [153] in favor of the latter. This algorithm was independently reinvented at least four times by different people at about the same time: Mohamed Amraoui at the Université de Rennes I [8], André Mariën and Bart Demoen at BIM and KUL [86, 88], Kent Boortz at SICS [16], and Micha Meier at ECRC [94]. Write mode propagation was discussed earlier by Andrew Turk [146].

Figures 6 and 8 show how the unification X=f(g(A),h(B)) is compiled in the WAM and by the two-stream algorithm. The actions of the instructions are represented as *primitive constraints* of two kinds: functor constraints (such as X=(f/2)) and argument constraints (such as X.1=Y).

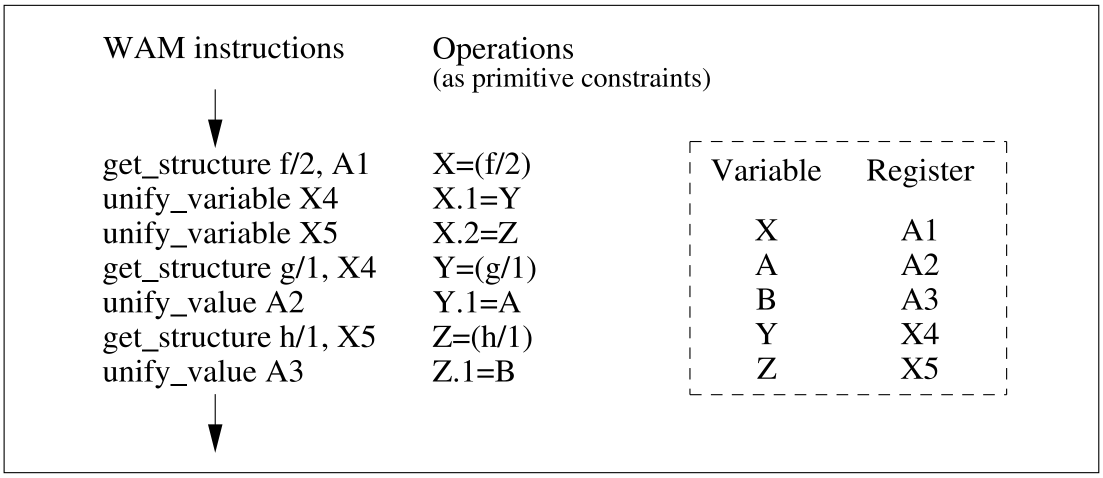

Figure 6: The WAM Compilation of the Unification X=f(g(A),h(B))

Functor and argument constraints correspond to the **get** and **unify** instructions in the WAM. An important advantage of the primitive constraint representation over the WAM is that the constraints may be executed in any order. In addition to providing a powerful conceptual description of the WAM, primitive constraints are useful in compiling more advanced logic languages [6, 84, 117].

The WAM compiles unification as a single sequence of instructions (see Figure 6). This has several problems:

- **Write mode is not propagated to subterms**. For example, the unification X=f(g(a)) is compiled as X=f(T), T=g(a). These two unifications are compiled independently. If X is unbound, the fact that T is created as an unbound variable in the first unification is not propagated to the second unification. This means a superfluous dereference, a superfluous trail check, and a superfluous binding.

- **Instructions have modes**. All instructions have two modes of execution, read mode and write mode. The current mode is stored in a global mode flag, which is set in **get_list** and **get_structure** instructions and tested in all **unify** instructions. Some implementations (*e.g.*, the intended implementation of the original WAM report, and Quintus) encode the mode flag in the program counter, which avoids the testing overhead.

- **Poor translation to native code**. The straightforward method for generating native code is to macro-expand the WAM instructions. This means that the read and write mode parts are interleaved, which results in many jumps. This is less of a problem on a microcoded machine since microcode jumps are often free (the destination address is part of the microword).

The key insight is that unification should be compiled into *two* instruction streams, one for read mode and one for write mode, with conditional jumps between them in both directions. With

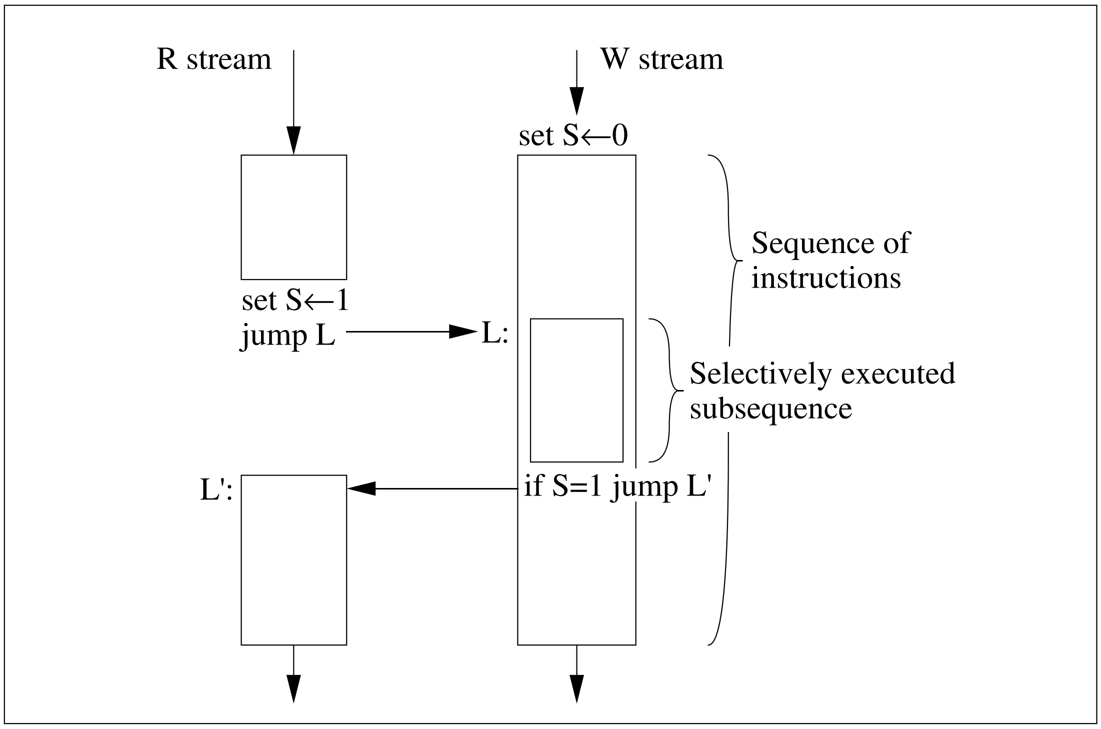

Figure 7: How to Execute a Particular Subsequence with Low Overhead

two streams one avoids superfluous operations while keeping a linear code size. The practical problems that remain are how to configure the instructions so they work correctly despite being jumped to from different places, and how to minimize bookkeeping overhead for the jumps.

Figure 7 illustrates a technique to execute any subsequence of a main instruction sequence with very little overhead. The idea is to give the main sequence an identifier (say, the integer 0) and the subsequence a different identifier (say, the integer 1). Then a single conditional jump is all that is required. If the subsequence is non-contiguous, then a single conditional jump to the next segment is needed per contiguous segment of the subsequence. If more than one subsequence has to be selected, then a unique identifier is needed for each one. The subsequences may be overlapping.

With the idea of selective execution in mind, arrange the primitive constraints of the term according to a depth-first traversal of the term (Figure 8). The resulting sequence satisfies the property that each subterm corresponds to a contiguous sequence of instructions. This is all one needs to implement the algorithm. At run-time, unification follows the read mode stream, and selectively executes contiguous parts of the write mode stream for subterms to be created.

A reduction of bookkeeping overhead is possible based on a second property of the sequence. Nested terms correspond to nested sequences of instructions. Number each subterm with an integer representing its *nesting level* within the term. With this numbering, an adjacent sequence of conditional jumps back to the read mode stream can be collapsed into a single

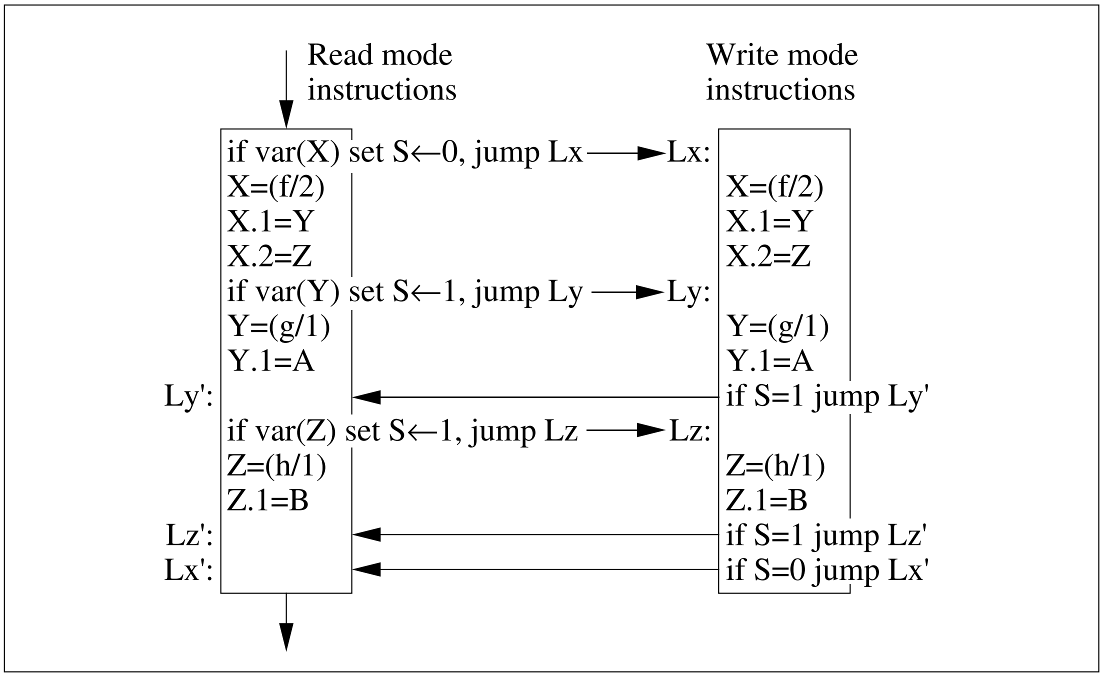

Figure 8: The Two-Stream Compilation of the Unification X=f(g(A),h(B))

conditional jump (changing the condition from "=" to "≤"). In Figure 8, the two conditional jumps **if** S=1 **jump** *Lz'* and **if** S=0 **jump** *Lx'* can be rewritten as the single jump **if** S≥0 **jump** *Lz'*. To collapse the maximum number of jumps, reorder the arguments of all subterms to unify the most complex subterms last.

The advantages of the two-stream algorithm are:

- **Low overhead**. The bookkeeping overhead is a small constant factor. The only bookkeeping is the set of jumps and register moves needed to manage the selective execution of subsequences. This is small compared to the work done in the primitive constraints. There is no explicit mode flag.

- **Downward propagation of write mode**. The write mode of a term is propagated at compile-time to all of its subterms. There are no superfluous dereferences, trail checks, or bindings.

- **Upward propagation of read mode**. The read mode of a term is propagated at compile-time to its siblings and ancestors.

- **Linear code size**. This contrasts with the algorithm of [150], which expands all cases without any sharing. That algorithm has zero bookkeeping overhead, but exponential code sizes occur in practice.

- **Efficient expansion to native code**. The number of instructions generated is about double that of the WAM, but the instructions themselves have less than half the complexity. The primitive constraints of Figure 8 are expanded differently in the read mode and write mode streams. Essentially, the internal operations of the WAM instructions have been made visible and arranged in an efficient order. There are no jumps inside the primitive constraints, but only between them, and then only when it is necessary to choose between read and write mode.

2.4.3 How to Compile Backtracking: Clause Selection Algorithms

This section surveys the clause-selection algorithms that have been developed since the WAM. The WAM supports first-argument selection. It has instructions that can choose clauses based on the main functor of the first argument. If all of a predicate's clauses contain different main functors, then a hash table can be constructed and calling the predicate will avoid a choice point creation when the first argument is not a variable. In the general case, predicates can be compiled to create at most one choice point between entry and the execution of the first clause [24, 148]. The original WAM report describes a two-level indexing scheme which creates up to two choice points [163].

Many programs cannot profit from first-argument selection. For example, selection may depend on more than one argument. The following example is extracted from an actual program. The first two arguments are integer inputs, the third is an output (all numbers are in base two):

```
get_relop(2'001, 2'001, 2'000).
get_relop(2'001, 2'010, 2'011).
get_relop(2'001, 2'011, 2'000).
... 33 more clauses ...
```

The second example is a predicate in which selection depends on arithmetic comparisons instead of unification only:

```
max(A, B, C) :- A≤B, C=B.
max(A, B, C) :- A>B, C=A.
```

In general, selection is possible if the compiler can determine that only a subset of the clauses in the definition can possibly succeed, given some particular argument types at the call. An appropriate definition of *type* is given in Section 2.4.5.

An ideal clause-selection algorithm would generate code that has the following properties:

- It takes advantage of argument types to try only the clauses that can possibly succeed.
- It avoids all useless choice point creations.
- Its size is linear in the size of the program.

- It creates choice points incrementally, *i.e.*, choice points contain only that part of the execution state that needs to be saved.
- Its performance degradation is gradual if insufficient type information is known.

There is no published algorithm that satisfies all these conditions. There are published algorithms that satisfy some of the conditions and do better clause selection than the WAM. Several algorithms create a selection tree or graph, *i.e.*, a series of tests that determine which subset of clauses to try given particular arguments (*e.g.*, [167]). Generating a naive selection tree may result in exponential code size for predicates encountered in real-world programs. The following algorithms are noteworthy:

- Van Roy, Demoen, and Willems [149] present a compilation algorithm that generates a naive selection tree and creates choice points incrementally. The algorithm compiles clauses with four entry points, depending on whether or not there are alternative clauses, and whether or not a previously executed clause has created a choice point. The algorithm was not implemented.
- Carlsson [25] has implemented a restricted version of the above algorithm in SICStus Prolog. Meier [92] has done a similar implementation in KCM-SEPIA. Choice point creation is split into two parts. The **try** and **try_me_else** instructions are modified to create a partial choice point that only contains P and TR. A new instruction, **neck**, is added. If a partial choice point exists when **neck** is executed, then the remaining registers are filled in. Two entry points are created for each clause: one when there are alternative clauses, and one where there are none. A **neck** instruction is only included in the first case. In SICStus, this algorithm results in a performance improvement of 7% to 15% for four large programs, at a cost of a 5% to 10% increase in code size.
- Hickey and Mudambi [61] present compilation algorithms to generate a tree of tests and to minimize work done in backtracking. One of their selection algorithms results in a tree that has a quadratic worst-case size. They improve choice point management. The **try** instruction only stores registers needed in clauses after the first clause. The **retry** and **trust** instructions restore only those registers needed in the clause and remove the registers not needed in subsequent clauses. The latter operation lets the garbage collector recover more memory. The technique of improved choice point management was independently invented earlier by Andrew Turk [146] and later by Van Roy [153]. The technique has not yet been quantitatively evaluated.
- Kliger [71, 72] presents a compilation algorithm that generates a directed acyclic graph of tests (a "decision graph"). The algorithm was extended by Korsloot and Tick for nondeterminate ("don't know") predicates [74]. The graph has two important properties. First, it never does worse than first-argument selection. Second, it has size linear in the number of clauses. This follows from the property that each clause corresponds to a unique path through the graph. Linear size is essential when compiling predicates consisting of a large number of clauses.

- The Aquarius system [153, 154] produces a selection graph for disjunctions containing three kinds of tests: unifications, type tests, and arithmetic comparisons. It uses heuristics to decide which tests to do first and whether to use linear search or hashing for table lookup. The nodes in the graph partition the tests occurring in the predicate. Each node corresponds to a subset of these tests. Unifications are only used as tests if it can be deduced from the predicate's type information that they will be executed in read mode. The *type enrichment* transformation adds type information to a predicate that lacks it. The performance of the resulting code is therefore always at least as good as first-argument selection. The *factoring* transformation allows the system to take advantage of tests on variables inside of terms, by performing the term unification once for all occurrences of the term. The problem with Aquarius selection is similar to that of the naive selection tree: if too much type information is given, then the selection graph may become too large.

- The Parma system [140] uses techniques similar to Aquarius. It produces efficient indexing code for the same three kinds of tests. To improve the clause selection, Parma uses transformations analogous to type enrichment and factoring. It uses optimal binary search for table lookup. Taylor's dissertation discusses how to choose between linear search, binary search, jump tables, and hashing.

