Can Logic Programming Execute as Fast as Imperative Programming?

Peter Lodewijk Van Roy

Volume 1

Chapters & References

Abstract

The purpose of this dissertation is to provide constructive proof that the logic programming language Prolog can be implemented an order of magnitude more efficiently than the best previous systems, so that its speed approaches imperative languages such as C for a significant class of problems. The driving force in the design is to encode each occurrence of a general feature of Prolog as simply as possible. The resulting system, Aquarius Prolog, is about five times faster than Quintus Prolog, a high performance commercial system, on a set of representative programs. The design is based on the following ideas:

- **Reduce instruction granularity**. Use an execution model, the Berkeley Abstract Machine (BAM), that retains the good features of the Warren Abstract Machine (WAM), a standard execution model for Prolog, but is more easily optimized and closer to a real machine.
- **Exploit determinism**. Compile deterministic programs with efficient conditional branches. Most predicates written by human programmers are deterministic, yet previous systems often compile them in an inefficient manner by simulating conditional branching with backtracking.
- **Specialize unification**. Compile unification to the simplest possible code. Unification is a general pattern-matching operation that can do many things in the implementation: pass parameters, assign values to variables, allocate memory, and do conditional branching.
- **Dataflow analysis**. Derive type information by global dataflow analysis to support these ideas.

Because of limitations of the dataflow analysis, the system is not yet competitive with the C language for all programs. I outline the work that is needed to close the remaining gap.

Acknowledgments

This project has been an enriching experience in many ways. It was a privilege to be part of a team consisting of so many talented people, and I learned much from them. It was by trial and error that I learned how to manage the design of a large program that does not all fit into my head at once. Interaction with my colleagues encouraged the development of the formal specifications of BAM syntax and semantics, which greatly eased interfacing the compiler with the rest of the system. The use of the compiler by several colleagues, in particular the development of the run-time system in Prolog by Ralph Haygood, improved its robustness.

I wish to thank all those who have contributed in some way to this work. Al Despain is a wonderful advisor and a source of inspiration to all his students. Paul Hilfinger's fine-tooth comb was invaluable. Bruce Holmer's unfailing sharpness of thought was a strong support. I also would like to thank many friends, especially Ariel, Bernt, Francis, Hervé, Josh, Mireille, Sue, and Dr. D. von Tischtiegel. Veel dank ook aan mijn familie, et gros bisous pour Brigitte.

This research was partially sponsored by the Defense Advanced Research Projects Agency (DoD) and monitored by Space & Naval Warfare Systems Command under Contract No. N00014-88-K-0579.

Table of Contents

Chapter 3: The Two Representation Languages

Chapter 4: Kernel transformations

Chapter 5: Compiling Kernel Prolog to BAM Code

Chapter 6: BAM Transformations

Chapter 7: Evaluation of the Aquarius system

Chapter 8: Concluding Remarks and Future Work

Chapter 1

Introduction

‘‘You’re given the form,
but you have to write the sonnet yourself.
What you say is completely up to you.’’
– Madeleine L’Engle, *A Wrinkle In Time*

1. Thesis statement

The purpose of this dissertation is to provide constructive proof that the logic programming language Prolog can be implemented an order of magnitude more efficiently than the best previous systems, so that its speed approaches imperative languages such as C for a significant class of problems.

The motivation for logic programming is to let programmers describe *what* they want separately from *how* to get it. It is based on the insight that any algorithm consists of two parts: a logical specification (the *logic*) and a description of how to execute this specification (the *control*). This is summarized by Kowalski’s well-known equation Algorithm = Logic + Control [40]. Logic programs are statements describing properties of the desired result, with the control supplied by the underlying system. The hope is that much of the control can be automatically provided by the system, and that what remains is cleanly separated from the logic. The descriptive power of this approach is high and it lends itself well to analysis. This is a step up from programming in imperative languages (like C or Pascal) because the system takes care of low-level details of how to execute the statements.

Many logic languages have been proposed. Of these the most popular is Prolog, which was originally created to solve problems in natural language understanding. It has successful commercial implementations and an active user community. Programming it is well understood and a consensus has developed regarding good programming style. The semantics of Prolog strike a balance between efficient implementation and logical completeness [42, 82]. It attempts to make programming in a subset of first-order logic practical. It is a naive theorem prover but a useful programming language because of its mathematical foundation, its simplicity, and its efficient implementation of the powerful concepts of unification (pattern matching) and search (backtracking).

Prolog is being applied in such diverse areas as expert systems, natural language understanding, theorem proving [57], deductive databases, CAD tool design, and compiler writing [22]. Examples of successful applications are AUNT, a universal netlist translator [59], Chat-80, a natural language query system [81], and diverse in-house expert systems and CAD tools. Grammars based on unification have become popular in natural language analysis [55, 56]. Important work in the area of languages with implicit parallelism is based on variants of Prolog. Our research group has used Prolog successfully in the development of tools for architecture analysis [12, 16, 35], in compilation [19, 73, 76], and in silicon compilation [11].

Prolog was developed in the early 70's by Colmerauer and his associates [38]. This early system was an interpreter. David Warren's work in the late 70's resulted in the first Prolog compiler [80]. The syntax and semantics of this compiler have become the de facto standard in the logic programming community, commonly known as the Edinburgh standard. Warren's later work on Prolog implementation culminated in the development of the Warren Abstract Machine (WAM) in 1983 [82], an execution model that has become a standard for Prolog implementation.

However, these implementations are an order of magnitude slower than imperative languages. As a result, the practical application of logic programming has reached a crossroads. On the one hand, it could degenerate into an interesting academic subculture, with little use in the real world. Or it could flourish as a practical tool. The choice between these two directions depends crucially on improving the execution efficiency. Theoretical and experimental work suggests that this is feasible—that it is possible for an implementation of Prolog to use the powerful features of logic programming only where they are needed.

Therefore I propose the following thesis:

A program written in Prolog can execute as efficiently as its implementation in an imperative language. This relies on the development of four principles:

(1) An instruction set suitable for optimization.
(2) Techniques to exploit the determinism in programs.
(3) Techniques to specialize unification.
(4) A global dataflow analysis.

2. The Aquarius compiler

I have tested this thesis by constructing a new optimizing Prolog compiler, the Aquarius compiler. The design goals of the compiler are (in decreasing order of importance):

(1) **High performance**. Compiled code should execute as fast as possible.

(2) **Portability**. The compiler's output instruction set should be easily retargetable to any sequential architecture.

(3) **Good programming style**. The compiler should be written in Prolog in a modular and declarative style. There are few large Prolog programs that have been written in a declarative style. The compiler will be an addition to that set.

I justify the four principles given in the thesis statement in the light of the compiler design:

(1) **Reduce instruction granularity**. To generate efficient code it is necessary to use an execution model and instruction set that allows extensive optimization. I have designed the Berkeley Abstract Machine (BAM) which retains the good features of the Warren Abstract Machine (WAM) [82], namely the data structures and execution model, but has an instruction set closer to a sequential machine architecture. This makes it easy to optimize BAM code as well as port it to a sequential architecture.

(2) **Exploit determinism**. The majority of predicates written by human programmers are intended to be executed in a deterministic fashion, that is, to give only one solution. These predicates are in effect case statements, yet systems too often compile them inefficiently by using backtracking to simulate conditional branching. It is important to replace backtracking by conditional branching.

(3) **Specialize unification.** Unification is the foundation of Prolog. It is a general pattern-matching operation that can match objects of any size. Its logical semantics correspond to many possible actions in an implementation, including passing parameters, assigning values to variables, allocating memory, and conditional branching. Often only one of these actions is needed, and it is important to simplify the general mechanism. For example, one of the most common actions is assigning a value to a variable, which can often be simplified to a single load or store.

(4) **Dataflow analysis**. A global dataflow analysis supports techniques to exploit determinism and specialize unification by deriving information about the program at compile-time. The BAM instruction set is designed to express the optimizations possible by these techniques.

Simultaneously with the compiler, our research group has developed a new architecture, the VLSI-BAM, and its implementation. The first of several target machines for the compiler is the VLSI-BAM. The interaction between the architecture and compiler design has significantly improved both. This dissertation describes only the Aquarius compiler. A description of the VLSI-BAM and a cost/benefit analysis of its features is given elsewhere [34, 35].

3. Structure of the dissertation

The structure of the dissertation mirrors the structure of the compiler. Figure 1.1 gives an overview of this structure. Chapter 2 summarizes the Prolog language and previous techniques for its high performance execution. Chapters 3 through 6 describe and justify the design of the compiler in depth. Chapter 3 discusses its two internal languages: kernel Prolog, which is close to the source program, and the BAM, which is close to machine code. Chapter 4 gives the optimizing transformations of kernel Prolog. Chapter 5 gives the compilation of kernel Prolog into BAM. Chapter 6 gives the optimizing transformations of BAM code. Chapter 7 does a numerical evaluation of the compiler. It measures its performance on several machines, does an analysis of the effectiveness of its optimizations, and briefly compares its performance with the C language. Finally, chapter 8 gives concluding remarks and suggestions for further work.

The appendices give details about various aspects of the compiler. Appendix A is a user manual for the compiler. Appendices B and C give a formal definition of BAM syntax and semantics. Appendix D is an English description of BAM semantics. Appendix E describes the extended DCG notation, a tool that is used throughout the compiler's implementation. Appendix F lists the source code of the C and Prolog benchmarks. Appendix G lists the source code of the compiler.

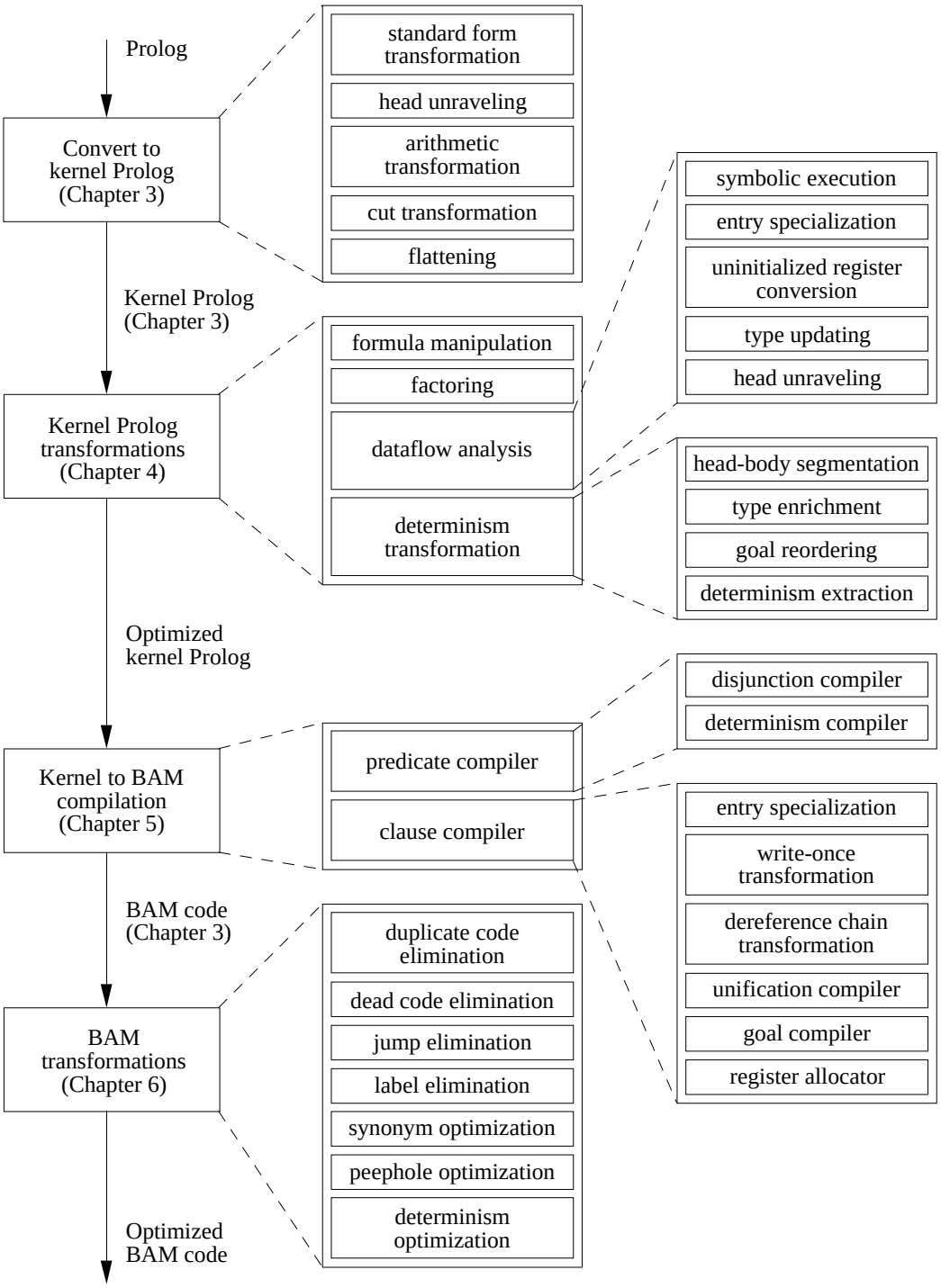

Figure 1.1 – Structure of the compiler and the dissertation

4. Contributions

4.1. Demonstration of high performance Prolog execution

A demonstration that the combination of a new abstract machine (the BAM), new compilation techniques, and a global dataflow analysis gives an average speedup of five times over Quintus Prolog [58], a high performance commercial system based on the WAM. This speedup is measured with a set of medium-sized, realistic Prolog programs. For small programs the dataflow analysis does better, resulting in an average speedup of closer to seven times. For programs that use built-in predicates in a realistic manner, the average speedup is about four times, since built-in predicates are a fixed cost. The programs for which dataflow analysis provides sufficient information are competitive in speed with a good C compiler.