2.4.4 Native Code Compilation

One way to improve the performance of a WAM-based system is to add instructions. For example, instructions can be added to do efficient arithmetic and to index on multiple arguments. Common instruction sequences can be collapsed into single instructions. This is quick to implement, but it is inherently a short-term solution. As the number of instructions increases, the system becomes increasingly unwieldy.

The main insight in speeding up Prolog execution is to represent the code in terms of simple instructions. The first published experiments using this idea were done in 1986 by Komatsu *et al* [73, 135] at IBM Japan. These experiments gave the first demonstration that specialized hardware is not essential for high-performance execution of Prolog. Compilation is done in three steps. The first step is to compile Prolog into a WAM-like intermediate code. In the second step the WAM-like code is translated into a directed graph. The graph is optimized using rewrite rules. In the final step, the result is translated into PL.8 intermediate code and compiled with an optimizing compiler. For several small programs, this system demonstrated a fourfold performance improvement using mode hints given by the programmer.

Around 1988, Andrew Taylor and I independently set about building full systems (Parma and Aquarius) that compile directly to a simple instruction set, using global analysis to provide information for optimizations. Both Parma and Aquarius bypass WAM instructions entirely during compilation. We were confident that the fine granularity of the instruction set would allow us to express all optimizations. Taylor presented results for his Parma system in two important papers [138, 139]. The first paper presents and evaluates a practical global analysis technique that reduces the need for dereferencing and trailing. The second paper presents

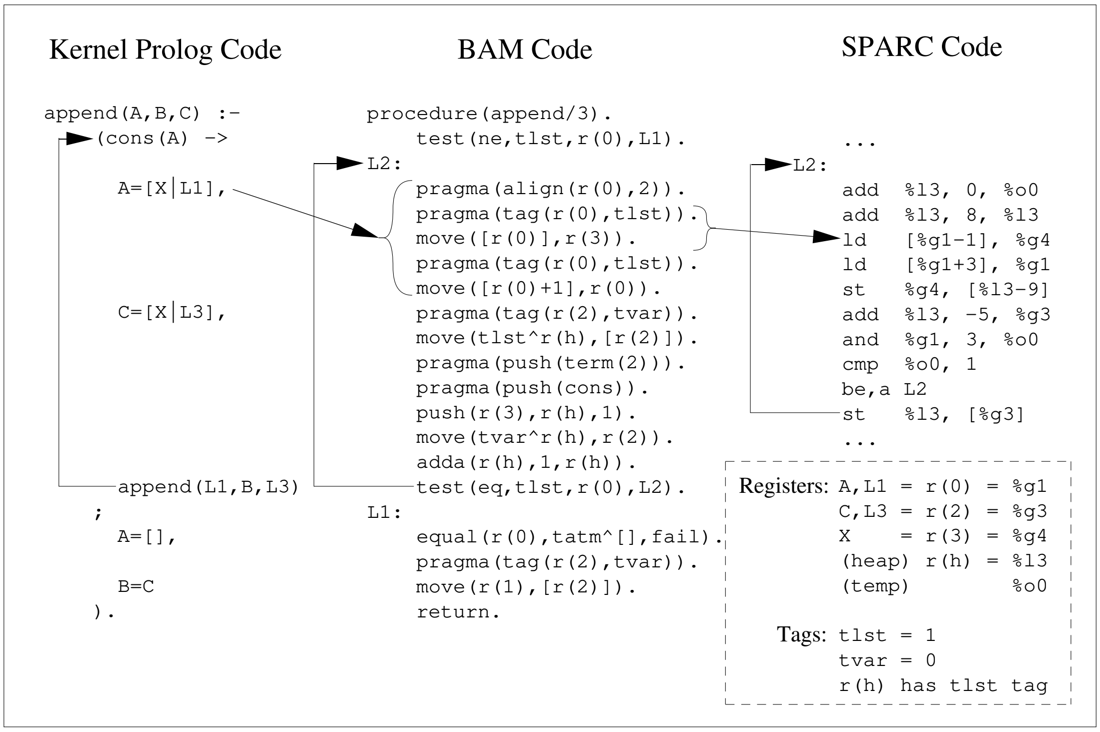

Figure 9: The Aquarius SPARC Code for *append/3* in Naive Reverse

performance results for Parma on a MIPS processor. The first results for Aquarius were presented in [63], which describes the VLSI-BAM processor and its simulated performance. A second paper measures the effectiveness of global analysis in Aquarius [152]. Both the Parma and Aquarius systems vastly outperform existing implementations. They prove the effectiveness of compiling directly to a low-level instruction set using global analysis to help optimize the code.

An important idea in both systems is *uninitialized variables*. An uninitialized variable is defined to be an unbound variable that is *unaliased*, *i.e.*, there is exactly one pointer to it. An uninitialized variable can be represented more efficiently than a standard WAM variable. Beer first proposed the idea of uninitialized variables after he noticed that most unbound variables in the WAM are bound soon afterwards [12]. For example, this is true for output arguments of predicates. WAM variables are created as self-referencing pointers in memory, and need to be dereferenced and trailed before being bound. This is time-consuming. Beer represents variables as pointers to memory words that have not been initialized. He introduces several new tags for these variables and keeps track of them at run-time. The creation of uninitialized variables is simpler and they do not have to be dereferenced or trailed. Binding them reduces to a single store operation. In Parma and Aquarius, these variables are derived by analysis at compile-time. They use the same tag as other variables.

Aquarius supports a further specialization: the "uninitialized register" variable. This idea is due to Bruce Holmer. This variable is an output that is passed in a register. No memory is allocated for uninitialized registers, unlike standard uninitialized variables. This reduces the space advantage of unsafe variables. The use of uninitialized registers allows Aquarius to run recursive integer functions faster than popular implementations of C [154].[6] In principle, all uninitialized variables can be transformed into uninitialized registers. In practice, to avoid losing last call optimization (see Section 2.2.4) only a subset is transformed [153]. The trade-off with last call optimization has not yet been studied quantitatively.

Figure 9 shows the Aquarius intermediate codes (kernel Prolog and BAM code) and the SPARC code generated for *append/3* in naive reverse. See Figures 3 and 4 for the Prolog source code and WAM code. Kernel Prolog is Prolog without syntactic sugar and extended with efficient conditionals, arithmetic, and cut.

The BAM (Berkeley Abstract Machine) is an execution model with a memory organization similar to the WAM. The BAM defines a load-store instruction set supplemented with tagged addressing modes, pragmas, and five Prolog-specific instructions (dereference, trail, general unification, choice point manipulation, and environment manipulation). Pragmas are not executable but give information that improves the translation to machine code.

In the SPARC code, tags are represented as the low two bits of a 32-bit word. This is a common representation that has low overhead for integer arithmetic and pointer dereferencing [52]. The tag of a pointer is always known at compile-time (it is put in a pragma). When following a pointer, the tag is subtracted off at zero cost with the SPARC's register+displacement addressing mode. The compiler derives the following types for *append/3*:

```
:- mode((append(A,B,C) :-
        ground(A), rderef(A),
        ground(B), rderef(B),
        uninit(C))).
```

An uninitialized argument is represented by `uninit`. A `ground` argument contains no unbound variables. A recursively dereferenced (`rderef`) argument is dereferenced and its arguments are recursively dereferenced. This type generalizes the DEC-10 mode:

```
:- mode append(++, ++, −).
```

which states that the first two arguments are ground and the last argument is an unbound variable.

[6] I posted this result to the Internet newsgroup comp.lang.prolog in February 1991, with the comment: "Don't believe it any more that there is an inherent performance loss when using logic programming". There was a barrage of responses, ranging from the incredulous (and incorrect) comment "Obviously, he's comparing apples and oranges, since the system must be doing memoization" to the encouraging "That's telling 'em Peter".

2.4.5 Global Analysis

Global analysis of logic programs is used to derive information to improve program execution. Both type and control information can be derived and used to increase speed and reduce code size. The analysis algorithms studied so far are all instances of a general method called *abstract interpretation* [34, 35, 69]. The idea is to execute the program over a simpler domain. If a small set of conditions are satisfied, this execution terminates and its results provide a correct approximation of information about the original program. Le Charlier *et al* [80, 81] have performed an extensive study of abstract interpretation algorithms and domains and their effectiveness in deriving types. Getzinger [50] has recently presented an extensive taxonomy of analysis domains and studied their effects on execution time and code size.

Since Mellish's early work in 1981 and 1985 [96, 98], global analysis has been considered useful for Prolog implementation. This section summarizes the work that has been done in making analysis part of a real system. By *type* we denote any information known (at compile-time or at run-time) about a variable's value at run-time. A *mode* is a restricted type that indicates whether the variable is used as an input (nonvariable) or an output (unbound variable). Useful types include argument values, compound structures, dependencies between variables, and operational information such as the length of dereference chains (see also Sections 2.4.6 and 5.1).

In 1982, Lee Naish performed an experiment with automatically generated *control* information for MU-Prolog [106]. Control information is all information related to the execution order of a program's procedures. The MU-Prolog interpreter supports *wait* declarations. A "wait" declaration defines a set of arguments of a predicate that may not be constructed by a call (*i.e.*, unified in write mode). When a call attempts to construct a term in any of these arguments, then the call delays until the argument is sufficiently instantiated so that no construction is done (*i.e.*, the argument is unified in read mode). This provides a form of coroutining. The automatic generation of "wait" declarations is based on a simple heuristic: to delay rather than guess one of an infinite number of bindings.[7] A "wait" declaration is inserted for each recursive call that does not progress in its traversal of a data structure. This algorithm was implemented and tested on some small examples. It significantly reduces the programmer's burden in managing control, but it does not always help: if the clause head is as general as the recursive call then no "wait" declaration is generated, even though one might be necessary.

A later system, NU-Prolog, supports *when* declarations. These are both more expressive and easier to compile into efficient code (see Section 3.1.3). A "when" declaration is a pattern containing a term with optional variables and a nested conjunction and/or disjunction of nonvariable and ground tests on these variables. Variables may not occur more than once in the term. A "when" declaration is true if unification between the term and the call succeeds and does not bind any variables in the call. A call will delay until its "when" declarations are true. This is called *one-way* unification or *matching*. NU-Prolog contains an analyzer that derives "when" declarations.

[7] This heuristic is closely related to the "Andorra principle" [33, 55]. The main difference is that the heuristic is applied at analysis time whereas the Andorra principle is applied at run-time.

In 1988, Richard Warren, Manuel Hermenegildo, and Saumya Debray did the first measurements of the practicality of global analysis in logic programming [60, 164]. They measured two systems, MA^3, the MCC And-parallel Analyzer and Annotator, and Ms, an experimental analysis scheme developed for SB-Prolog. The paper concludes that both dataflow analyzers are effective in deriving types and do not unduly increase compilation time.

In 1989, André Mariën *et al* [87] performed an interesting experiment in which several small Prolog predicates (recursive list operations) were hand-compiled with four levels of optimization based on information derivable from a global analysis. The levels progressively include unbound variable and ground modes, recursively defined types, lengths of dereference chains, and liveness information for compile-time garbage collection. Execution time measurements show that each analysis level significantly improves speed over the previous level. This experiment shows that a simple analysis can achieve good results on small programs.

Despite this experimental evidence, there was until 1993 no generally available sequential Prolog system that did global analysis, and since 1988 only a few research systems doing analysis. Why is this? I think the most important reason is that other areas of system development were considered more important. Commercial systems worked on improving their development environments: source-level debugging, a proper foreign language interface, and useful libraries. Research systems worked in other areas such as language design and parallelism. A second reason may be that the structure of the WAM (high-level compact instructions) does not lend itself well to the optimizations that analysis supports. A whole new instruction set would be needed, and the development effort involved may have seemed prohibitive given the existing investment in the WAM. A third reason is that analysis was erroneously considered impractical.

Currently, there are at least seven systems that do global analysis of logic programs:

- Ms, the analyzer for SB-Prolog, written by Saumya Debray. Ms derives ground and nonvariable types.
- MA^3, the analyzer for &-Prolog, written by Manuel Hermenegildo and Richard Warren. MA^3 derives variable sharing (aliasing) and groundness information. This information is used to eliminate run-time checks in the And-parallel execution of Prolog. This was the first practical application of abstract interpretation to logic programs. The &-Prolog system both derives information and uses it for optimization. PLAI, the successor to MA^3, subsumes it and has been extended to analyze programs in constraint languages [49] and languages with delaying [90].
- The FCP(:,?) compiler (Flat Concurrent Prolog with Ask and Tell guards and read-only variables), written by Shmuel Kliger, has a global analysis phase [72].
- The Parma system, written by Andrew Taylor, is an implementation of Prolog with global analysis targeted to the MIPS processor [140].
- The Aquarius system is an implementation of Prolog with global analysis targeted to the VLSI-BAM processor and various general-purpose processors [58, 153]. An extensive

System	Speedup factor		Code size reduction	
	Small	Medium	Small	Medium
Aquarius	1.5	1.2	2.2	1.8
Parma	3.0	2.1	2.9	2.0

Table 3: The Effectiveness of Analysis for Small and Medium-Size Programs

study of improved analyzers and their integration in the Aquarius system is given in [50].

- The MU-Prolog analyzer generates "wait" declarations for coroutining [106]. Its improved NU-Prolog version generates "when" declarations.

- The IBM Prolog analyzer. It determines whether choice points have been created or destroyed during execution of a predicate, and whether there are pointers into the local stack. This improves the handling of unbound variables and the management of environments. The IBM Prolog analyzer has been available since 1989 (see Section 3.1.6). There is no published information on the analyzer.

Of these systems, five were developed for sequential Prolog (the MU-Prolog and IBM Prolog analyzers, Ms, Parma, and Aquarius) and two for parallel systems (MA^3 and the FCP(:,?) analyzer). Three (MA^3, Parma, and Aquarius) have been integrated into Prolog systems and their effects on performance evaluated and published [60, 140, 153]. The analysis domains of Aquarius and Parma are shown in Figure 10. For both analyzers the analysis time is linear in program size and performance is adequate. Four analyzers (MA^3, FCP(:,?), Aquarius, and IBM Prolog) are robust enough for day-to-day programming.

The effect of the Aquarius and Parma analyzers on speed and code size is shown in Table 3. The "Small" column refers to a standard set of small benchmarks (between 10 and 100 lines). The "Medium" column refers to a standard set of medium-size benchmarks (between 100 and 1000 lines). These benchmarks are well-known in the Prolog programming community [156]. They do tasks for which Prolog is well-suited and are written in a good programming style. The numbers are taken from [140, 152, 153]. The numbers can be significantly improved by tuning the programs to take advantage of the analyzers.

For the medium-size benchmarks, the Aquarius analyzer finds uninitialized, ground, nonvariable, and recursively dereferenced types in 23%, 21%, 10%, and 17% of predicate arguments, respectively, and 56% of predicate arguments have types.[8] One third of the uninitialized types are uninitialized register types, so about one twelfth of all predicate arguments are outputs passed in registers. On the VLSI-BAM this results in a reduction of dereferencing from 11% to 9% of execution time and a reduction of trailing from 2.3% to 1.3% of execution time.

The Parma analyzer's domain has been split into parts and their effects on performance measured separately. For the medium-size benchmarks, performance is improved through dereference chain analysis by 14%, trailing analysis by 8%, structure/list analysis by 22%, and uninitialized

[8] Arguments can have more than one type.