On the VLSI-BAM processor, programs compiled with the Aquarius compiler execute in 1/3 the cycles of the PLM [28], a special-purpose architecture implementing the WAM in microcode. Static code size is three times the PLM, which has byte-coded instructions. The WAM was implemented on SPUR, a RISC-like architecture with extensions for Lisp [8], by macro-expansion. Programs compiled with Aquarius execute in 1/7 the cycles of this implementation with 1/4 the code size [34].

4.2. Test of the thesis statement

A test of the thesis that Prolog can execute as efficiently as an imperative language. The results of this test are only partially successful. Performance has been significantly increased over previous Prolog implementations; however the system is competitive with imperative languages only for problems for which dataflow analysis is able to provide sufficient information. This is due to the following factors:

- I have imposed restrictions on the dataflow analysis to make it practical. As programs become larger, these restrictions limit the quality of the results.
- The fragility of Prolog: minor changes in program text often greatly alter the efficiency with which the program executes. This is due to the under-specification of many Prolog programs, i.e. their logical meaning rules out computations but the compiler cannot deduce all cases where this happens.

For example, often a program is deterministic (does not do backtracking) even though the compiler cannot figure it out. This can result in an enormous difference in performance: often the addition of a single cut operation or type declaration reduces the time and space needed by orders of magnitude.

- The creation and modification of large data objects. The compilation of single assignment semantics into destructive assignment (instead of copying) in the implementation, also known as the *copy avoidance problem*, is a special case of the general problem of efficiently representing time in logic. A quick solution is to use nonlogical built-in predicates such as `setarg/3` [63]. A better solution based on dataflow analysis has not yet been implemented.
- Prolog's apparent need for architectural support. A general-purpose architecture favors the implementation of an imperative language. To do a fair comparison between Prolog and an imperative language, one must take the architecture into account. For the VLSI-BAM processor, our research group has analyzed the costs and benefits of one carefully chosen set of architectural extensions. With a 5% increase in chip area there is a 50% increase in Prolog performance.

4.3. Development of a new abstract machine

The development of a new abstract machine for Prolog implementation, the Berkeley Abstract Machine (BAM). This abstract machine allows more optimization and gives a better match to general-purpose architectures. Its execution flow and data structures are similar to the WAM but it contains an instruction set that is much closer to the architecture of a real machine. It has been designed to allow extensive low-level optimization as well as compact encoding of operations that are common in Prolog. The BAM includes simple instructions (register-transfer operations for a tagged architecture), complex instructions (frequently needed complex operations), and embedded information (allows better translation to the assembly language of the target machine). BAM code is designed to be easily ported to general-purpose architectures. It has been ported to several platforms including the VLSI-BAM, the SPARC, the MIPS, and the MC68020.

4.4. Development of the Aquarius compiler

The development of the Aquarius compiler, a compiler for Prolog into BAM. The compiler is sufficiently robust that it is used routinely for large programs. The compiler has the following distinguishing features:

- It is written in a modular and declarative style. Global information is only used to hold information about compiler options and type declarations.
- It represents types as logical formulas and uses a simple form of deduction to propagate information and improve the generated code. This extends the usefulness of dataflow analysis, which derives information about predicates, by propagating this information inside of predicates.
- It is designed to exploit as much as possible the type information given in the input and extended by the dataflow analyzer.
- It incorporates general techniques to generate efficient deterministic code and to encode each occurrence of unification in the simplest possible form.
- It supports a class of simplified unbound variables, called *uninitialized variables*, which are cheaper to create and bind than standard variables.

The compiler development proceeded in parallel with the development of a new Prolog system, Aquarius Prolog [31]. For portability reasons the system is written completely in Prolog and BAM code. The Prolog component is carefully coded to make the most of the optimizations offered by the compiler.

4.5. Development of a global dataflow analyzer

The development of a global dataflow analyzer as an integral part of the compiler. The analyzer has the following properties:

- It uses abstract interpretation on a lattice. Abstract interpretation is a general technique that proceeds by mapping the values of variables in the program to a (possibly finite) set of *descriptions*. Execution of the program over the descriptions completes in finite time and gives information about the execution of the original program.

- It derives a small set of types that lets the compiler simplify common Prolog operations such as variable binding and unification. These types are uninitialized variables, ground terms, nonvariable terms, and recursively dereferenced terms. On a representative set of Prolog programs, the analyzer finds nontrivial types for 56% of predicate arguments: on average 23% are uninitialized (of which one third are passed in registers), 21% are ground, 10% are nonvariables, and 17% are recursively dereferenced. The sum of these numbers is greater than 56% because arguments can have multiple types.
- It provides a significant improvement in performance, reduction in static code size, and reduction in the Prolog-specific operations of trailing and dereferencing. On a representative set of Prolog programs, analysis reduces execution time by 18% and code size by 43%. Dereferencing is reduced from 11% to 9% of execution time and trailing is reduced from 2.3% to 1.3% of execution time.
- It is limited in several ways to make it practical. Its type domain is small, so it is not able to derive many useful types. It has no explicit representation for aliasing, which occurs when two terms have variables in common. This simplifies implementation of the analysis, but sacrifices potentially useful information.

4.6. Development of a tool for applicative programming

The development of a language extension to Prolog to simplify the implementation of large applicative programs (Appendix E). The extension generalizes Prolog's Definite Clause Grammar (DCG) notation to allow programming with multiple named accumulators. A preprocessor has been written and used extensively in the implementation of the compiler.

Chapter 2

Prolog and Its High Performance Execution

This chapter gives an overview of the features of the Prolog language and an idea of what it means to program in logic. It summarizes previous work in its compilation and the possibilities of improving its execution efficiency. It concludes by giving an overview of related work in the area of high performance Prolog implementation.

1. The Prolog language

This section gives a brief introduction to the language. It gives an example Prolog program, and goes on to summarize the data objects and control flow. The syntax of Prolog is defined in Figure 2.2 and the semantics are defined in Figure 2.3 (section 2.1). Sterling and Shapiro give a more detailed account of both [62], as do Pereira and Shieber [56].