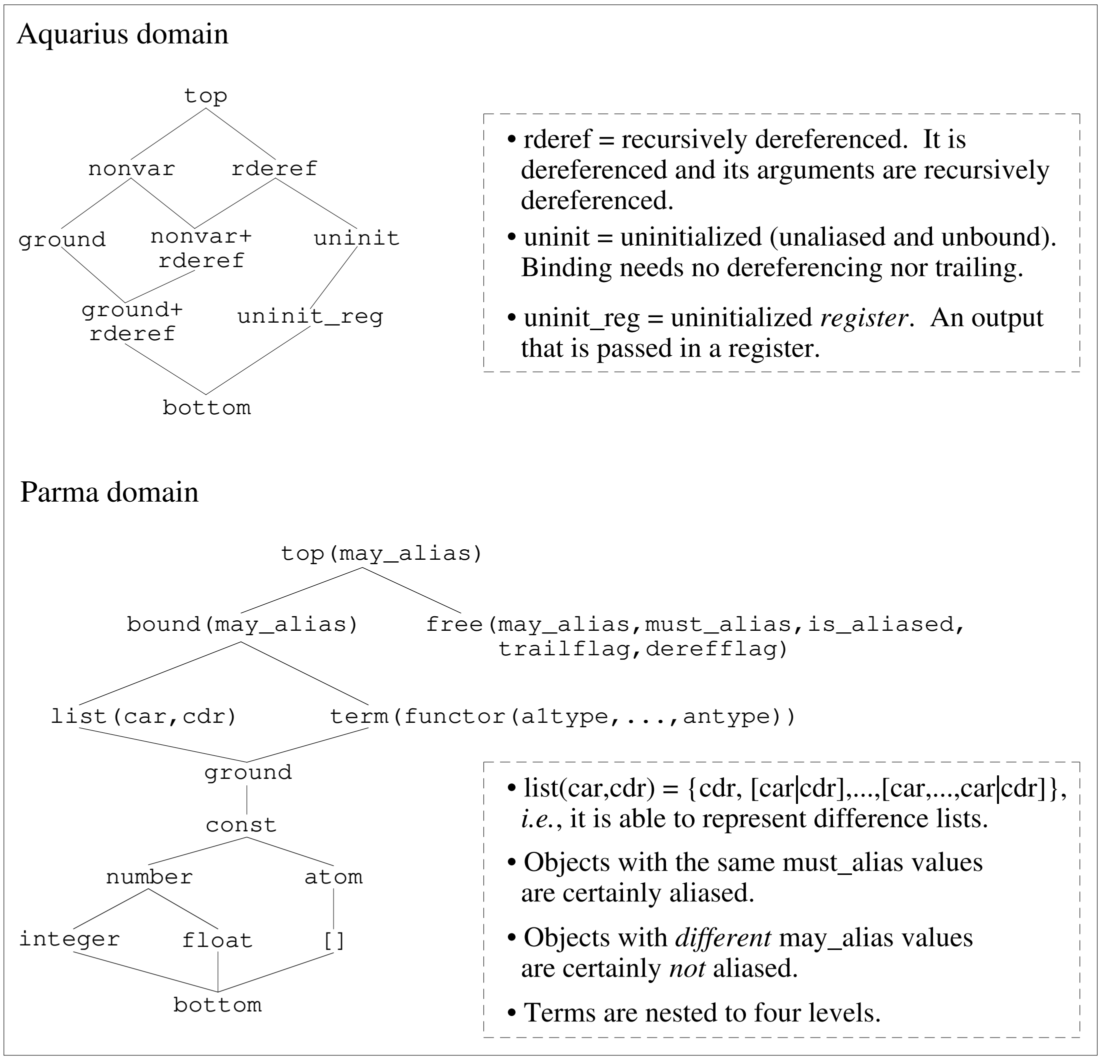

Figure 10: The Analysis Domains of the Aquarius and Parma Systems

variables by 12%. The combined benefit of two analysis features is usually not their product, since features may compete with or enhance each other. For example, uninitialized variables do not need to be trailed, and this fact will often also be determined by the trailing analysis.

Two conclusions can be drawn by studying the effects of analysis in the Parma and Aquarius systems. Analysis results in both code size reduction and speed increase. A first conclusion is that the effects of analysis on code size and speed are fundamentally different. Derived types allow both *tests* and the *code* that handles other types to be removed. Tests are usually fast. The code to handle all possible outcomes of the tests can be very large. For Aquarius the code size reduction is greater than the performance improvement. This is partly due to the lack of structure and list types in the Aquarius domain, which means that run-time type tests are still needed. For Parma the code size reduction is about the same as the performance improvement.

A second conclusion can be drawn regarding the types that are most useful for the compiler. Deriving types that have a logical meaning is not sufficient. Performance increases significantly when the analysis is able to derive types that have only an operational meaning, such as dereference (reference chains), trailing, and aliasing-related types (uninitialized variables).

2.4.6 Using Types when Compiling Unification

It is just as hard to *use* analysis in the compiler as it is to *do* analysis. Very little has been published on how to use analysis. This section shows how unification is compiled in the Aquarius system to take maximum advantage of the types known at compile-time. The code generated by the two-stream algorithm of Section 2.4.2 handles the general case when no types are known. If types are known then compiling unification becomes a large case analysis.[9] Even after common cases are factored out, the number of cases remains large. Figure 11 gives a simplified view of the top two levels of the case analysis done in Aquarius.

Table 4 gives details of the case analysis done in Aquarius for the compilation of the unification X=Y with type T. The compiler attempts to use all possible type information to simplify the code. A general unify instruction is only generated once (in oldvar_oldvar), namely when unifying two initialized variables for which nothing is known. For simplicity, the table omits the generation of dereference and trail instructions, the handling of uninitialized memory and uninitialized register variables, the updating of type information when variables are bound, the generation of pragmas, and various less important optimizations. See Section 2.4.4 for more information.

The variable T denotes the type information known at the start of the unification. The implication (T⇒ground(X)) is true if T implies that X is bound to a ground term at run-time. The conditions var(X) and (T⇒var(X)) are very different: the first tests whether X is a variable at compile-time, and the second tests whether X is a variable at run-time. The condition new(X) succeeds if X does not yet have a value, *i.e.*, for the first occurrence of X in the clause or if X is uninitialized. The condition old(X) is the negation of new(X). The function atomic_value(T,X) succeeds if T implies that X is an atomic term whose value is known at compile-time. The

[9] Compiling a goal invocation (a call) is also a large case analysis [153].

Definition of routines		
Name	Condition	Actions
unify(X,Y)	var(X),var(Y)	var_var(X,Y)
	var(X),nonvar(Y)	var_nonvar(X,Y)
	nonvar(X),var(Y)	var_nonvar(Y,X)
	nonvar(X),nonvar(Y)	nonvar_nonvar(X,Y)
nonvar_nonvar(X,Y)		For all arguments X_i, Y_i: unify(X_i,Y_i)
var_nonvar(X,Y)	T⇒new(X)	new_old(X,Y)
	T⇒ground(X)	old_old(X,Y)
	otherwise	old_old(X,Y) (with depth limiting)
var_var(X,Y)	T⇒(old(X),old(Y))	oldvar_oldvar(X,Y)
	T⇒(old(X),new(Y))	Generate store instruction
	T⇒(new(X),old(Y))	Generate store instruction
	T⇒(new(X),new(Y))	new_new(X,Y)
new_new(X,Y)		Generate store and move instructions
new_old(X,Y)	compound(Y)	write_sequence(X,Y)
	atomic(Y)	Generate store instruction
	var(Y)	var_var(X,Y)
old_old(X,Y)	compound(Y),(T⇒nonvar(X))	Test Y's type, then old_old_read(X,Y)
	atomic(Y),(T⇒nonvar(X))	old_old_read(X,Y)
	nonvar(Y),(T⇒var(X))	old_old_write(X,Y)
	compound(Y)	Generate switch, old_old_read(X,Y), old_old_write(X,Y)
	atomic(Y)	Generate unify_atomic instruction
	var(Y)	var_var(X,Y)
oldvar_oldvar(X,Y)	A=atomic_value(T,X)	unify(Y,A)
	A=atomic_value(T,Y)	unify(X,A)
	T⇒(atomic(X),atomic(Y))	Generate comparison instruction
	T⇒(var(X),nonvar(Y))	Generate store instruction
	T⇒(nonvar(X),var(Y))	Generate store instruction
	otherwise	Generate general unify instruction
old_old_write(X,Y)	compound(Y)	write_sequence(X,Y)
	atomic(Y)	Generate store instruction
old_old_read(X,Y)	compound(Y)	Test Y's functor, then for all arguments X_i, Y_i: old_old(X_i,Y_i)
	atomic(Y)	Generate comparison instruction
write_sequence(X,Y)		Generate instructions to create compound term Y in X

Table 4: The Case Analysis in the Aquarius Compilation of the Unification X=Y with Type T

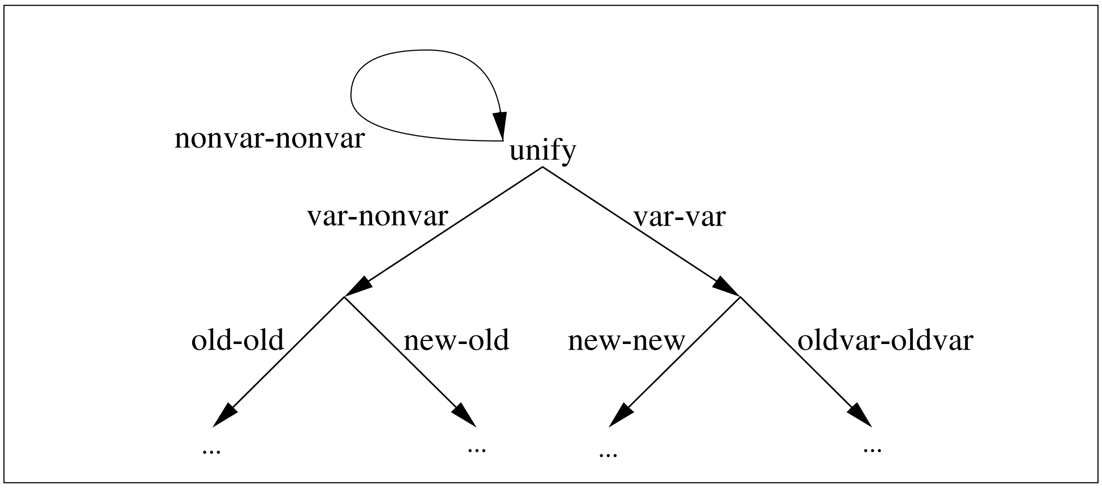

Figure 11: Case Analysis in Compiling Unification

function returns this atomic term. For example, if T is (X==a), then the function returns the atom 'a'.

The general unify instruction is expanded into code that handles inline the cases where one or both of the arguments are variables. Measurements of the dynamic behavior of unification on four real programs show that one or both of the arguments are variables about 85% of the time [63]. A subroutine call is made only if both arguments are nonvariables.

2.5 Beyond the WAM: Radically Different Execution Models

Some recent developments in Prolog implementation are based on novel models of execution very different from the WAM. The Vienna Abstract Machine (VAM) is based on partial evaluation of each call. The BinProlog system is based on the explicit passing of success continuations.

2.5.1 The Vienna Abstract Machine (VAM)

The VAM is an execution model developed by Andreas Krall at the Technische Universität Wien (Vienna, Austria) [77]. The VAM is considerably faster than the WAM. The insight of the VAM is that the WAM's separation of argument setup from argument unification is wasteful. In the WAM, all of a predicate's arguments are built *before* the predicate is called. The VAM does argument setup and argument unification at the same time. During the call the operations of argument setup and unification are combined into a single operation that does the minimal work necessary. This results in considerable savings in many cases. For example, consider the call p(X,[a,b,c],Y) to the definition p(A,_,B). The second argument [a,b,c] is not created because it is a void variable in the head of the definition. In the WAM, the second argument would be created and then ignored in the definition.

There exist two versions of the VAM: the VAM_{1p} and the VAM_{2p}. The difference is in how the argument traversal is done. In the VAM_{2p} there are two pointers. One points to the caller's arguments and one points to the definition's arguments. The operation to be performed for each argument is obtained by a two-dimensional array lookup depending on the types of the caller argument and the definition argument. This lookup operation can be made extremely fast by a technique similar to direct threaded coding, where the address of the abstract instruction is obtained by adding two offsets. In the VAM_{1p} there is a single pointer that points to compiled code representing the caller-definition pair. The code size for the VAM_{1p} is much greater than for the VAM_{2p}, since the called predicate must be compiled separately for each call. Currently, the VAM_{2p} is a practical implementation, whereas the VAM_{1p} is not because of code size explosion.

2.5.2 BinProlog

BinProlog[10] is a high performance C-based emulator developed by Paul Tarau at the Université de Moncton (Canada) [36, 136, 137]. BinProlog has two key ideas: transforming clauses to binary clauses and passing success continuations. The resulting instruction set is essentially a simplified subset of the WAM. Implementing Prolog by means of continuations is an old technique. It was used to implement Prolog on Lisp machines and in Pop-11, see for example [23, 97]. The technique has recently received a boost by Tarau's highly efficient implementation. Functional languages have more often been implemented by means of continuations. A good example is the Standard ML of New Jersey system, which uses an intermediate representation in which all continuations are explicit ("Continuation-Passing Style") [9].

The idea of BinProlog is to transform each Prolog clause into a *binary* clause, *i.e.*, a clause containing only one body goal. Predicates that are expanded inline (such as simple built-ins) are not considered as goals. The body goal is given an extra argument, which represents its success continuation, *i.e.*, the sequence of goals to be executed if the body completes successfully. This representation has two advantages. First, no environments are needed. Second, the continuations are represented at the source level. For example, the clauses:

```
p(X, X).
p(A, B) :- q(A, C), r(C, D), s(D, B).
```

are transformed into:

```
p(X, X, Cont) :- call(Cont).
p(A, B, Cont) :- q(A, C, r(C, D, s(D, B, Cont))).
```

Each predicate is given an additional argument and each clause is converted into a binary clause.

With a well-chosen data representation, the binary clause transformation can be the basis of a system that uses very little memory yet compiles and executes very quickly. The technique

[10] There is no relationship with BIM Prolog.

as currently implemented has two problems. First, the continuations are put on the heap ("long-lived" memory), hence they do not disappear on forward execution as environments would in the WAM. That is, there is no last call optimization (see Section 2.2.4) if the original clause body contains more than one goal. Second, if the first goal fails, then the creation of the continuation is an overhead that is avoided in the WAM. Both of these problems are less severe than they appear at first glance. The first problem goes away with a suitable garbage collector. A copying collector has execution time proportional to the amount of active memory. A generational mark-and-sweep collector can perform even better in practice [168]. The second problem almost never occurs in real programs.

An important potential use of the binary clause transformation is as a tool for source transformation in Prolog compilers. By making the continuations of the WAM explicit as data terms, a series of optimizing transformations becomes possible at the source level [110]. After doing the optimizations, a reverse transformation to standard clauses can be done.

3 The Systems View

The previous sections have summarized developments from the technical viewpoint, focusing on particular developments without giving further information about the systems that pioneered them.

This section concentrates on the systems themselves. It tells the stories of some of the more popular and influential systems, of the people and institutions behind them, and of the particular problems they encountered. The section is divided into two parts. Section 3.1 talks about software systems and Section 3.2 talks about hardware systems.

3.1 Software Sagas

Since the development of the WAM in 1983 there have been many software implementations of Prolog. At the end of 1993, more than fifty systems were listed in the Prolog Resource Guide [70]. The systems discussed here are MProlog, IF/Prolog, SNI-Prolog, MU-Prolog, NU-Prolog, Quintus, BIM, IBM Prolog, SEPIA, ECLiPSe, SB-Prolog, XSB, SICStus, and Aquarius.

All of these systems are substantially compatible with the Edinburgh standard. They have been released to users and used to build applications. Many have served as foundations for implementation experiments. In particular, MU-Prolog, NU-Prolog, SB-Prolog, XSB, SICStus, and Aquarius are delivered with full source code. Quintus, MProlog, IF/Prolog, and SICStus are probably the implementations that have been ported to the largest number of platforms. The most popular systems on workstations today are SICStus and Quintus. C-Prolog was also very popular at one point.

For each system are listed its most important contributions to implementation technology. These lists are not exhaustive. Most of the important "firsts" have since been incorporated into many other systems. In some cases, a contribution was developed jointly or spread too fast to identify a particular system as the pioneer. For example, almost all commercial systems support modules. Likewise, almost all commercial systems have a full-featured foreign language interface, and many of them (including Quintus, BIM, IF/Prolog, SNI-Prolog, SICStus, and ECLiPSe) allow arbitrarily nested calls between Prolog and C.

IF/Prolog, SNI-Prolog, IBM Prolog, SEPIA, ECLiPSe, and SICStus support rational tree unification. Rational trees account for term equations which express cycles. For example, the term equation X=f(X) has a solution over rational trees, but does not over finite trees [66, 68].

All of the compiled systems except MProlog and Aquarius are based on the WAM instruction set, but modified and extended to increase performance. MProlog, BIM, IBM Prolog, SEPIA, ECLiPSe, and Aquarius support mode declarations and multiple-argument indexing. The other systems do not support mode declarations. Quintus, NU-Prolog, and XSB provide some support for multiple-argument indexing, and IF/Prolog, SNI-Prolog, and SICStus do not implement it. IBM Prolog, SEPIA, ECLiPSe, and Aquarius index on some other conditions than unification, for example on arithmetic comparisons and type tests. Quintus, BIM, SEPIA, ECLiPSe, XSB, SB-Prolog, but not SICStus, compile conditionals (if-then-else) deterministically in the special case where the condition is an arithmetic comparison or a type test.

The most interesting problems of system building are related to input size. These *scalability* problems tend to occur only when one exercises a system on large inputs. They are the main obstacles on the long path between research prototype and production quality systems. For each system are listed some of the more interesting such problems.

3.1.1 MProlog

The first commercial Prolog system was MProlog.[11] MProlog was developed in Hungary starting in 1978 at NIMIGUSZI (Computer Center of the Ministry of Heavy Industries) [13, 47]. The main developer is Péter Szeredi, aided by Zsuzsa Farkas and Péter Köves. MProlog was completed at SZKI (Computer Research and Innovation Center), a computer company set up a few years before. The implementation is based on Warren's pre-WAM three-stack model of DEC-10 Prolog. The first public demonstration was in 1980 and the first sale in September 1982.