A *Prolog program* is a set of clauses (logical sentences) written in a subset of first-order logic called *Horn clause logic*, which means that they can be interpreted as *if*–statements. A *predicate* is a set of clauses that defines a relation, i.e. all the clauses have the same name and arity (number of arguments). Predicates are often referred to by the pair `name/arity`. For example, the predicate `in_tree/2` defines membership in a binary tree:

```
in_tree(X, tree(X,_,_)).
in_tree(X, tree(V,Left,Right)) :- X<V, in_tree(X, Left).
in_tree(X, tree(V,Left,Right)) :- X>V, in_tree(X, Right).
```

(Here ‘‘`:-`’’ means *if*, the comma ‘‘`,`’’ means *and*, variables begin with a capital letter, `tree(V,L,R)` is a compound object with three fields, and the underscore ‘‘_’’ is an anonymous variable whose value is ignored.) In English, the definition of `in_tree/2` can be interpreted as: ‘‘X is in a tree if it is equal to the node value (first clause), or if it is less than the node value and it is in the left subtree (second clause), or if it is greater than the node value and it is in the right subtree (third clause).’’

The definition of `in_tree/2` is directly executable by Prolog. Depending on which arguments are inputs and which are outputs, Prolog’s execution mechanism will execute the definition in different ways. The definition can be used to verify that X is in a given tree, or to insert or look up X in a tree.

The execution of Prolog proceeds as a simple theorem prover. Given a query and a set of clauses, Prolog attempts to construct values for the variables in the query that make the query true. Execution proceeds depth-first, i.e. clauses in the program are tried in the order they are listed and the predicates inside each clause (called *goals*) are invoked from left to right. This strict order imposed on the execution makes Prolog rather weak as a theorem prover, but useful as a programming language, especially since it can be implemented very efficiently, much more so than a more general theorem prover.

1.1. Data

The data objects and their manipulation are modeled after first order logic.

1.1.1. The logical variable†

A variable represents any data object. Initially the value of the variable is unknown, but it may become known by *instantiation.* A variable may be instantiated only once, i.e. it is *single-assignment*. Variables may be bound to other variables. When a variable is instantiated to a value, this value is seen by all the variables bound to it. Variables may be passed as predicate arguments or as arguments of compound data objects. The latter case is the basis of a powerful programming technique based on partial data structures which are filled in by different predicates.

1.1.2. Dynamic typing

Compound data types are first class objects, i.e. new types can be created at run-time and variables can hold values of any type. Common types are atoms (unique constants, e.g. `foo`, `abcd`), integers, lists (denoted with square brackets, e.g. `[Head|Tail]`, `[a,b,c,d]`), and structures (e.g. `tree(X,L,R)`, `quad(X,C,B,F)`). Structures are similar to C structs or Pascal records—they have a name (called the *functor*) and a fixed number of arguments (called the *arity*). Atoms, integers, and lists are used also in Lisp.

---

† Not to be confused with variables of type LOGICAL in Fortran.

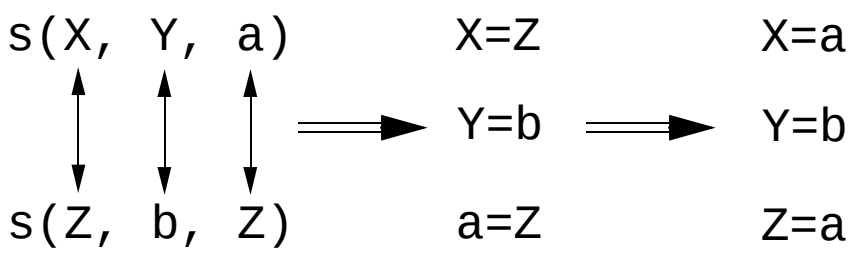

Figure 2.1 – An example of unification

1.1.3. Unification

Unification is a pattern-matching operation that finds the most general common instance of two data objects. A formal definition of unification is given by Lloyd [42]. Unification is able to match compound data objects of any size in a single primitive operation. Binding of variables is done by unification. As a part of matching, the variables in the terms are instantiated to make them equal. For example, unifying `s(X,Y,a)` and `s(Z,b,Z)` (Figure 2.1) matches X with Z, Y with `b`, and `a` with Z. The unified term is `s(a,b,a)`, Y is equal to `b`, and both X and Z are equal to `a`.

1.2. Control

During execution, Prolog attempts to satisfy the clauses in the order they are listed in the program. When a predicate with more than one clause is invoked, the system remembers this in a *choice point*. If the system cannot make a clause true (i.e. execution *fails*) then it backtracks to the most recent choice point (i.e. it undoes any work done trying to satisfy that clause) and tries the next clause. Any bindings made during the attempted execution of the clause are undone. Executing the next clause may give variables different values. In a given execution path a variable may have only one value, but in different execution paths a variable may have different values. Prolog is a single-assignment language: if unification attempts

to give a variable a different value then failure causes backtracking to occur. For example, trying to unify `s(a,b)` and `s(X,X)` will fail because the constants `a` and `b` are not equal.

There are four features that are used to manage the control flow. These are the ‘‘cut’’ operation (denoted by ‘‘`!`’’ in programs), the disjunction, the if-then-else construct, and negation-as-failure.

1.2.1. The cut operation

The cut operation is used to manage backtracking. A cut in the body of an clause effectively says: ‘‘This clause is the correct choice. Do not try any of the following clauses in this predicate when backtracking.’’ Executing a cut has the same effect in forward execution as executing `true`, i.e. it has no effect. But it alters the backtracking behavior. For example:

```
p(A) :- q(A), !, r(A).
p(A) :- s(A).
```

During execution of `p(A)`, if `q(A)` succeeds then the cut is executed, which removes the choice points created in `q(A)` as well as the choice point created when `p(A)` was invoked. As a result, if `r(A)` fails then the whole predicate `p(A)` fails. If the cut were not there, then if `r(A)` fails execution backtracks first to `q(A)`, and if that fails, then it backtracks further to the second clause of `p(A)`, and only when `s(A)` in the second clause fails does the whole predicate `p(A)` fail.

1.2.2. The disjunction

A disjunction is a concise way to denote a choice between several alternatives. It is less verbose than defining a new predicate that has each alternative as a separate clause. For example:

```
q(A) :- ( A=a ; A=b ; A=c ).
```

This predicate returns the three solutions `a`, `b`, and `c` on backtracking. It is equivalent to:

```
q(a).
q(b).
q(c).
```

1.2.3. If-then-else

The if-then-else construct is used to denote a selection between two alternatives in a clause when it is known that if one alternative is chosen then the other will not be needed. For example, the predicate `p(A)` above can be written as follows with an if-then-else:

```
p(A) :- ( q(A) -> r(A) ; s(A) ).
```

This has identical semantics as the first definition. The arrow `->` in an if-then-else acts as a cut that removes choice points back to the point where the if-then-else starts.