MProlog is a full-featured structure-sharing system with all Edinburgh built-ins, debugging, foreign language interface, and sophisticated I/O. It shows that structure-sharing is as efficient as structure-copying [75]. Its implementation was among the most advanced of its day. Early on, it had a native code compiler, memory recovery on forward execution (including tail recursion optimization) and support for mode declarations (including multiple-argument indexing). It had garbage collection for the symbol table and code area. It did not and does not do garbage collection for the stacks. MProlog is currently a product of IQSOFT, a company formed in 1990 from the Theoretical Lab of SZKI.

[11] M for Modular or Magyar.

3.1.2 IF/Prolog and SNI-Prolog

IF/Prolog was developed at InterFace Computer GmbH, which was founded in 1982 in Munich, Germany. Nothing has been published about the implementation of IF/Prolog. The following information is due to Christian Pichler. IF/Prolog was commercialized in 1983. The first release was an interpreter. A WAM-based compiler was released in 1985. The origin of the compiler is an early WAM compiler developed by Christian Pichler [116]. The main developers of IF/Prolog were Preben Folkjær, Christian Reisenauer, and Christian Pichler. Siemens-Nixdorf Informationssysteme AG bought the IF/Prolog sources in 1986. They ported and extended the system, which then became SNI-Prolog.

In 1990, SNI-Prolog was completely redesigned from scratch. Pichler went to Siemens-Nixdorf to help in the redesign. The main developers of SNI-Prolog are Reinhard Enders and Christian Pichler. The current system conforms to the ISO Prolog standard [122], supports constraints, has been ported to more platforms and has improved system behavior (more flexible interfaces and less memory usage). The design of the new system benefited from the fact that Siemens is one of the shareholders of ECRC. Siemens-Nixdorf bought the rights to IF/Prolog in 1993 after InterFace disappeared. They plan to integrate the best features of IF/Prolog and SNI-Prolog into a single system.

Both systems support rational tree unification. In addition, SNI-Prolog has delaying, indefinite precision rational arithmetic, and constraint solvers for boolean constraints, linear inequalities, and finite domains. It has metaterms, which allow constraint solvers to be written in the language itself (see Section 3.1.7).

Both SNI-Prolog and IF/Prolog have extensive C interoperability. With regard to interoperability they can best be compared with Quintus (see Section 3.1.4). They allow redefinition of the C and Prolog top levels and arbitrary calls between Prolog and C to any level of nesting with efficient passing of arbitrary data (including compound terms). They have configurable memory management and garbage collection of all Prolog memory areas. They are designed to interact correctly with the Unix memory system and to support signal handlers.

3.1.3 MU-Prolog and NU-Prolog

MU-Prolog and NU-Prolog were developed at Melbourne University by Lee Naish and his group [106]. Both systems do global analysis to generate delaying declarations (see Section 2.4.5). Neither system does garbage collection.

MU-Prolog is a structure-sharing interpreter. The original version (1.0) was written by John Lloyd in Pascal. Version 2.0 was written by Naish and completed in 1982. Version 2.0 supports delaying, has a basic module system and transparent database access. Performance is slightly less than C-Prolog.

NU-Prolog is a WAM-based emulator written in C primarily by Jeff Schultz and completed in 1985. It is interesting for its pioneering implementation of logical negation, quantifiers, if-then-else, and inequality, through extensions to the WAM [105]. The delay declarations ("when" declarations) are compiled into decision trees with multiple entry points. This avoids repeating already performed tests on resumption. It results in indexing on multiple arguments in practice. NU-Prolog was the basis for many implementation experiments, *e.g.*, related to parallelism [107, 114], databases [118], and programming environments [108].

3.1.4 Quintus Prolog

Quintus Prolog is probably the best-known commercial Prolog system. Its syntax and semantics have become a de facto standard, for several reasons. It is close to the Edinburgh syntax and is highly compatible with C-Prolog. It was the first widely known commercial system. Several other influential systems (*e.g.*, SICStus Prolog) were designed to be compatible with it. The pending ISO standard for Prolog [122] will most likely be close in syntax and semantics to the current behavior of Quintus.

Quintus Computer Systems was founded in 1984 in Palo Alto, California. It is currently called Quintus Corporation, and is a wholly-owned subsidiary of Intergraph Corporation. The founders of Quintus are David H. D. Warren, Lawrence Byrd, William Kornfeld, and Fernando Pereira. They were joined by David Bowen shortly thereafter, and Richard O'Keefe in 1985. Tim Lindholm was responsible for many improvements including discontiguous stacks and the semantics for self-modifying code (see below). Many other people contributed to the implementation. Quintus Prolog 1.0 first shipped in 1985.

Quintus Prolog compiles to an efficient and compact direct threaded-code representation. For portability and convenience, the emulator is written in *Progol*,[12] a macro-language which is essentially a macro-assembler for Prolog using Prolog syntax. The mode flag does not exist explicitly, but is cleverly encoded in the program counter by giving the **unify** instructions two entry points.

Quintus Prolog made several notable contributions, including those listed below.

- It is the Prolog system that generates the most compact code. Common sequences of operations are encoded as single opcodes. The code size is several times smaller than native code implementations. For example, the code generated for a given input program is about one fifth the size of that generated by the BIM compiler. The code size is between one fifth and one half that of Aquarius Prolog. The figure of one half holds only when the global analysis of Aquarius performs well [154]. For applications with large databases, compact code can become significant. The recent rapid increase in physical memory size makes reducing code size less of a priority, although there will always be applications (*e.g.*, databases and natural language) that require compact code to run well.

- It was the Prolog system that first developed a foreign language interface. Since then, it is the Prolog system that has put the most effort into making the system embeddable. It is important to be able to seamlessly integrate Prolog code with existing code. This implies a set of capabilities to make the system well-behaved and expressive. Quintus

[12] The name is a contraction of Prolog and Algol.

is able to redefine the C and Prolog top levels. It allows arbitrary calls between Prolog and C, with efficient manipulation of Prolog terms by C and vice versa. It has an open interface to the operating system that lets one redefine the low-level interfaces to memory management and I/O. It does efficient memory management, *e.g.*, it was the first system to run with discontiguous stacks. This "small footprint" version has been available since release 3.0. It carefully manages the Prolog memory area to avoid conflicts with C. It provides tools for the user including source-level debugging on compiled code and an Emacs interface.

- It was the first system to provide a clean and justified semantics for self-modifying code (assert and retract), namely the *logical view* [82]. A predicate in the process of being executed sees the definition that existed at the time of the call.
- It is the system that comes with the largest set of libraries of useful utilities. More than one hundred libraries are provided.

3.1.5 BIM Prolog (ProLog by BIM)

BIM Prolog was developed by the BIM company in Everberg, Belgium in close collaboration with the Catholic University of Louvain (KUL). The name has recently been changed to "ProLog by BIM" due to a copyright conflict with the prefix "BIM" in the United States.

Logic programming research at KUL started in the mid 1970's. Maurice Bruynooghe had developed one of the early Prolog systems, Pascal Prolog, which was used at BIM at that time. The BIM Prolog project started in October 1983. It was then called P-Prolog (P for Professional). Its execution model was originally derived from the PLM model in Warren's dissertation, but was quickly changed to the WAM.

The first version of BIM Prolog, release 0.1, was distributed in October 1984 and used in an ESPRIT project. It was a simple WAM-based compiler and emulator. Meanwhile, Quintus had released their first system. The BIM team realized that they needed to go further than emulation to match the speed of Quintus, so they decided immediately to do a native code implementation through macro-expansion of WAM instructions. In contrast to Quintus, which intended to cover all major platforms from the start, BIM initially concentrated on Sun and decided to do a really good implementation there. By 1985 the team consisted of the three main developers who are still there today: Bart Demoen, André Mariën, and Alain Callebaut. Other people have contributed to the implementation. Because BIM Prolog only ran on a few machines, it was possible for different implementation ideas to be tried over the years. For more information on the internals of BIM Prolog, see [89].

BIM Prolog made several notable contributions, including those listed below.

- It was the first WAM-based system:
  - To do native code compilation.
  - To do heap garbage collection. The Morris constant-space pointer-reversal algorithm was available in release 1.0 in 1985.

  - To do symbol table garbage collection. This is important if the system is interfaced to an external database.
  - To support mode declarations and do multiple-argument indexing, instead of indexing only on the first argument.
  - To provide modules.

  These abilities were provided earlier by DEC-10 Prolog (see Section 2.1) and MProlog (see Section 3.1.1).

- It was the first system to provide a source-level graphical debugger, an external database interface, and separate compilation.

3.1.6 IBM Prolog

IBM Prolog was developed primarily by Marc Gillet at IBM Paris. Nothing has been published about the implementation. The following information is due to Gillet and the system documentation [67]. The first version, a structure-sharing system, was written in 1983–1984 and commercialized in 1985 as VM/Prolog. A greatly rewritten and extended version was commercialized in 1989 as IBM Prolog.[13] It runs on system 370 under the VM and MVS operating systems. The system was ported to OS/2 with a 370 emulator.

The system is WAM-based and supports delaying, rational tree unification, and indefinite precision rational arithmetic. The system does global analysis at the level of a single module (see Section 2.4.5). It supports mode declarations, but may generate incorrect code if the declarations are incorrect. The system generates native 370 code and has a foreign language interface.

3.1.7 SEPIA and ECLiPSe

ECRC (European Computer-Industry Research Centre) was created in Munich, Germany in 1984 jointly by three companies: ICL (UK), Bull (France), and Siemens (Germany). ECRC has done research in sequential and parallel Prolog implementation, in both software and hardware. See Section 3.2.2 for a discussion of the hardware work. The constraint language CHIP was built at ECRC (see Section 2.3.1).

Several Prolog systems were built at ECRC. An early system is ECRC-Prolog (1984–1986), a Prolog-to-C compiler for an enhanced MU-Prolog. At the time, ECRC-Prolog had the fastest implementation of delaying. The next system, SEPIA (Standard ECRC Prolog Integrating Advanced Features), first released in 1988, was a major improvement [93]. Other systems are Opium [44], an extensible debugging environment, and MegaLog [15], a WAM-based system with extensions to manage databases (*e.g.*, persistence). The most recent system, ECLiPSe (ECRC Common Logic Programming System) [45, 95], integrates the facilities of SEPIA, MegaLog, CHIP, and Opium. The system supports rational tree unification and indefinite

[13] Curiously, both systems are written mostly in assembly code, several hundred thousand lines worth.

precision rational arithmetic. It provides libraries that implement constraint solvers for atomic finite domains and linear inequalities.

ECLiPSe is a WAM-based emulator with extensive support for delaying [95]. This makes it easy to write constraint solvers in the language itself. ECLiPSe supports this with two concepts: *metaterms* and *suspensions*. A metaterm is a variable with a set of user-defined attributes. The set of attributes is similar to a Lisp property list. A suspension is a closure. It is an opaque data type at the Prolog level. A goal can be delayed explicitly by making it into a suspension and inserting it into a list of delayed goals. The list is stored as an attribute of a variable. When the variable is unified, an event handler is invoked. The handler is free to manipulate the suspended goals in any way. Through metaterms, the wakeup order of suspended goals can be programmed by the user.

The ECLiPSe compiler is incremental and compilation time is probably the lowest of any major system. The debugger uses compiled code supplemented with debugging instructions. Because of its fast compilation, ECLiPSe has no need of an interpreter. ECLiPSe (and SEPIA before it) uses two-word (64-bit) data items, with a 32-bit tag and a 32-bit value field. This allows more flexibility in tag assignment and full pointers can be stored directly in the value field. It also makes for a more straightforward C interface.

3.1.8 SB-Prolog and XSB

SB-Prolog is a WAM-based emulator developed by a group led by David Scott Warren at SUNY (State University of New York) in Stony Brook. The compiler was written by Saumya Debray and the system was bootstrapped with C-Prolog. After several years of development, SB-Prolog was made available by Debray from Arizona in 1986. Because it was free and portable, it became quite popular. Neither it nor XSB does garbage collection. The worst problem regarding portability was the use of the BSD Unix *syscall* system call which supports arbitrary system calls through a single interface.

SB-Prolog was the basis for much exploration related to language and implementation (*e.g.*, [37]): backtrackable assert, existential variables in asserted clauses, memoizing evaluation, register allocation, mode and type inferencing (see Section 2.4.5), module systems, and compilation.

The most recent system, XSB, is SB-Prolog extended with memoization (tabling) and HiLog syntax [119]. The resulting engine is the SLG-WAM (see Section 2.3.3). XSB 1.3 implements the SLG-WAM for *modularly stratified* programs, *i.e.*, for programs that do not dynamically have recursion through negation.

3.1.9 SICStus Prolog

SICStus Prolog[14] was developed at SICS (Swedish Institute of Computer Science) near Stockholm, Sweden. SICS is a private foundation founded in late 1985 which conducts research in many areas of computer science. It is sponsored in part by the Swedish government and in

[14] The name is a pun on Quintus.

part by private companies. The guiding force and main developer of SICStus is Mats Carlsson. Many other people have been part of the development team and have made significant contributions.

In 1993, SICStus Prolog was probably the most popular high performance Prolog system running on workstations. SICStus is cheap, robust, fast, and highly compatible with the "Edinburgh standard". It has been ported to many machines. It has flexible coroutining, rational tree unification, indefinite precision integer arithmetic, and a boolean constraint solver.

The first version of SICStus Prolog, release 0.3, was distributed in 1986. SICStus became popular with the 0.5 release in 1987. Originally, SICStus was an emulated system written in C. MC680X0 and SPARC native code versions were developed in 1988 and 1991. The current version, release 2.1, has been available since late 1991.

SICStus is the first system to do path compression ("variable shunting") of dereference chains during garbage collection [120]. The parts of a dereference chain in the same choice point segment are removed. This lets the garbage collector recover more memory. This is essential for Prologs that have *freeze* or similar coroutining programming constructs [24], since the intermediate variables in a dereference chain may contain large frozen goals that can be recovered.

Among the scalability problems encountered during the development of SICStus are those listed below.

- Interface with malloc/free, the Unix memory allocation library. SICS wrote their own version of the malloc/free library that better handles the allocation done by their system. Increasing the size of system areas is done by calling realloc.
- Native code limitations. A problem for large programs is that the offsets in machine instructions have a limited size. For example, the SPARC's load and store instructions use a register+displacement addressing mode with a displacement limited to 12 bits. Other native code systems (*e.g.*, IBM Prolog) have run into the same problem.
- The space versus time trade-off. Native code implementations have larger code size than emulated implementations. This difference can be quite significant: a factor of five or more. For large programs, *e.g.*, natural language parsers with large databases, having compact code can mean the difference between a program that runs and one that thrashes. SICStus minimizes the size of its generated native code by calling little-used operations as subroutines rather than putting them inline. For example, the dereference operation is inlined only for a predicate's first argument.

3.1.10 Aquarius Prolog

Aquarius Prolog was originally developed in the context of the Aquarius project at U.C. Berkeley as the compiler for the VLSI-BAM processor [153] (See Section 3.2.3 for the hardware side of the story). After our relationship with the hardware side of the project ended in the spring of

1991, Ralph Haygood (the main developer of the back-end, run-time system, and built-ins) and I decided to continue part-time work on the software so that it could be released to the general public [58, 70]. We were joined by Tom Getzinger at USC. The system achieved 1.1 MLIPS on a SPARCstation 1+ in February 1991. It first successfully compiled itself in February 1992. It was completed and released as Aquarius Prolog 1.0 in April 1993.

Aquarius Prolog made several contributions, including those listed below.

- It is the first system to compile to native code without a WAM-like intermediate stage. It compiles first to BAM code (see Section 2.4.4), and then macro-expands to native code.
- It is the first well-documented system to do global analysis. See Section 2.4.5 for more information on the analyzer and Sections 2.4.4 and 2.4.6 for information on how the analyzer is used to improve code generation. Type and mode declarations are supported. They are used to supplement the information generated by analysis. The system may generate incorrect code if the declarations are incorrect.
- It is the first system in which most built-ins are written in Prolog with little or no performance penalty. A technique called *entry specialization* replaces built-ins by more specialized entry points depending on argument types known at compile-time.
- It is the first system to generate code which rivals the performance of an optimizing C compiler on a nontrivial class of programs [154].

The main disadvantage of Aquarius in its current state is the time of compilation. This has little to do with the sophistication of the optimizations performed, but is due primarily to the naive representation of types in the compiler. The representation was chosen for ease of development, not speed. It is user-readable and new types can be added easily.

Among the scalability problems encountered during the development of Aquarius are those listed below.

- Garbage collection with uninitialized variables. Before they are bound, uninitialized variables contain unpredictable information. The garbage collector must be able to handle this correctly. In Aquarius, the garbage collector follows all pointers, including uninitialized variables. Hence, it does not recover all the memory it could. As far as we can tell, following all pointers does not adversely affect the system in practice. All programs we have tried, including very long-running ones, have stable memory sizes.
- Interaction of memory management with malloc. The observed behavior was that the system crashed because some stdio routines called malloc, which returned memory blocks inside the Prolog heap. Calling the default malloc is incompatible with our memory manager because it expands memory size if more memory is needed. After such an expansion, the malloc-allocated memory is inside a Prolog stack. On some platforms there is a routine, f_prealloc, that ensures that stdio routines do all of their allocation at startup. This does not work for all platforms. Our final solution uses a

public domain malloc/free package (written by Michael Schroeder) that is given its own region of memory upon startup.

- During the DECstation port, a bug was found in the MIPS assembler provided with the system. The assembler manual states that registers t0-t9 ($8-$15, $24-$25) are not preserved across procedure calls. The MIPS instruction scheduler apparently assumes that they need not be saved even across branches, but this is not documented. We solved the problem with the directive ".set nobopt", which prevents the scheduler from moving an instruction at a branch destination into the delay slot. This results in slightly lower performance. The problem went undiscovered until we made the system self-compiling.

3.2 Hardware Histories

Starting in the early 1980's there was interest in building hardware architectures optimized for Prolog. Two events catalyzed this interest: the start of the Japanese Fifth Generation Project in 1982 and the development of the WAM in 1983. In 1984 Tick and Warren proposed a paper design of a microcoded WAM that was influential for these developments [142]. At first, the specialized architectures were mostly microcoded implementations of the WAM (*e.g.*, the PLM and the PSI-II). Later architectures (*e.g.*, the KCM and the VLSI-BAM) modified the WAM design.

Some of the most important efforts are the PSI and CHI machine projects primarily at ICOT, the KCM project at ECRC, the POPE project at the GMD in Berlin, the Pegasus project at Mitsubishi, the Aquarius project at U.C. Berkeley (with its commercial offspring, Xenologic Inc.), and the IPP project at Hitachi. All these groups built working systems.

The POPE (Parallel Operating Prolog Engine) design is based on extracting fine-grain parallelism in WAM instructions [11]. The POPE was built in Berlin at the GMD (Gesellschaft für Mathematik und Datenverarbeitung) in the late 1980's. The machine is a ring of up to seven tightly coupled sequential Prolog processors. Parallelism is achieved at each call by interleaving argument setup with head unification. The head unification is done on the next machine in the ring. In this fashion, the machine is automatically load balanced and achieves a speedup of up to seven.

The IPP (Integrated Prolog Processor) [78] is a Hitachi ECL superminicomputer of cycle time 23ns ($\approx$ 43.5 MHz) with 3% added hardware support for Prolog. The IPP was built in the late 1980's. The support comprises an increased microcode memory of 2 KW and tag manipulation hardware. The IPP implements a microcoded WAM instruction set modified to reduce pipeline bubbles and memory references. Its performance is comparable to Aquarius Prolog on a SPARCstation 1+ (see Table 7).

In the late 1980's came the first efforts to build RISC processors for Prolog. These include Pegasus, LIBRA [101], and Carmel-2 [56] (the latter supports Flat Concurrent Prolog). For lack of appropriate compiler technology, these systems executed macro-expanded WAM code or hand-coded assembly code.

The Pegasus project began in 1986 at Mitsubishi. They designed and fabricated three single-chip RISC microprocessors in the period 1987–1990 [125]. The first two chips were fabricated in October 1987 and August 1988 [123, 124, 166]. The first chip contains 80,000 transistors in an area of 10 mm square (≈ 100 mm^2) with 2μ CMOS. The second chip contains 80,000 transistors in an area of 9.7 mm square (≈ 94 mm^2) with 1.5μ CMOS. The third and last chip, Pegasus-II, was fabricated in September 1990 and at 10 MHz achieves a performance comparable to the KCM (see Table 7). The third chip contains 144,000 transistors in an area of 9.3 mm square (≈ 86.5 mm^2) with 1.2μ CMOS. The last two chips ran the Warren benchmarks a few months after fabrication. The chips have a bank of shadow registers to improve the performance of shallow backtracking. They provide support for tagging and dereferencing with ideas similar to those of the VLSI-BAM and KCM. Pegasus-II has two interesting features. It provides support for context dependent execution (which the designers call "dynamic execution switching") of read/write mode in unification (see Section 3.2.2). It provides compound instructions (pop & jump, push & jump, pop & move, push & move) to exploit data path parallelism.

By 1990, the appropriate compiler technology was developed on two RISC machines. The VLSI-BAM, a special-purpose processor, ran Aquarius Prolog [63]. The MIPS R3000, a general-purpose processor, ran Parma [139]. The VLSI-BAM has a modest amount of architectural support for Prolog (10.6% of active chip area). Parma achieves a somewhat greater performance on a general-purpose processor at the same clock rate (see Table 7). The major difference between the two systems is that Parma has a bigger type domain in its analysis (see Figure 10 and Section 2.4.5).

The experience with Aquarius and Parma proves that there is nothing inherent in the Prolog language that prevents it from being implemented with execution speed comparable to that of imperative languages. Comparing the two systems shows that improved analysis lessens the need for architectural support.

Since 1990 the main interest in special-purpose architectures has been as experiments to guide future general-purpose designs. The interest in building special-purpose architectures for their own sake has died down. Better compilation techniques and increasingly faster general-purpose machines have taken the wind out of its sails (see also Section 5.1). This parallels the history of Lisp machines.

The rest of this section examines three projects in more detail: the PSI machine project (ICOT/Mitsubishi/Oki), the KCM project (ECRC), and the Aquarius project (U.C. Berkeley). I have chosen these projects for two reasons: they show clearly how system performance improved as Prolog was better understood, and detailed measurements were performed on them.

3.2.1 ICOT and the PSI Machines

The FGCS (Fifth Generation Computer System) project at ICOT (Japanese Institute for New Generation Technology) has designed and built a large number of sequential and parallel Prolog machines [134, 147]. Both in manpower and machines, the FGCS was the largest architecture project in the logic programming community. Two series of sequential machines were built: the PSI (Personal Sequential Inference) machines (PSI-I, PSI-II, and PSI-III) and the CHI (Cooperative High performance sequential Inference) machines (CHI-I and CHI-II) [54]. I will limit the discussion to the PSI machines, which were the most popular. All the PSI machines are horizontally microprogrammed and have 40-bit data words with 8-bit tag and 32-bit value fields.

The PSI-I was developed before the WAM [133]. After the development of the WAM this was followed by two WAM-based machines, the PSI-II and PSI-III. The three models were manufactured by Mitsubishi and Oki, and commercialized by Mitsubishi in Japan. Several multiprocessors were built at ICOT with these processors as their sequential processing elements. The PSI-II is the PE of the Multi-PSI/v2 and the PSI-III is the PE of the PIM/m.

The PSI-I was designed as a personal workstation for logic programming. It was first operational in December 1983 at a clock rate of 5 MHz. It runs ESP (Extended Sequential Prolog), a Prolog extended with object-oriented features. More than 100 machines were shipped. The first ESP implementation was an interpreter written in microcode (not a WAM). A WAM emulator was later written for the PSI-I and ran twice as fast. The main advantage of the PSI-I was not speed, but memory. It had 80 MB of physical memory, a huge amount in its day.

The PSI-II was first operational in December 1986 [109]. More than 500 PSI-II machines were shipped from 1987 until 1990 and delivered primarily to ICOT. Its clock was originally 5 MHz but was quickly upgraded to 6.45 MHz. At the higher clock, its average performance is 3 to 4 times that of the interpreted PSI-I.

The PSI-III was first operational near the end of 1990. More than 200 PSI-III machines have been shipped. It is binary compatible with the PSI-II and has almost the same architecture with a clock rate of 15 MHz. The microcode was ported from the PSI-II by an automatic translator. Its average performance is 2 to 3 times that of the upgraded PSI-II.

3.2.2 ECRC and the KCM

The architecture work at ECRC culminated in the KCM (Knowledge Crunching Machine) project, which started in 1987 [14, 112]. The KCM was probably the most sophisticated Prolog machine of the late 1980's. It had an innovative architecture and significant compiler design was done for it. It was preceded by two years of preliminary studies (the ICM, ICM3, and ICM4 architectures) [111, 165]. The KCM was built by Siemens. The first prototypes were operational in July 1988 and ran at a clock speed of 12.5 MHz. About 50 machines were delivered to ECRC and its member companies [141].

The KCM is a single user, single tasking, dedicated coprocessor for Prolog, used as a back-end to a Unix workstation. It is a tagged general-purpose design with support for Prolog, and hence is not limited to Prolog. It uses 64-bit data words, with a 32-bit tag and a 32-bit value field.

The KCM's instruction set consists of two parts: a general-purpose RISC part and a microcoded WAM-like part. Prolog compilation for the KCM is still WAM-like, but the instructions have

Feature	Benefit (%)
multiway tag branch (MWAC)	23.1
context dependent execution (flags)	11.4
dereferencing support	10.0
trail support	7.2
load term	5.7
fast choice point creation/restoration	2.3
Total	59.7

Table 5: The Benefits of Prolog-Specific Features in the KCM

evolved greatly from Warren's original design (see [92, 112]). The KCM supports the delayed creation of choice points. The KCM runs KCM-SEPIA, a large subset of SEPIA that was ported to it (see Section 3.1.7).

The Prolog support on the KCM improves its performance by a factor of ≈ 1.60 [112, 141]. The architectural features and their effects on performance are given in Table 5.

The MWAC (Multi-Way Address Calculator) is a functional unit that does a 16-way microcode branch depending on the types of two arguments. It calculates the target address during the last step of dereferencing. The MWAC is used in the execution of all unification operations. It is similar to the partial unification unit of the LIBRA [101].

Context dependent execution uses flags in addition to the opcode during instruction decoding. Three flags are used: read/write mode for unification, choicepoint/nochoicepoint for delayed choice point creation, and deep/shallow for fast fail in shallow backtracking.

3.2.3 The Aquarius Project: The PLM and the VLSI-BAM

In 1983, Alvin Despain and Yale Patt at U.C. Berkeley initiated the Aquarius project. Its main goal was to design high performance computer systems with large symbolic and numeric components. The project continued at Berkeley until 1991.[15] They decided to focus on Prolog architectures, being inspired by the FGCS project and seduced by the mathematical simplicity of Prolog. As soon as Warren presented the WAM at Berkeley, Despain turned the project to focus on hardware support for the WAM. He proposed that I write a compiler for their architecture, the PLM. The compiler was completed and the report was delivered to the university on August 22, 1984.[16] This was the first published WAM compiler [148].[17]

A whole series of sequential and parallel Prolog architecture designs came out of Aquarius. The sequential designs that were built are:

[15] Despain is continuing this work at USC's Advanced Computer Architecture Laboratory.

[16] The exact day of my flight back to Belgium.

[17] In January 1991, I toured several German universities and research institutes to talk about Aquarius Prolog. At ECRC a scientist from East Berlin came to me after the talk. He explained that they had *typed in* the source code of the PLM compiler from the appendix of the report.

- The PLM [42, 43] (1983–87). The Programmed Logic Machine.[18] This is a microcoded WAM.
- The VLSI-PLM [128, 129] (1985–89). This is a single-chip implementation of the PLM.
- The Xenologic X-1. This is a commercial version of the PLM, designed as a coprocessor for the Sun-3. Due to weaknesses in its system software, this system was not commercially successful.
- The VLSI-BAM [63] (1988–91). The VLSI Berkeley Abstract Machine. This is a single-chip RISC processor with extensions for Prolog.

The PLM was wire-wrapped and ran a few small programs in 1985. The Xenologic X-1 has been running at 10 MHz since 1987. The VLSI-PLM was fabricated and ran all benchmarks at 10 MHz in June 1989. The VLSI-BAM was designed to run at 30 MHz. It contains 110,000 transistors in an active area of 91 mm^2 with 1.2μ CMOS. It was fabricated in November 1990 and ran most benchmarks of [154, 156] at 20 or 25 MHz on its custom cache board in November 1991.

The core of the VLSI-BAM is a RISC in the classic sense: it is a 32-bit pipelined load-store architecture with single-cycle instructions, 32 registers, and branch delay slots. The processor is extended with support for Prolog and for multiprocessing, which together form 10.6% of the active chip area and improve Prolog performance by a factor of ≈ 1.70 [63]. The VLSI-BAM executes the same Prolog program in one third the cycles of the VLSI-PLM, a gain due to improved compilation.

The primary purpose in building the VLSI-BAM was not to achieve the best absolute performance–a university project cannot compete in performance with industry–but to quantify the usefulness of its architectural features. The intention was that the results could be used to guide the design of other machines.

The Prolog support takes the form of six architectural features and new instructions using them. The architectural features, their performance benefits, and their active chip area are given in Table 6. The benefit figures cannot be directly added up because the effects of the architectural features are not independent.

Except for dereference, the instructions are all single-cycle. There are two- and three-way tagged branches to support unification and a conditional push to support trailing. The instructions for data structure creation (write-mode unification) were derived automatically using constrained exhaustive search [64]. VLSI-BAM measurements [63] show that with advanced compilation techniques, multiway branches for general unification are effective only up to a three-way branch.[19] Multiple-cycle (primarily dereference) and conditional instructions are implemented by logic to insert or remove opcodes in the pipeline. The opcode pipe has space

[18] The name correspondence with the PLM model in Warren's dissertation [159] is a coincidence.

[19] This does not contradict the measurements of the KCM's MWAC since the latter is used for all unification operations, not just general unification.

Feature	Benefit (%)	Area (%)
fast tag logic (tagged branching)	18.9	1.6
double-word memory port	17.1	1.9
tag and segment mapping	10.3	4.8
multi-cycle/conditional	9.1	0.1
tagged-immediates	7.9	2.2
arithmetic overflow detect	1.4	<0.1
Total	70.1	10.6

Table 6: The Benefits and Chip Area of Prolog-Specific Features in the VLSI-BAM

for both user instructions and added “internal” instructions. The double-word memory port (with double bandwidth to cache) improves general-purpose memory operations as well as choice point creation and restoration speed.

4 The Evolution of Performance

Due to faster machines and improved compilation technology, the performance of Prolog has increased about two orders of magnitude since DEC-10 Prolog. Table 7 gives the execution time ratios, relative to DEC-10 Prolog, of a set of representative systems running the five Warren benchmarks [159]. For the reasons given below, the numbers in Table 7 do *not* generalize to large programs. *They should be seen only as indicating trends.*

Table 7 is split into two parts. The first five rows show the performance of specialized hardware. The following rows show general-purpose hardware. For the first five rows and for DEC-10 Prolog the year in which the systems were first running is given. For the other systems the architecture is given. Results for the benchmarks nreverse, qsort, deriv, serialise, and query are given in columns N, Q, D, S, and R, respectively. Table 8 gives their absolute execution times on DEC-10 Prolog. The benchmarks were timed with a failure-driven loop. The deriv benchmark is the sum of the four benchmarks times10, log10, divide10, and ops8. The last column of Table 7 gives the harmonic mean of the speedup ratios.

Performance is one of the few quantifiable measures of a system. Many other measures are just as important, but are hard to quantify. For example, it is difficult to assign numbers to embeddability, robustness, debuggability, portability, and the usefulness of the available built-in operations. The overall quality of a system depends on how well it meets the needs of the task at hand. A rough indication of overall quality can be obtained from the software sagas presented earlier. This should be refined for a particular system by using it to solve a relevant problem.

The systems marked by † are research systems. The systems on general-purpose hardware that are marked by ‡ are native code systems. The others are emulated. The numbers for XSB 1.3 are within 10% of SB-Prolog 3.1. Many of the systems generate better code if the program has mode declarations. For example, IBM Prolog is about 1.5 times faster with mode declarations.