1.2.4. Negation-as-failure

Negation in Prolog is implemented by negation-as-failure, denoted by `\+(Goal)`. This is not a true negation in the logical sense so the symbol `\+` is chosen instead of `not`. A negated goal succeeds if the goal itself fails, and fails if the goal succeeds. For example:

```
r(A) :- \+ t(A).
```

The predicate `r(A)` will succeed only if `t(A)` fails. This has identical semantics as:

```
r(A) :- t(A), !, fail.
r(A).
```

In other words, if `t(A)` succeeds then the `fail` causes failure, and the cut ensures that the second clause is not tried. If `t(A)` fails then the second clause is tried because the cut is not executed. Note that negation-as-failure never binds any of the variables in the goal that is negated. This is different from a purely logical negation, which must return all results that are not equal to the ones that satisfy the goal. Negation-as-failure is sound (i.e. it gives logically correct results) if the goal being negated has no unbound variables in it.

1.3. Syntax

Figure 2.2 gives a Prolog definition of the syntax of a clause. The definition does not present the names of the primitive goals that are part of the system (e.g. arithmetic or symbol table manipulation). These primitive goals are called ‘‘built-in predicates.’’ They are defined in the Aquarius Prolog user

```
clause(H)      :- head(H).
clause((H:-B)) :- head(H), body(B).

head(H) :- goal_term(H).

body(G) :- control(G, A, B), body(A), body(B).
body(G) :- goal(G).

goal(G) :- \+control(G, _, _), goal_term(G).

control((A;B),  A, B).
control((A,B),  A, B).
control((A->B), A, B).
control(\+(A),  A, true).

term(T) :- var(T).
term(T) :- goal_term(T).

goal_term(T) :- nonvar(T), functor(T, _, A), term_args(1, A, T).

term_args(I, A, _) :- I>A.
term_args(I, A, T) :- I=<A, arg(I, T, X), term(X), I1 is I+1, term_args(I1, A, T).

% Built-in predicates needed in the definition:
functor(T, F, A) :- (Term T has functor F and arity A).
arg(I, T, X) :- (Argument I of compound term T is X).
var(T) :- (Argument T is an unbound variable).
nonvar(T) :- (Argument T is a nonvariable).
```

Figure 2.2 – The syntax of Prolog

manual [31]. The figure defines the syntax after a clause has already been read and converted to Prolog's internal form. It assumes that lexical analysis and parsing have already been done. Features of Prolog that depend on the exact form of the input (i.e. operators and the exact format of atoms and variables) are not defined here.

To understand this definition it is necessary to understand the four built-in predicates that it uses. The predicates `functor(T, F, A)` and `arg(I, T, X)` are used to examine compound terms. The predicates `var(T)` and `nonvar(T)` are opposites of each other. Their meaning is straightforward: they check whether a term T is unbound or bound to a nonvariable term. For example, `var(_)` succeeds whereas `var(foo(_))` does not.

2. The principles of high performance Prolog execution

The first implementation of Prolog was developed by Colmerauer and his associates in France as a by-product of research into natural language understanding. This implementation was an interpreter. The first Prolog compiler was developed by David Warren in 1977. Somewhat later Warren developed an execution model for compiled Prolog, the Warren Abstract Machine (WAM) [82]. This was a major improvement over previous models, and it has become the de facto standard implementation technique. The WAM defines a high-level instruction set that corresponds closely to Prolog.

This section gives an overview of the operational semantics of Prolog, the principles of the WAM, a summary of its instruction set, and how to compile Prolog into it. For more detailed information, please consult Maier & Warren [43] or Ait-Kaci [1]. The execution model of the Aquarius compiler, the BAM (Chapter 3), uses data structures similar to those of the WAM and has a similar control flow, although its instruction set is different.

2.1. Operational semantics of Prolog

This section summarizes the operational semantics of Prolog. It gives a precise statement of how Prolog executes without going into details of a particular implementation. This is useful to separate the execution of Prolog from the many optimizations that are done in the WAM and BAM execution models. This section may be skipped on first reading.

Figure 2.3 defines the semantics of Prolog as a simple resolution-based theorem prover. For clarity, the definition has been limited in the following ways: It does not assume any particular representation of terms. It does not show the implementation of cut, disjunctions, if-then-else, negation-as-failure, or built-in predicates. It assumes that variables are renamed when necessary to avoid conflicts. It assumes that failed unifications do not bind any variables. It assumes also that the variable bindings formed in successful unifications are accumulated until the end of the computation, so that the final bindings give the computed answer.

Terminology: A *goal* G is a predicate call, which is similar to a procedure call. A *resolvent* R is a list of goals [G_1 , G_2 , ... , G_r]. The *query* Q is the goal that starts the execution. The *program* is a list of

```
function prolog_execute(Q : goal) : boolean;
var
    B : stack of pair (list of goal, integer);  /* the backtrack stack */
    R : list of goal;  /* the resolvent */
    i  : integer;  /* index into program clauses */
begin
    R := [ Q ];
    B := empty;
    push (R,1) on B;
    while true do begin
        /* Control step: find next clause. */
        if empty(B) then return false else pop B into (R, i);
        if (R = [ ]) then return true;
        if ( i+1 ≤ n ) then push (R, i+1) on B;

        /* Resolution step: try to unify with the clause. */
        /* At this point, R = [ G1 , ... , Gr ] and Ai = (Hi :- Ai1 , ... , Aia_i) */
        /* Unify the first goal in R with clause Ai . */
        unify G1 and Hi;
        if successful unification then begin
            /* In R, replace G1 by the body of Ai  */
            /* If Ai does not have a body, then R is shortened by one goal */
            R := [ Ai1 , ... , Aia_i , G2 , ... , Gr ];
            push (R,1) on B   /* proceed to next goal */
        end
    end
end;
```

Figure 2.3 – Operational definition of Prolog execution

clauses [A_1 , A_2 , ... , A_n]. The number of clauses in the program is denoted by n. Each clause A_i has a head H_i and an optional body given as a list of goals [A_{i1} , A_{i2} , ... , A_{ia_i}].

Execution starts by setting the initial resolvent R to contain the query goal Q. In a resolution-based theorem prover, the resolvent is transformed in successive steps until (1) it becomes empty, in which case execution succeeds, (2) all the clause choices are exhausted, in which case execution fails, or (3) the program goes into an infinite loop. In a single transformation step, a goal G is taken from the current resolvent R and unified with a clause in the program. The next resolvent is obtained by replacing G by the body of the clause.

This process is nondeterministic, and much work has been done in the area of automatic theorem proving to reduce the size of its search space [7]. To get efficiency, the approach of Prolog is to restrict the

process in two ways: by always taking the *first* goal from R and by trying clauses in the *order* they are listed in the program (Figure 2.3). If no successful match is found, then the program backtracks—a previous resolvent is popped off the backtrack stack and execution continues. Therefore the execution flow of Prolog is identical to that of a procedural language, with the added ability to backtrack to earlier execution states.

The function prolog_execute(Q) returns a boolean that indicates whether execution was successful or not (Figure 2.3). If execution was successful, then there is a set of bindings for the variables in Q that gives the result of the computation. As a definition, prolog_execute(Q) faithfully mirrors the execution of Prolog. As an implementation, however, it is incredibly inefficient. For each clause that is tried, it pushes and pops the complete resolvent (which can be very large) on the backtrack stack. The backtrack stack grows with each successful resolution step. A practical implementation avoids much of this overhead.

The next section describes the WAM, an execution model that is much more efficient. In the WAM, the resolvents are stored in a compact form on several stacks. Only the differences between successive resolvents are stored, so that memory usage is much less. The stack discipline is used to make backtracking efficient. The WAM also defines a representation for data items that allows an efficient implementation of unification.

2.2. Principles of the WAM

The WAM defines a mapping between the terminology of logic and of a sequential machine (Figure 2.4). Predicates correspond to procedures. Procedures are always written as one large case statement. Clauses correspond to the arms of this case statement. The scope of variable names is a single clause. (Global variables exist; however their use is inefficient and is discouraged.) Goals in a clause correspond to calls. Unification corresponds to parameter passing and assignment. Tail recursion corresponds to iteration. Features that do not map directly are the single-assignment nature and altering backtracking behavior with the cut operation.

The WAM is based on four ideas: use tagged pointers to represent dynamically typed data, optimize backtracking (exploit determinism by doing a conditional branch on the first argument), specialize

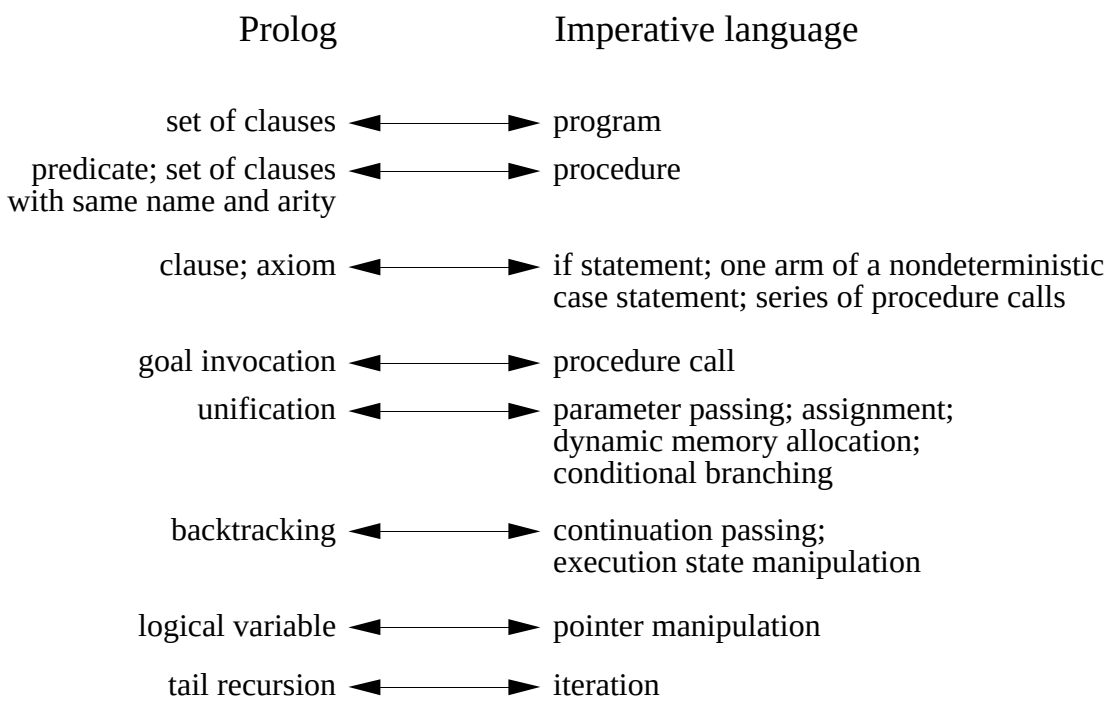

Figure 2.4 – Mapping between Prolog and an imperative language (according to WAM)

unification (instead of compiling a general unification algorithm, compile instructions that unify with a known term), and map the execution of Prolog to a real machine. The WAM defines a high-level instruction set to represent these operations.

2.2.1. Implementation of dynamic typing with tags

Data is represented by objects that fit in a register and consist of two parts: the tag field (which gives the type) and the value field (Figure 2.5). The value field is used for different purposes in different types: it gives the value of integers, the address of variables and compound terms (lists and structures), and it ensures that each atom has a unique value different from all other atoms. Unbound variables are implemented as self-referential pointers (that is, they point to themselves) or as pointers to other unbound variables. The semantics of unification allow variables to be unified together, so that they have identical values from then on. In the implementation, such variables can point to other variables. Therefore retrieving the value of a variable requires following this pointer chain to its end, an operation called *dereferencing*.