System	Machine (Year)	Clock	Benchmark					
		(MHz)	N	Q	D	S	R	mean
†PLM compiler [148]	PLM [43] (1985)	10.	19	12	9	12	8	11
ESP	PSI-II (1986)	6.45	41	25	12	18	10	16
KCM-SEPIA [112]	KCM (1989)	12.5	83	57	37	33	15	32
†Pegasus compiler [125]	Pegasus-II (1990)	10.	91	69	39	40	19	39
†Aquarius [63]	VLSI-BAM (1991)	20.	270	260	75	57	32	72
	Machine (Architecture)							
‡DEC-10 Prolog [159]	DEC-10 (1977)		1	1	1	1	1	1
XSB 1.3	SPARCstation 1+ (SPARC)	25	7	4	2	4	3	3
Quintus 2.0 [63]	Sun 3/60 (MC68020)	20	11	4	3	4	3	4
‡MProlog 2.3	IBM PC clone (386)	33	13	6	5	5	2	5
ECLiPSe 3.3.7	SPARCstation 1+ (SPARC)	25	11	6	4	6	3	5
NU-Prolog 1.5.38	SPARCstation 1+ (SPARC)	25	22	7	5	7	2	5
SICStus 2.1	DECstation 5000/200 (R3000)	25	37	16	10	10	5	10
Quintus 2.5 [154]	SPARCstation 1+ (SPARC)	25	33	16	9	13	8	12
‡BIM 3.1 beta [154]	SPARCstation 1+ (SPARC)	25	34	21	8	16	8	13
‡SICStus 2.1	SPARCstation 1+ (SPARC)	25	39	26	15	20	8	17
‡†Aquarius [154]	SPARCstation 1+ (SPARC)	25	120	140	28	25	12	29
‡IBM Prolog	ES/9000 Model 9021 (370)		120	59	74	69	33	60
‡Aquarius 1.0	DECstation 5000/200 (R3000)	25	180	210	63	44	46	71
‡†Parma [140]	MIPS R3230 (R3000)	25	330	350	130	170	59	140

Table 7: The Evolution of Prolog Performance

MProlog 2.3 is about 1.2 times faster with mode and indexing declarations. On the same PC clone, emulated SICStus 2.1 is 1.5 times slower than MProlog 2.3 and five times slower than native SICStus 2.1 on a SPARCstation 1+.

The Warren benchmarks were chosen because reliable performance numbers for them are available for many machines. They are *not* a good measure of the performance of real programs. A more realistic benchmark set that subsumes the Warren benchmarks is used in [140, 154] and may be obtained from [156].

The Warren benchmarks are small and many systems have been optimized to execute them fast. The speedup for nreverse is greater than average because more effort has been done to optimize it. The speedup for query is less than average because it is dominated by integer multiplication and division. Due to limitations in their analysis domains (see Section 2.4.5), Aquarius and Parma have lower performance for large programs unless the programs are tuned. Large programs are more likely to spend most of their time doing built-in operations, which are a fixed cost since they are usually implemented in a lower level language.

In older publications, a common unit in Prolog performance is the LIPS, or Logical Inferences Per Second, *i.e.*, the number of goal invocations or procedure calls per second. Because the amount of work done by a procedure call is not constant, the LIPS number is an unreliable indicator of system performance and is not given. By convention, published LIPS numbers are measured for nreverse, which reverses a 30-element list in 496 logical inferences.

Benchmark	N	Q	D	S	R
Time (ms)	53.7	75.0	10.1	40.2	185.0

Table 8: The Execution Times of the Warren Benchmarks on DEC-10 Prolog

It is difficult to compare the performance of two systems unless they are running on identical hardware. For example, the same system can vary greatly in speed even when running the same CPU-bound program on two machines with the same processor, clock speed, and cache size. This could be the case because the write buffers are of different sizes. Among the machine-related factors that affect performance are clock speed, but also the memory system (*i.e.*, cache and virtual memory structure, memory size and bandwidth), the operating system (*e.g.*, speed of I/O and context switching overhead), the data path (*e.g.*, pipeline structure, multiple functional units, out-of-order and superscalar execution), and the implementation of various primitive operations (*e.g.*, multiplication can vary an order of magnitude in speed even on systems with the same clock). An important difference between the SPARC-based and R3000-based systems in Table 7 is that the latter have a faster memory system.

5 Future Paths in Logic Programming Implementation

This section gives a personal view of the trends in sequential logic programming implementation. It is important to distinguish three levels of evolution. First, the low level trends. What will be the basic improvements in implementation technology for Prolog and related languages? Second, the high level trends. What will be the new tools, new languages, and programming paradigms? Finally, what will be the relation between Prolog and the mainstream computing community? See [48] for an early but still useful discussion of these issues.

5.1 Low Level Trends

There are many ways in which Prolog implementation technology can be improved. Here are some of the important ones, given in order of increasing difficulty:

- **Overlap with mainstream compiler technology**. As Prolog compilers approach imperative language performance, the standard optimizations of imperative language compilers (global register allocation, code motion, instruction reordering, and so forth) become important. Some of these are being implemented in current systems [38]. One approach is to compile to C. This shortens development time, gains portability, and (to a lesser degree) takes advantage of what the C compiler does (*e.g.*, register allocation). This approach has traditionally had a performance loss over native code of a factor of two to three. This will change in the future. For example, because of its first-class labels and global register declarations, the recently released GNU C 2.X compiler has a smaller performance loss than other C compilers [36, 57]. Recent work shows that the overhead of compilation to C can be reduced to less than 30%, while keeping the system portable [99]. C is becoming a portable assembly language.

- **Type inference and operational types**. When writing a program, a programmer often has definite intentions about the types of predicate arguments. This includes information on the structure of compound terms (*e.g.*, recursive types such as lists and trees) and on operational types (see Section 2.4.5). For analysis to work well with large programs as well as small benchmarks, the analysis domain has to represent this information, to track variable dependencies, and to correctly handle built-in predicates. Objects whose type is known at compile-time can be represented *unboxed*, *i.e.*, accessible without tagging or other overhead. Current systems only unbox variables (see the discussion on uninitialized variables in Section 2.4.4) and numbers within arithmetic expressions.

- **Determinism extraction**. Often, a deterministic user-defined predicate is used to select a clause. This is currently compiled by creating a choice point, executing the predicate, and backtracking if it fails. It would be more efficient to compile such a predicate as a boolean function and to do a conditional jump on its result.

- **Multiple specialization**. Different calls to the same predicate frequently have different types in the same argument. The predicate will run faster if it is compiled separately for each pattern of calling types. As a first step, multiple specialization can be enabled by a directive. Profiling could supply the directives. Measurements show that adding these directives is often fruitful. For example, in the chat_parser benchmark the inner loop is a two-clause predicate, *terminal/5*, that is called 22 times. Making 22 copies and recompiling with analysis under Aquarius Prolog results in a 16% performance improvement. In programs with tighter inner loops the performance improvement can be much greater. For example, the SEND+MORE=MONEY puzzle shows a tenfold speedup [155].

- **Compile-time garbage collection**. Prolog creates three kinds of data objects in memory: choice points, compound data terms, and environments. When a data object becomes inaccessible, a new object can often reuse part of the old one. For example, a program that uses an array can destructively update the array if it is unaliased (see Section 2.4.4). Unaliased arrays are called *single-threaded*. Recent developments indicate that it is more practical to enforce single-threadedness syntactically (through source transformation) than to use an analyzer-compiler combination [65]. See for example the use of monads in functional programming [158] and the Extended Definite Clause Grammar notation of [151, 153] which is extended in [7].

- **Dynamic to static conversion**. All data in Prolog is allocated dynamically, *i.e.*, at run-time. It is accessed through tagged pointers. Often, it is necessary to follow a chain of pointers to find the data. Since CPU speed is increasing faster than memory speed [59], the overhead of memory access will become relatively more important in the future. The software and hardware approaches to speed up memory access are complementary:

  - A future compiler could statically allocate part of the dynamically allocated data to reduce access time and improve locality. This requires analysis to determine the evolution of aliasing during program execution. For example, objects that are unaliased, that exist only in one copy at any given time, and whose size is known can be allocated statically.

- A future architecture could be designed to tolerate memory latency. If the architecture could follow one level of tagged pointer in zero time, then the execution model of Prolog could be changed drastically and would run faster. Two techniques that help are starting to appear in existing architectures: asynchronous loads (decoupling the load request and arrival of the result) and multithreading (fast switching between register sets). These are useful for all languages, not just Prolog.

5.2 High Level Trends

The development of both Prolog and more advanced logic languages are active areas of research. In recent years, the implementation of logic programming systems has continued in two main directions.

- **Further development of Prolog**.
  - Software engineering aspects: this development has been mostly in the area of extended usability of the system rather than performance. For example, many systems including Quintus, SICStus, BIM, and ECLiPSe, have a foreign language interface that allows arbitrary calls between Prolog and C, to any level of nesting. Debugging has improved, and several systems now have source-level debuggers and profilers [51]. Many systems have eased the strict control flow by including coroutining facilities (such as *freeze*). There is an ISO standard for Prolog that is essentially complete [122].
  - "Cleaner" Prologs: these languages aim to keep the ideas and functionality of Prolog, but to replace the "dirty" operational features (such as assert, var, and cut) by clean declarative ones. It is not yet obvious whether this is possible without losing expressivity and performance. This group includes the MU-Prolog and NU-Prolog family [104] (see Section 3.1.3), xpProlog [83], and the Gödel language [62].
- **Other logic programming languages**. These can be roughly subdivided into three main families. The families overlap, but the division is still useful.
  - Concurrent languages: these languages include the committed-choice languages [126] (*e.g.*, Parlog, FGHC, and FCP) and languages based on the "Andorra principle" [33, 55] (an elegant synthesis of Prolog and committed-choice languages).
  - Constraint languages: a language that does incremental global constraint solving in a particular domain is called a *constraint language*. These languages come in two flavors. The general-purpose languages (such as Prolog, Trilogy [157], and LIFE [5]) provide domains that are useful for most programming tasks. For example, unification in Prolog handles equality constraints over finite trees. The

special-purpose languages (such as Prolog III, CLP(R), and CHIP) provide specialized domains that are useful for particular kinds of problems. For example, linear arithmetic inequalities on real numbers and membership in finite domains. These languages allow practical solutions to many problems previously considered intractable such as optimization problems with large search spaces.

- "Synthesis" languages: there are now serious attempts to make syntheses of different styles of programming [53]. For example, λProlog [100] and languages based on narrowing are syntheses of logic and functional programming, LIFE is a synthesis of logic, functional, and object-oriented programming, and AKL [55] is a synthesis of concurrent and constraint languages [121]. An important principle is that a synthesis must start from a simple theoretical foundation.

5.3 Prolog and the Mainstream

As measured by the number of users, commercial systems, and practical applications, Prolog is by far the most successful logic programming language. Its closest competitors are surely the special-purpose constraint languages. But it is true that logic programming in particular and declarative programming in general remain outside of the mainstream of computing. Two important factors that hinder the widespread acceptance of Prolog are:

- **Compatibility**. Existing code works and investment in it is large. Therefore people will not easily abandon it for new technology. Therefore a crucial condition for acceptance is that Prolog systems be embeddable. This problem has been solved to varying degrees by commercial vendors (see Section 3.1.4).
- **Public perception**. To the commercial computing community, the terms "Prolog" and "logic programming" are at best perceived as useful in an academic or research setting, but not useful for industry. This image is not based on any rational deduction. Changing the image requires both marketing and application development.

The ideas of logic programming will continue to be used in those application domains for which it is particularly suited. This includes domains in which program complexity is beyond what can be managed in the imperative paradigm.

6 Summary and Conclusions

This survey summarizes the technical developments in sequential Prolog implementation during the past decade and the systems that pioneered them. Much has happened in this time, and I hope that the survey is successful in capturing most of the important developments and in pointing out some intriguing trends for the future.

The WAM opened the floodgates for a proliferation of systems and ideas. It was the substrate upon which most sequential Prolog development took place in the past decade. Nowadays, the

WAM is no longer the best model to use for high performance. But it continues to be useful as a conceptual model, and the compilation principle that underlies the WAM is still highly relevant: to compile a logic language, simplify each occurrence of one of its basic operations with all the information at one's disposal. The last decade has seen an increased understanding of how this can be done: by measuring actual programs to optimize frequent operations, by learning how to compile unification and backtracking, and by using simpler instruction sets and global analysis.

The Prolog language has proven to be an elegant implementation target. The language has been generalized in many ways. There have been large advances in implementation technology, but there is still plenty to do, both in implementing Prolog and its successors. The next decade promises to be as interesting as the first.

References

1. Abderrahmane Aggoun and Nicolas Beldiceanu. Time Stamps Techniques for the Trailed Data in Constraint Logic Programming Systems. In *Actes du Séminaire 1990–Programmation en Logique*, CNET, Tregastel, France, May 1990.

2. Abderrahmane Aggoun and Nicolas Beldiceanu. Overview of the CHIP Compiler System. In *8th ICLP*, pages 775–789, MIT Press, June 1991.

3. Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools*. Addison–Wesley, 1986.

4. Hassan Aït-Kaci. *Warren's Abstract Machine, A Tutorial Reconstruction*. MIT Press, Cambridge, MA, 1991.

5. Hassan Aït-Kaci and Andreas Podelski. *Towards a Meaning of LIFE*. DEC PRL Research Report 11, Digital Equipment Corporation, Paris Research Laboratory, June 1991 (Revised October 1992).

6. Hassan Aït-Kaci and Andreas Podelski. *Functions as Passive Constraints in LIFE*. DEC PRL Research Report 13, Digital Equipment Corporation, Paris Research Laboratory, June 1991 (Revised November 1992).

7. Hassan Aït-Kaci, Bruno Dumant, Richard Meyer, Andreas Podelski, and Peter Van Roy. *The Wild LIFE Handbook*. Digital Equipment Corporation, Paris Research Laboratory, 1994.

8. Mohamed Amraoui. *Une Expérience de Compilation de PrologII sur MALI* (in French). Doctoral dissertation, Université de Rennes I, France, January 1988.

9. Andrew W. Appel. *Compiling with Continuations*. Cambridge University Press, 1992.

10. Bilbo Baggins and Frodo Baggins. *The Memoirs of Bilbo and Frodo of the Shire, Supplemented by the Accounts of Their Friends and the Learning of the Wise.* The Shire, Arnor, 3021 TA.

11. Joachim Beer. *Concepts, Design, and Performance Analysis of a Parallel Prolog Machine*. Ph.D. dissertation, Technische Universität Berlin, September 1987.

12. Joachim Beer. The Occur-Check Problem Revisited. In *JLP*, Elsevier North-Holland, vol. 5, no. 3, pages 243–261, September 1988.

13. Judit Bendl, Péter Köves, and Péter Szeredi. The MPROLOG System. In *Logic Programming Workshop*, pages 201–209, Debrecen, Hungary, 1980.

14. Hans Benker, J. M. Beacco, S. Bescos, M. Dorochevsky, Th. Jeffre, A. Pohimann, J. Noyé, B. Poterie, A. Sexton, J. C. Syre, O. Thibault, and G. Watzlawik. KCM: A Knowledge Crunching Machine. In *16th ISCA*, pages 186–194, IEEE Computer Society Press, May 1989.

15. J. Bocca. MegaLog–A Platform for Developing Knowledge Base Management Systems. In *2nd International Symposium on Database Systems for Advanced Applications*, pages 374–380, Tokyo, April 1991.

16. Kent Boortz. *SICStus Maskinkodskompilering* (in Swedish). SICS Technical Report T91:13, August 1991.

17. David L. Bowen, Lawrence M. Byrd, and William F. Clocksin. A Portable Prolog Compiler. In *Logic Programming Workshop*, pages 74–83, Algarve, Portugal, 1983.

18. Kenneth A. Bowen, Kevin A. Buettner, Ilyas Cicekli, and Andrew K. Turk. The Design and Implementation of a High-Speed Incremental Portable Prolog Compiler. In *3rd ICLP*, pages 650–656, Springer-Verlag LNCS 225, July 1986.

19. Roger S. Boyer and Jay S. Moore. The Sharing of Structure in Theorem Proving Programs. In *Machine Intelligence 7*, pages 101–116, Edinburgh University Press, New York, 1972.

20. Pascal Brisset and Olivier Ridoux. *The Compilation of λProlog and its Execution with MALI*. IRISA Publication Interne 687, Rennes, France, November 1992. Also published as INRIA Rapport de Recherche 1831, January 1993.

21. Maurice Bruynooghe. An Interpreter for Predicate Logic Programs. Report CW 10, Department of Computer Science, Katholieke Universiteit Leuven, Belgium, October 1976.

22. Maurice Bruynooghe. The Memory Management of Prolog Implementations. In *Logic Programming*, ed. K. Clark and S. Tärnlund, pages 83–98, Academic Press, 1982.

23. Mats Carlsson. On Implementing Prolog in Functional Programming. In *1st ICLP*, pages 154–159, IEEE Computer Society Press, February 1984.

24. Mats Carlsson. Freeze, Indexing, and Other Implementation Issues in the WAM. In *4th ICLP*, pages 40–58, MIT Press, May 1987.

25. Mats Carlsson. On the Efficiency of Optimising Shallow Backtracking in Compiled Prolog. In *6th ICLP*, pages 3–16, MIT Press, June 1989.

26. M. Carlsson, J. Widèn, J. Andersson, S. Andersson, K. Boortz, H. Nilsson, and T. Sjöland. *SICStus Prolog User's Manual*. SICS, Box 1263, 164 28 Kista, Sweden, 1991.

27. Weidong Chen and David Scott Warren. Query Evaluation under the Well-Founded Semantics. In *12th Symposium on Principles of Database Systems (PODS '93)*, ACM, 1993.