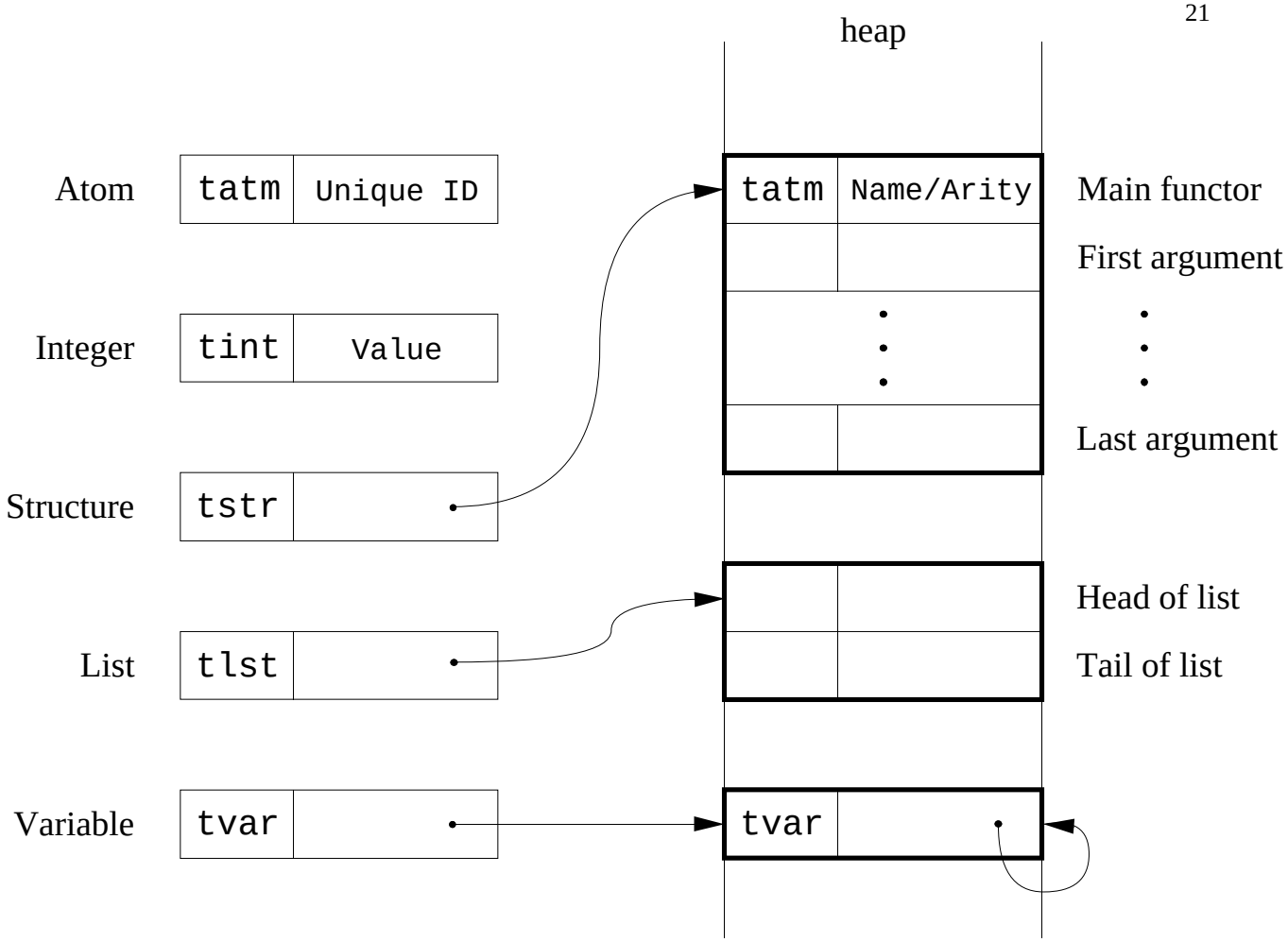

Figure 2.5 – Representation of Prolog terms in WAM and BAM

2.2.2. Exploit determinism

It is often possible to reduce the number of clauses of a predicate that must be tried. The WAM has instructions that hash on the value of the first argument and do a four-way branch on the tag of the first argument. These instructions avoid the execution of clauses that could not possibly unify with the goal. The four-way branch distinguishes between the four data types—variables, constants (atoms and integers), lists (cons cells), and structures. The hashing instructions hash into tables of constants and tables of structures. For example:

```
week(monday).
week(tuesday).
week(wednesday).
week(thursday).
week(friday).
week(saturday).
week(sunday).
```

This is a set of seven clauses with constant arguments. If the argument X of the call `week(X)` is a constant, then at most one clause can unify successfully with it. Hashing is used to pick that clause. If X is an unbound variable then no such optimization is possible and all clauses are tried in order.

2.2.3. Specialize unification

Most uses of unification are special cases of the general unification algorithm and can be compiled in a simpler way using information known at compile-time. For example, consider the following clause which is part of a queue-handling package:

```
% queue(X,Q) is true
% if Q is a queue containing the single element X.

queue(X, q(s(0),[X|C],C)).
```

A queue is represented here as a compound term. The complexity of this term is typical of real programs. In the WAM, a unification in the source code is compiled into a sequence of high-level instructions. The compiled code executes as if the original clause had been defined as follows, with the nested term `q/3` completely unraveled:

```
queue(X, Q) :- Q=q(A,B,C), A=s(0), B=[X|C].
```

(The notation `P=Q` means to unify the two terms P and Q.) The compiled code is:

```
procedure  queue/2

get_structure  q/3,r(1)    % Q=q(     <- Start unification of q/3
unify_variable  r(2)       %     A,
unify_variable  r(3)       %     B,
unify_variable  r(4)       %     C)
get_structure  s/1,r(2)    % A=s(     <- Start unification of s/1
unify_constant  0          %     0)
get_list  r(3)             % B=       <- Start unification of list
unify_value  r(0)          %    [X
unify_value  r(4)          %    |C]
proceed                    %          <- Return to caller
```

(`r(0)` and `r(1)` are registers holding the arguments X and Q, and `r(2)`, `r(3)`, ... are temporary registers.) Unification of the nested structure is expanded into a sequence of operations that do special cases of the general algorithm. These operations are encapsulated in the `get` and `unify` instructions. Unification has two modes of operation: it can take apart an existing structure or it can create a new one.

In the WAM, the decision which mode to use is made at run-time in the `get` instructions by checking the type of the object being unified. A mode flag is set which affects the actions of the following `unify` instructions (up to the next `get`). A more detailed overview of the WAM instruction set is given in section 2.3 below.

2.2.4. Map execution to a real machine

The control flow of Prolog is mapped to multiple stacks. The stack representation holds the resolvents in a form that makes each resolution step as efficient as a procedure call in an imperative language. The stack-based structure allows fast recovery of memory on backtracking. As a result, some applications do not need a garbage collector.

A further optimization maps Prolog variables to registers. The variables in a clause are partitioned into three classes (temporary, permanent, and void) depending on their lifetimes. Void variables have no lifetime and need no storage. Temporary variables do not need to survive across procedure calls, so they can be stored in machine registers. Permanent variables are stored in environments (i.e. stack frames) local to a clause.

2.3. Description of the WAM

The previous section gave an overview of the ideas in the WAM, with a simple example of generated code. This section completes that description by presenting the data storage, execution state, and instruction set of the WAM in full. It also gives a larger example of generated code and a scheme to compile Prolog into WAM.

2.3.1. Memory areas

Memory of the WAM is divided into six logical areas (Figure 2.6): three stacks for the data objects, one stack to support unification, one stack to support the interaction of unification and backtracking, and one area as code space.

(1) **The global stack**. This stack is also known as the *heap*, although it follows a stack discipline. This stack holds terms (lists and structures, the compound data of Prolog).