28. William F. Clocksin. Design and Simulation of a Sequential Prolog Machine. In *Journal of New Generation Computing (NGC)*, pages 101–120, vol. 3, no. 1, 1985.

29. Philippe Codognet and Daniel Diaz. Boolean Constraint Solving using clp(FD). In *10th ILPS*, pages 525–539, MIT Press, October 1993.

30. Helder Coelho and José C. Cotta. *Prolog by Example: How to Learn, Teach and Use It*. Springer-Verlag, 1988.

31. A. Colmerauer, H. Kanoui, and M. V. Caneghem. Prolog, Theoretical Principles and Current Trends. In *Technology and Science of Informatics*, 2(4):255–292, 1983.

32. A. Colmerauer. The Birth of Prolog. In *The Second ACM-SIGPLAN History of Programming Languages Conference*, pages 37–52, ACM SIGPLAN Notices, March 1993.

33. Vitor Santos Costa, David H. D. Warren, and Rong Yang. Andorra-I: A Parallel Prolog System that transparently exploits both And- and Or-parallelism. In *Proc. 3rd ACM SIGPLAN Conference on Principles and Practice of Parallel Programming*, pages 83–93, August 1991.

34. Patrick Cousot and Radhia Cousot. Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints. In *4th POPL*, pages 238–252, January 1977.

35. Patrick Cousot and Radhia Cousot. Abstract Interpretation and Application to Logic Programs. In *JLP*, Elsevier North-Holland, pages 103–179, vol. 13, nos. 2-3, July 1992.

36. Koen De Bosschere and Paul Tarau. *Continuation Passing Style Prolog-to-C Mapping at Native WAM-speed*. ELIS Technical Report DG 93-15, Universiteit Gent, Vakgroep Elektronica en Informatiesystemen, November 1993. Summary in *ACM Symposium on Applied Computing (SAC '94)*, March 1994 (forthcoming).

37. Saumya Debray. *Global Optimization of Logic Programs*. Ph.D. dissertation, Computer Science Department, SUNY Stony Brook, September 1986.

38. Saumya Debray. A Simple Code Improvement Scheme for Prolog. In *JLP*, Elsevier North-Holland, pages 57–88, vol. 13, no. 1, May 1992.

39. Saumya Debray. Implementing Logic Programming Systems: The Quiche-Eating Approach. In *ICLP '93 Workshop on Practical Implementations and Systems Experience*, Budapest, Hungary, June 1993.

40. Daniel Diaz and Philippe Codognet. A Minimal Extension of the WAM for clp(FD). In *10th ICLP*, pages 774–790, Budapest, Hungary, MIT Press, June 1993.

41. M. Dincbas, P. Van Hentenryck, H. Simonis, A. Aggoun, T. Graf, and F. Berthier. The Constraint Logic Programming Language CHIP. In *FGCS '88*, pages 693–702, Tokyo, November 1988.

42. T. P. Dobry. Performance Studies of a Prolog Machine Architecture. In *12th ISCA*, IEEE Computer Society Press, June 1985.

43. T. P. Dobry. *A High Performance Architecture for Prolog*. Ph.D. dissertation, Department of Computer Science, U.C. Berkeley, Report UCB/CSD 87/352, April 1987. Also published by Kluwer Academic Publishers, 1990.

44. Mireille Ducassé. Opium: An Advanced Debugging System. In *2nd Logic Programming Summer School*, Esprit Network of Excellence in Computational Logic (COMPULOG-NET), ed. G. Comyn and N. Fuchs, Springer-Verlag LNAI 636, September 1992.

45. ECRC. *ECLiPSe 3.2 User Manual*. August 1992.

46. E. W. Elcock. Absys: The First Logic Programming Language–A Retrospective and a Commentary. In *JLP*, Elsevier North-Holland, pages 1–17, vol. 9, no. 1, July 1990. Also published as Technical Report #210, Department of Computer Science, University of Western Ontario, July 1988.

47. Zsuzsa Farkas, Péter Köves, and Péter Szeredi. MProlog: An Implementation Overview. In *ICLP '93 Workshop on Practical Implementations and Systems Experience*, Budapest, Hungary, June 1993.

48. Hervé Gallaire. Boosting Logic Programming. In *4th ICLP*, pages 962–988, Melbourne, Australia, May 1987.

49. M. García de la Banda and M. Hermenegildo. A Practical Approach to the Global Analysis of CLP Programs. In *10th ILPS*, pages 435–455, MIT Press, October 1993.

50. Thomas Walter Getzinger. *Abstract Interpretation for the Compile-Time Analysis of Logic Programs*. Ph.D. dissertation, Advanced Computer Architecture Laboratory, University of Southern California, Report ACAL-TR-93-09, September 1993.

51. Michael M. Gorlick and Carl F. Kesselman. Gauge: A Workbench for the Performance Analysis of Logic Programs. In *5th ICSLP*, pages 548–561, MIT Press, August 1988.

52. David Gudeman. *Representing Type Information in Dynamically Typed Languages*. University of Arizona, Department of Computer Science, Report TR93-27, September 1993.

53. Yi-Ke Guo and Hendrik C. R. Lock. *A Classification Scheme for Declarative Programming Languages*. GMD-Studien Nr. 182, GMD, Germany, August 1990.

54. S. Habata, R. Nakazaki, A. Atarashi, and M. Umemara. Co-operative High Performance Sequential Inference Machine: CHI. In *International Conference on Computer Design (ICCD '87)*, pages 601–604, IEEE Computer Society Press, October 1987.

55. Seif Haridi and Sverker Janson. Kernel Andorra Prolog and its Computation Model. In *7th ICLP*, pages 31–48, MIT Press, Cambridge, June 1990.

56. Arie Harsat and Ran Ginosar. CARMEL-2: A Second Generation VLSI Architecture for Flat Concurrent Prolog. In *FGCS '88*, pages 962–969, Tokyo, November 1988.

57. Bogumił Hausman. Turbo Erlang. In *10th ILPS*, page 662, MIT Press, October 1993.

58. Ralph Clarke Haygood. Aquarius Prolog User Manual. In *Aquarius Prolog 1.0 documentation*, U.C. Berkeley, April 1993.

59. John L. Hennessy and David A. Patterson. *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann, 1990.

60. Manuel Hermenegildo, Richard Warren, and Saumya Debray. Global Flow Analysis as a Practical Compilation Tool. In *JLP*, Elsevier North-Holland, pages 349–367, vol. 13, no. 4, August 1992.

61. Timothy Hickey and Shyam Mudambi. Global Compilation of Prolog. In *JLP*, Elsevier North-Holland, pages 193–230, vol. 7, no. 3, November 1989.

62. P. M. Hill and J. W. Lloyd. *The Gödel Programming Language*. Technical Report CSTR-92-27, Department of Computer Science, University of Bristol, October 1992 (Revised May 1993).

63. Bruce K. Holmer, Barton Sano, Michael Carlton, Peter Van Roy, Ralph Haygood, Joan M. Pendleton, T. P. Dobry, William R. Bush, and Alvin M. Despain. Fast Prolog with an Extended General Purpose Architecture. In *17th ISCA*, pages 282–291, IEEE Computer Society Press, May 1990.

64. Bruce K. Holmer. *Automatic Design of Computer Instruction Sets*. Ph.D. dissertation, Department of Computer Science, U.C. Berkeley, 1993.

65. Paul Hudak. Reflections on Program Optimization. Invited talk, *1993 Workshop on Static Analysis (WSA '93)*, page 193, Springer-Verlag LNCS 724, September 1993.

66. Gérard Huet. *Résolution d'Équations dans des Langages d'Ordre 1, 2, ..., ω* (in French). Thèse de Doctorat d'État, Université Paris VII, September 1976.

67. IBM. *IBM SAA AD/Cycle Prolog/MVS & VM Programmer's Guide and Language Reference*, Release 1, December 1992.

68. Joxan Jaffar. Efficient Unification over Infinite Terms. In *Journal of New Generation Computing (NGC)*, pages 207–219, vol. 2, no. 3, 1984.

69. Gerda Janssens and Maurice Bruynooghe. Deriving Descriptions of Possible Values of Program Variables by means of Abstract Interpretation. In *JLP*, Elsevier North-Holland, pages 205–258, vol. 13, nos. 2-3, July 1992.

70. Mark Kantrowitz. *Prolog Resource Guide*. Regularly posted on Internet newsgroups comp.lang.prolog and comp.answers.

71. Shmuel Kliger and Ehud Shapiro. From Decision Trees to Decision Graphs. In *NACLP90*, pages 97–116, MIT Press, October 1990.

72. Shmuel Kliger. *Compiling Concurrent Logic Programming Languages*. Ph.D. dissertation, Weizmann Institute, Rehovot, October 1992.

73. H. Komatsu, N. Tamura, Y. Asakawa, and T. Kurokawa. An Optimizing Prolog Compiler. In *Logic Programming '86*, pages 104–115, Springer-Verlag LNCS 264, June 1986.

74. M. Korsloot and E. Tick. Compilation Techniques for Nondeterminate Flat Concurrent Logic Programming Languages. In *8th ICLP*, pages 457–471, MIT Press, June 1991.

75. Péter Köves and Péter Szeredi. Getting the Most Out of Structure-Sharing. In *Collection of Papers on Logic Programming*, pages 69–84, SZKI, Budapest, 1988 (Revised November 1993).

76. Andreas Krall, Tim Lindholm, *et al*. Net Talk: Term Comparisons with Variables. In *ALP Newsletter*, pages 18–21, November 1992. From Internet newsgroup comp.lang.prolog, July 1992.

77. Andreas Krall and Ulrich Neumerkel. The Vienna Abstract Machine. In *PLILP '90*, pages 121–135, Springer-Verlag LNCS 456, August 1990.

78. K. Kurosawa, S. Yamaguchi, S. Abe, and T. Bandoh. Instruction Architecture for a High Performance Integrated Prolog Processor IPP. In *5th ICSLP*, pages 1506–1530, MIT Press, August 1988.

79. Peter Kursawe. How to Invent a Prolog Machine. In *3rd ICLP*, pages 134–148, Springer-Verlag LNCS 225, July 1986. Also in *Journal of New Generation Computing*, vol. 5, pages 97–114, 1987.

80. Baudouin Le Charlier, Kaninda Musumbu, and Pascal Van Hentenryck. A Generic Abstract Interpretation Algorithm and its Complexity Analysis (Extended Abstract). In *8th ICLP*, pages 64–78, MIT Press, June 1991.

81. Baudouin Le Charlier, Olivier Degimbe, Laurent Michel, and Pascal Van Hentenryck. Optimization Techniques for General Purpose Fixpoint Algorithms: Practical Efficiency for the Abstract Interpretation of Prolog. In *1993 Workshop on Static Analysis (WSA '93)*, pages 15–26, Springer-Verlag LNCS 724, September 1993.

82. Tim Lindholm and Richard A. O'Keefe. Efficient Implementation of a Defensible Semantics for Dynamic Prolog Code. In *4th ICLP*, pages 21–39, MIT Press, May 1987.

83. Peter Ludemann. *xpProlog: High Performance Extended Pure Prolog*. Master's thesis, University of British Columbia, 1988.

84. Michael J. Maher. Logic Semantics for a Class of Committed-Choice Programs. In *4th ICLP*, pages 858–876, MIT Press, 1987.

85. David Maier and David Scott Warren. *Computing with Logic–Logic Programming with Prolog*. Benjamin/Cummings, 1988.

86. André Mariën. An Optimal Intermediate Code for Structure Creation in a WAM-based Prolog Implementation. Katholieke Universiteit Leuven, Belgium, May 1988.

87. André Mariën, Gerda Janssens, Anne Mulkers, and Maurice Bruynooghe. The Impact of Abstract Interpretation: An Experiment in Code Generation. In *6th ICLP*, pages 33–47, MIT Press, June 1989.

88. André Mariën and Bart Demoen. A New Scheme for Unification in WAM. In *ILPS*, pages 257–271, MIT Press, October 1991.

89. André Mariën. *Improving the Compilation of Prolog in the Framework of the Warren Abstract Machine*. Ph.D. dissertation, Katholieke Universiteit Leuven, September 1993.

90. Kim Marriott, Maria García de la Banda, and Manuel Hermenegildo. Analyzing Logic Programs with Dynamic Scheduling. In *20th POPL*, ACM, 1994.

91. K. Mehlhorn and A. Tsakalidis. Data Structures. Chapter 6 of *Handbook of Theoretical Computer Science*, Volume A: Algorithms and Complexity, pages 301–341, MIT Press/Elsevier, 1990.

92. Micha Meier. *Shallow Backtracking in Prolog Programs*. Internal report, ECRC, Munich, Germany, February 1987.

93. M. Meier, A. Aggoun, D. Chan, P. Dufresne, R. Enders, D. Henry de Villeneuve, A. Herold, P. Kay, B. Perez, E. van Rossum, and J. Schimpf. SEPIA–An Extendible Prolog System. In *Proceedings of the 11th World Computer Congress IFIP'89*, pages 1127–1132, San Francisco, August 1989.

94. Micha Meier. Compilation of Compound Terms in Prolog. In *NACLP90*, pages 63–79, MIT Press, October 1990.

95. Micha Meier. *Better Late Than Never*. Internal report, ECRC, Munich, Germany, 1993. In *ICLP '93 Workshop on Practical Implementations and Systems Experience*, Budapest, Hungary, June 1993.

96. C. S. Mellish. *Automatic Generation of Mode Declarations for Prolog Programs* (draft). Department of Artificial Intelligence, University of Edinburgh, August 1981.

97. C. S. Mellish and S. Hardy. Integrating Prolog in the POPLOG environment. In *Implementations of PROLOG*, ed. J. A. Campbell, 1984, pages 147–162.

98. C. S. Mellish. Some Global Optimizations for a Prolog Compiler. In *JLP*, Elsevier North-Holland, pages 43–66, vol. 1, 1985.

99. Richard Meyer. Private communication. Digital Equipment Corporation, Paris Research Laboratory, December 1993.

100. Dale Miller and Gopalan Nadathur. Higher-Order Logic Programming. In *3rd ICLP*, pages 448–462, Springer-Verlag LNCS 225, July 1986.

101. Jonathan Wayne Mills. *LIBRA: A High-Performance Balanced Computer Architecture for Prolog*. Ph.D. dissertation, Arizona State University, December 1988.

102. Chris Moss and Ken Bowen (chairs). International Conference on the Practical Application of Prolog. ALP, London, April 1992.

103. Chris Moss and Al Roth. *The Prolog 1000 database*. Available through anonymous ftp from src.doc.ic.ac.uk in packages/prolog-progs-db/prolog1000.v1, August 1993.

104. Lee Naish. *MU-Prolog 3.1db Reference Manual*. Computer Science Department, University of Melbourne, Melbourne, Australia, May 1984.

105. Lee Naish. Negation and Quantifiers in NU-Prolog. In *3rd ICLP*, pages 624–634, Springer-Verlag LNCS 225, July 1986.

106. Lee Naish. *Negation and Control in Prolog*. Ph.D. dissertation, University of Melbourne, published as Springer-Verlag LNCS 238, 1986.

107. Lee Naish. Parallelizing NU-Prolog. In *5th ICSLP*, pages 1546–1564, MIT Press, August 1988. Also published as Technical Report 87/17, Department of Computer Science, University of Melbourne.

108. Lee Naish, Philip W. Dart, and Justin Zobel. The NU-Prolog Debugging Environment. In *6th ICLP*, pages 521–536, MIT Press, June 1989.

109. Hiroshi Nakashima and Katsuto Nakajima. Hardware Architecture of the Sequential Inference Machine: PSI-II. In *SLP*, pages 104–113, IEEE Computer Society Press, August 1987.

110. Ulrich Neumerkel. Une Transformation de Programme Basée sur la Notion d'Équations entre Termes (in French). In *Journées Francophones de Programmation en Logique (JFPL)*, pages 215–229, Nîmes, France, May 1993.

111. Jacques Noyé, Hans Benker, *et al*. *ICM3: The Abstract Machine*. Technical Report CA-19, ECRC, Munich, February 1987.

112. Jacques Noyé. An Overview of the Knowledge Crunching Machine. ECRC, Munich, Germany, 1993. In *Emerging Trends in Database and Knowledge-base Machines*. IEEE Computer Society Press (forthcoming).

113. Richard A. O'Keefe. *The Craft of Prolog*. MIT Press, 1990.

114. Doug Palmer and Lee Naish. NUA-Prolog: An Extension to the WAM for Parallel Andorra. In *8th ICLP*, pages 429–442, MIT Press, June 1991.

115. Fernando C. N. Pereira and Stuart M.Shieber. *Prolog and Natural-Language Analysis*. Center for the Study of Language and Information (CSLI), Lecture Notes Number 10, 1987.

116. Christian Pichler. *Prolog-Übersetzer* (in German). Master's thesis (Diplomarbeit), Institut für Praktische Informatik, Technische Universität Wien, November 1984.

117. Andreas Podelski and Peter Van Roy. The Beauty and the Beast Algorithm: Testing Entailment and Disentailment Incrementally. In *10th ILPS*, page 653, MIT Press, October 1993. Also Research Report (forthcoming), Digital Equipment Corporation, Paris Research Laboratory.

118. Kotagiri Ramamohanarao, John Shepherd, Isaac Balbin, Graeme Port, Lee Naish, James Thom, Justin Zobel, and Philip Dart. The NU-Prolog Deductive Database System. In *IEEE Data Engineering*, pages 10–19, vol. 10, no. 4, December 1987. Also in *Prolog and Databases*, Ellis Horwood, 1988. Also Technical Report 87/19, Department of Computer Science, University of Melbourne.

119. Konstantinos Sagonas, Terrance Swift, and David Scott Warren. *XSB: An Overview of its Use and Implementation*. SUNY Stony Brook, October 1993. Available through anonymous ftp from cs.sunysb.edu in pub/TechReports/warren/xsb_overview.ps.Z.

120. Dan Sahlin and Mats Carlsson. Variable Shunting for the WAM. In *NACLP '90 Workshop on Logic Programming Architectures and Implementations*, Austin, Texas, November 1990. Also available as SICS Research Report R91:07, March 1991.

121. Vijay Saraswat. *Concurrent Constraint Programming Languages*. Ph.D. dissertation, Carnegie-Mellon University, 1989. Revised version published by MIT Press, 1992.

122. Roger Scowen. ISO Draft Prolog Standard (N72). ISO/IEC JTC1 SC22 WG17, June 1991.

123. Kazuo Seo and Takashi Yokota. Pegasus: A RISC Processor for High-Performance Execution of Prolog Programs. In *VLSI '87*, Vancouver, Canada, pages 261–274, Elsevier North-Holland, August 1987.

124. Kazuo Seo and Takashi Yokota. Design and Fabrication of Pegasus Prolog Processor. In *VLSI '89*, Munich, Germany, pages 265–274, Elsevier North-Holland, August 1989.

125. Kazuo Seo. *Study of a VLSI Architecture for the Logic Programming Language Prolog* (in Japanese). Ph.D. dissertation, Keio University, March 1993.

126. Ehud Shapiro. The Family of Concurrent Logic Programming Languages. In *ACM Computing Surveys*, pages 412–510, vol. 21, no. 3, September 1989.

127. Aaron Sloman. The Evolution of Poplog and Pop-11 at Sussex University. In *POP-11 Comes of Age: The Advancement of an AI Programming Language*, ed. J. A. D. W. Anderson, pages 30–54, Ellis Horwood, 1989.

128. Vason P. Srini, J. Tam, T. Nguyen, C. Chen, A. Wei, J. Testa, Y. Patt, and A. M. Despain. VLSI Implementation of a Prolog Processor. In *Stanford VLSI Conference*, March 1987.

129. Vason P. Srini, Jerric V. Tam, Tam M. Nguyen, Yale N. Patt, Alvin M. Despain, Maurice Moll, and Dan Ellsworth. A CMOS Chip for Prolog. In *International Conference on Computer Design (ICCD '87)*, pages 605–610, IEEE Computer Society Press, October 1987.

130. Leon Sterling and Ehud Shapiro. *The Art of Prolog*. MIT Press, 1986.

131. Terrance Swift and David Scott Warren. *Compiling OLDT Evaluation: Background and Overview*. SUNY Stony Brook Technical Report 92/04, 1992. Available through anonymous ftp from cs.sunysb.edu in pub/TechReports/warren/xwam_overview.dvi.Z, June 1993.

132. Terrance Swift and David Scott Warren. *Performance of Sequential SLG Evaluation*. SUNY Stony Brook. Available through anonymous ftp from cs.sunysb.edu in pub/TechReports/warren/xsb-perf.ps.Z, November 1993.

133. K. Taki, M. Yokota, A. Yamamoto, H. Nishikawa, S. Uchida, H. Nakashima, and A. Mitsuishi. Hardware Design and Implementation of the Personal Sequential Inference Machine (PSI). In *FGCS '84*, 1984.

134. Kazuo Taki. Parallel Inference Machine PIM. In *FGCS '92*, 1992.

135. N. Tamura. Knowledge-Based Optimization in Prolog Compiler. In *ACM/IEEE Computer Society Fall Joint Conference*, November 1986.

136. Paul Tarau. A Compiler and a Simplified Abstract Machine for the Execution of Binary Metaprograms. In *Proceedings of the Logic Programming Conference '91*, pages 119–128, ICOT, Tokyo, September 1991.

137. Paul Tarau. *WAM-optimizations in BinProlog: Towards a Realistic Continuation Passing Prolog Engine*. Technical Report, Université de Moncton, Canada, 1992.

138. Andrew Taylor. Removal of Dereferencing and Trailing in Prolog Compilation. In *6th ICLP*, pages 48–60, MIT Press, June 1989.

139. Andrew Taylor. LIPS on a MIPS: Results from a Prolog Compiler for a RISC. In *7th ICLP*, pages 174–185, MIT Press, June 1990.

140. Andrew Taylor. *High-Performance Prolog Implementation*. Ph.D. dissertation, Basser Department of Computer Science, University of Sydney, June 1991.

141. Olivier Thibault. Hardware evaluation of KCM. ECRC, Munich, Germany, May 1990. In *Tools for Artificial Intelligence 1990*. IEEE Computer Society Press, November 1990.

142. Evan Tick and David H. D. Warren. Towards a Pipelined Prolog Processor. In *SLP*, pages 29–40, February 1984. Also in *Journal of New Generation Computing*, pages 323–345, vol. 2, no. 4, 1984.

143. Evan Tick. Memory- and Buffer-referencing Characteristics of a WAM-based Prolog. In *JLP*, Elsevier North-Holland, pages 133–162, November 1991.

144. R. M. Tomasulo. An Efficient Algorithm for Exploiting Multiple Arithmetic Units. In *IBM Journal*, pages 25–33, vol. 11, January 1967. Also as Chapter 19 in *Computer Structures: Principles and Examples*, ed. Siewiorek, Bell, and Newell, McGraw-Hill, pages 293–302, 1982.

145. Hervé Touati and Alvin M. Despain. An Empirical Study of the Warren Abstract Machine. In *SLP*, pages 114–124, IEEE Computer Society Press, August 1987.

146. Andrew K. Turk. *Compiler Optimizations for the WAM*. In *3rd ICLP*, pages 657–662, Springer-Verlag LNCS 225, July 1986.

147. Shunichi Uchida. Summary of the Parallel Inference Machine and its Basic Software. In *FGCS '92*, 1992.

148. Peter Van Roy. *A Prolog Compiler for the PLM*. Report UCB/CSD No. 84/203, Master's Report, U.C. Berkeley, November 1984.

149. Peter Van Roy, Bart Demoen, and Yves D. Willems. Improving the Execution Speed of Compiled Prolog with Modes, Clause Selection, and Determinism. In *TAPSOFT '87*, pages 111–125, Springer-Verlag LNCS 250, March 1987, also Report CW 51, K. U. Leuven.

150. Peter Van Roy. An Intermediate Language to Support Prolog's Unification. In *NACLP 89*, pages 1148–1164, MIT Press, October 1989.

151. Peter Van Roy. A Useful Extension to Prolog's Definite Clause Grammar notation. In *ACM SIGPLAN Notices*, pages 132–134, November 1989. See also *Extended DCG Notation: A Tool for Applicative Programming in Prolog*, Report UCB/CSD 90/583, U.C. Berkeley, July 1990.

152. Peter Van Roy and Alvin M. Despain. The Benefits of Global Dataflow Analysis for an Optimizing Prolog Compiler. In *NACLP 90*, pages 491–515, MIT Press, October 1990.

153. Peter Van Roy. *Can Logic Programming Execute as Fast as Imperative Programming?* Ph.D. dissertation, Department of Computer Science, U.C. Berkeley, Report UCB/CSD 90/600, December 1990. Revised version published as *Fast Logic Program Execution* by Intellect Books.

154. Peter Van Roy and Alvin M. Despain. High-Performance Logic Programming with the Aquarius Prolog Compiler. In *IEEE Computer*, pages 54–68, vol. 25, no. 1, January 1992.

155. Peter Van Roy. How to Get the Most Out of Aquarius Prolog. In *Aquarius Prolog 1.0 documentation*, Digital Equipment Corporation, Paris Research Laboratory, April 1993.

156. Peter Van Roy *et al*. *Aquarius Benchmarks*. Available through anonymous ftp from gatekeeper.dec.com in pub/plan/prolog/AquariusBenchmarks.tar.Z.

157. Paul J. Voda. Types of Trilogy. In *5th ICSLP*, pages 580–589, MIT Press, August 1988.

158. Philip Wadler. The Essence of Functional Programming. In *19th POPL*, pages 1–14, ACM Press, January 1992.

159. David H. D. Warren. *Applied Logic–Its Use and Implementation as a Programming Tool*. Ph.D. dissertation, University of Edinburgh, DAI Research Reports 39 & 40, 1977, also SRI Technical Report 290, 1983.

160. David H. D. Warren. Prolog on the DECsystem-10. In *Expert Systems in the Micro-Electronic Age*, Edinburgh University Press, 1979.

161. David H. D. Warren and Fernando C. N. Pereira. An Efficient Easily Adaptable System for Interpreting Natural Language Queries. In *American Journal of Computational Linguistics*, pages 110–122, vol. 8, nos. 3-4, July-December 1982.

162. David H. D. Warren. Higher-Order Extensions to Prolog–Are They Needed? In *Machine Intelligence 10*, Ellis Horwood, 1982.

163. David H. D. Warren. *An Abstract Prolog Instruction Set*. Technical Note 309, SRI International Artificial Intelligence Center, October 1983.

164. Richard Warren, Manuel Hermenegildo, and Saumya K. Debray. On the Practicality of Global Flow Analysis of Logic Programs. In *5th ICSLP*, pages 684–699, MIT Press, August 1988.

165. Günter Watzlawik, Hans Benker, Jacques Noyé, *et al*. *ICM4*. Technical Report CA-25, ECRC, Munich, Germany, February 1987.

166. Takashi Yokota and Kazuo Seo. Pegasus–An ASIC Implementation of High-Performance Prolog Processor. In *EURO ASIC '90*, Paris, France, pages 156–159, IEEE Computer Society Press, May 1990.

167. Neng-Fa Zhou. *Backtracking Optimizations in Compiled Prolog*. Ph.D. dissertation, Kyushu University, Fukuoka, Japan, November 1990.

168. Benjamin G. Zorn. *Comparative Performance Evaluation of Garbage Collection Algorithms*. Ph.D. dissertation, Department of Computer Science, U.C. Berkeley, Report UCB/CSD 89/544, December 1989.

PRL Research Reports

The following documents may be ordered by regular mail from:

Librarian – Research Reports
Digital Equipment Corporation
Paris Research Laboratory
85, avenue Victor Hugo
92563 Rueil-Malmaison Cedex
France.

It is also possible to obtain them by electronic mail. For more information, send a message whose subject line is `help` to `doc-server@prl.dec.com` or, from within Digital, to `decprl::doc-server`.

Research Report 1: *Incremental Computation of Planar Maps*. Michel Gangnet, Jean-Claude Hervé, Thierry Pudet, and Jean-Manuel Van Thong. May 1989.

Research Report 2: *BigNum: A Portable and Efficient Package for Arbitrary-Precision Arithmetic*. Bernard Serpette, Jean Vuillemin, and Jean-Claude Hervé. May 1989.

Research Report 3: *Introduction to Programmable Active Memories*. Patrice Bertin, Didier Roncin, and Jean Vuillemin. June 1989.

Research Report 4: *Compiling Pattern Matching by Term Decomposition*. Laurence Puel and Ascánder Suárez. January 1990.

Research Report 5: *The WAM: A (Real) Tutorial*. Hassan Aït-Kaci. January 1990.†

Research Report 6: *Binary Periodic Synchronizing Sequences*. Marcin Skubiszewski. May 1991.

Research Report 7: *The Siphon: Managing Distant Replicated Repositories*. Francis J. Prusker and Edward P. Wobber. May 1991.

Research Report 8: *Constructive Logics. Part I: A Tutorial on Proof Systems and Typed λ-Calculi*. Jean Gallier. May 1991.

Research Report 9: *Constructive Logics. Part II: Linear Logic and Proof Nets*. Jean Gallier. May 1991.

Research Report 10: *Pattern Matching in Order-Sorted Languages*. Delia Kesner. May 1991.

†This report is no longer available from PRL. A revised version has now appeared as a book: "Hassan Aït-Kaci, *Warren's Abstract Machine: A Tutorial Reconstruction. MIT Press, Cambridge, MA (1991)*."

Research Report 11: *Towards a Meaning of LIFE*. Hassan Aït-Kaci and Andreas Podelski. June 1991 (Revised, October 1992).

Research Report 12: *Residuation and Guarded Rules for Constraint Logic Programming*. Gert Smolka. June 1991.

Research Report 13: *Functions as Passive Constraints in LIFE*. Hassan Aït-Kaci and Andreas Podelski. June 1991 (Revised, November 1992).

Research Report 14: *Automatic Motion Planning for Complex Articulated Bodies*. Jérôme Barraquand. June 1991.

Research Report 15: *A Hardware Implementation of Pure Esterel*. Gérard Berry. July 1991.

Research Report 16: *Contribution à la Résolution Numérique des Équations de Laplace et de la Chaleur*. Jean Vuillemin. February 1992.

Research Report 17: *Inferring Graphical Constraints with Rockit*. Solange Karsenty, James A. Landay, and Chris Weikart. March 1992.

Research Report 18: *Abstract Interpretation by Dynamic Partitioning*. François Bourdoncle. March 1992.

Research Report 19: *Measuring System Performance with Reprogrammable Hardware*. Mark Shand. August 1992.

Research Report 20: *A Feature Constraint System for Logic Programming with Entailment*. Hassan Aït-Kaci, Andreas Podelski, and Gert Smolka. November 1992.

Research Report 21: *The Genericity Theorem and the Notion of Parametricity in the Polymorphic λ-calculus*. Giuseppe Longo, Kathleen Milsted, and Sergei Soloviev. December 1992.

Research Report 22: *Sémantiques des langages impératifs d'ordre supérieur et interprétation abstraite*. François Bourdoncle. January 1993.

Research Report 23: *Dessin à main levée et courbes de Bézier : comparaison des algorithmes de subdivision, modélisation des épaisseurs variables*. Thierry Pudet. January 1993.

Research Report 24: *Programmable Active Memories: a Performance Assessment*. Patrice Bertin, Didier Roncin, and Jean Vuillemin. March 1993.

Research Report 25: *On Circuits and Numbers*. Jean Vuillemin. November 1993.

Research Report 26: *Numerical Valuation of High Dimensional Multivariate European Securities*. Jérôme Barraquand. March 1993.

Research Report 27: *A Database Interface for Complex Objects*. Marcel Holsheimer, Rolf A. de By, and Hassan Aït-Kaci. March 1993.

Research Report 28: *Feature Automata and Sets of Feature Trees*. Joachim Niehren and

Andreas Podelski. March 1993.

Research Report 29: *Real Time Fitting of Pressure Brushstrokes*. Thierry Pudet. March 1993.

Research Report 30: *Rollit: An Application Builder*. Solange Karsenty and Chris Weikart. April 1993.

Research Report 31: *Label-Selective λ-Calculus*. Hassan Aït-Kaci and Jacques Garrigue. May 1993.

Research Report 32: *Order-Sorted Feature Theory Unification*. Hassan Aït-Kaci, Andreas Podelski, and Seth Copen Goldstein. May 1993.

Research Report 33: *Path Planning through Variational Dynamic Programming*. Jérôme Barraquand and Pierre Ferbach. September 1993.

Research Report 34: *A Penalty Function Method for Constrained Motion Planning*. Pierre Ferbach and Jérôme Barraquand. September 1993.

Research Report 35: *The Typed Polymorphic Label-Selective λ-Calculus*. Jacques Garrigue and Hassan Aït-Kaci. October 1993.

Research Report 36: *1983–1993: The Wonder Years of Sequential Prolog Implementation*. Peter Van Roy. December 1993.

Research Report 37: *Pricing of American Path-Dependent Contingent Claims*. Jérôme Barraquand and Thierry Pudet. January 1994.

Research Report 38: *Numerical Valuation of High Dimensional Multivariate American Securities*. Jérôme Barraquand and Didier Martineau. April 1994.

36

1983–1993: The Wonder Years of Sequential Prolog Implementation

Peter Van Roy

PARIS RESEARCH LABORATORY
85, Avenue Victor Hugo
92563 RUEIL MALMAISON CEDEX
FRANCE