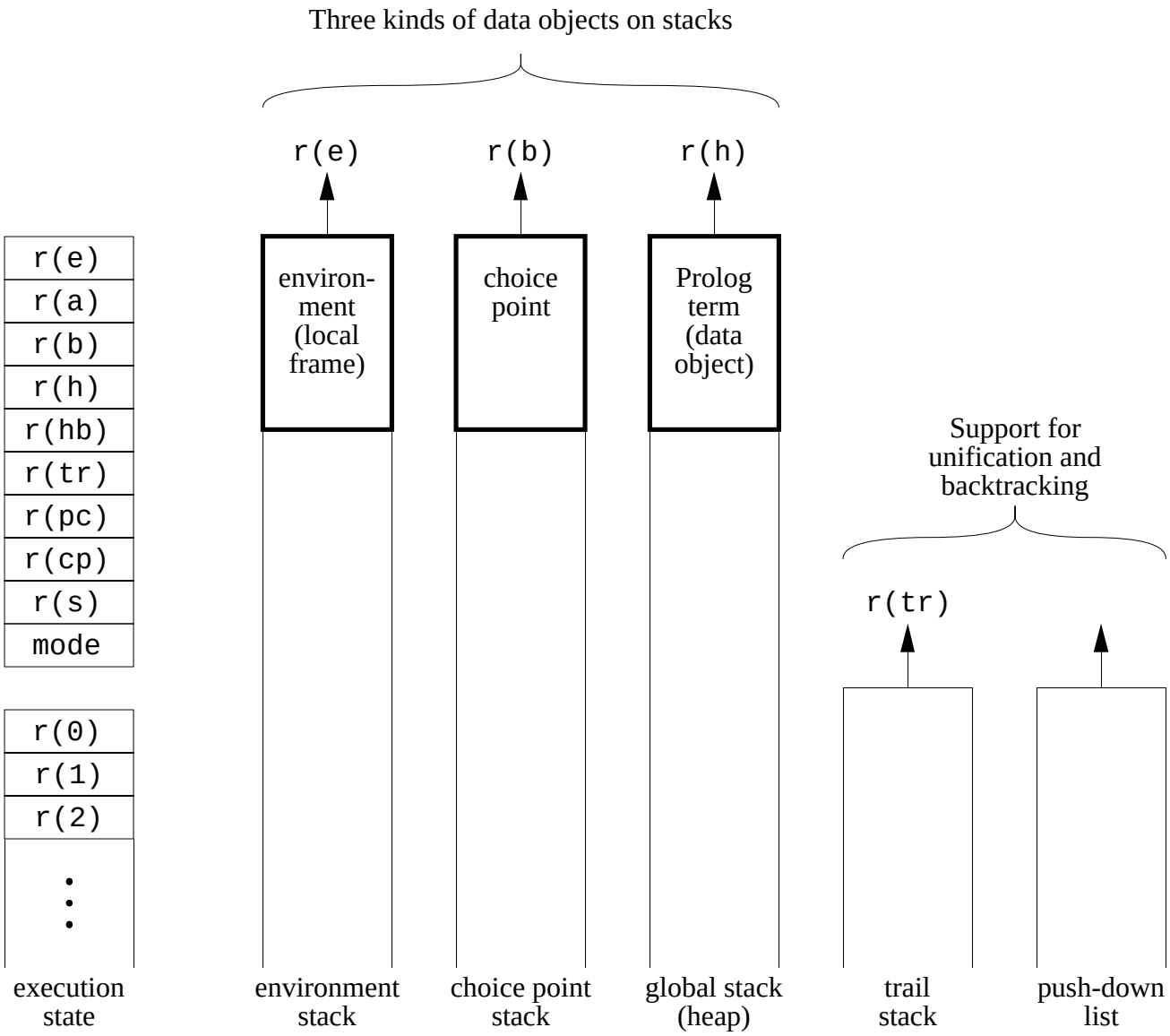

Figure 2.6 – Data structures of WAM and BAM

(2) **The environment stack**. This stack holds environments (i.e. local frames) which contain variables local to a clause. Because of backtracking (control may return to a clause whose environment is deep inside the stack), this area does not follow a strict stack discipline, however, convention has kept this naming. (The other stacks in the WAM do follow a stack discipline.)

(3) **The choice point stack**. Also known as the backtrack stack, this stack holds choice points, data objects similar to closures that encapsulate the execution state for backtracking.

(4) **The trail**. The trail stack is used to save locations of bound variables that have to be unbound on backtracking. Saving variables is called *trailing*, and restoring them to 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 heap or the environment is older than the most recent choice point. This is called the *trail condition.*

(5) **The push-down stack**. This stack is used as a scratch-pad during the unification of nested compound terms.

(6) **The code space**. This area holds the compiled code of a program.

It is possible to vary the organization of the memory areas somewhat without changing anything substantial about the execution. For example, some Prolog systems (including the Aquarius system) combine the environment and choice point stacks into a single memory area. This area is often called the local stack. Since the push-down stack is only used during general unification, it can be kept on the top of the heap.

2.3.2. Execution state

The internal state of the WAM and the BAM is given in Table 2.1. The differences between WAM and BAM are indicated in the table: The BAM adds the register `r(tmp_cp)` for efficient interfacing of Prolog predicates with assembly language. The WAM adds the register `r(s)` and the mode flag `mode` for use by the unification instructions. The registers `p(I)` are not machine registers, but locations in the current environment, pointed to by `r(e)`.

Table 2.1 – Execution state of WAM and BAM

Register	Description
`r(e)`	Current environment on the environment stack.
`r(a)`	Top of the environment stack (WAM only).
`r(b)`	Top-most choice point on the choice point stack.
`r(h)`	Top of the heap.
`r(hb)`	Top of heap when top-most choice point was created.
`r(tr)`	Top of the trail stack.
`r(pc)`	Program counter.
`r(cp)`	Continuation pointer (return address).
`r(tmp_cp)`	Continuation pointer to interface with assembly (BAM only).
`r(s)`	Structure pointer (WAM only).
`mode`	Unification mode flag (value is `read` or `write`, WAM only).
`r(0)`, `r(1)`, ...	Registers for argument passing and temporary storage.
`p(0)`, `p(1)`, ...	Locations in the current environment (permanent variables).

2.3.3. The instruction set

Table 2.2 contains the WAM instruction set, with a brief description of what each instruction does. The `get_(...)` and `unify_(...)` instructions echo the `put` instructions, so their listing is abbreviated. `v(N)` is shorthand notation for `r(N)` or `p(N)`. ‘‘Globalizing’’ a variable (see the `put_unsafe_value` instruction) moves an unbound variable from the environment to the heap to avoid dangling pointers.

Table 2.2 – The WAM instruction set

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_unsafe_value v(N), r(I)`	Move `v(N)` to `r(I)` (and globalize).
`put_constant C, r(I)`	Move immediate value `C` to `r(I)`.
`put_nil r(I)`	Move `nil` to `r(I)`.
`put_structure F, r(I)`	Create functor `F`, put in `r(I)`.
`put_list r(I)`	Create a list pointer, put in `r(I)`.
Unifying with registers and structure arguments (head unification)	
`get_(...), r(I)`	Unify `(...)` with `r(I)`.
`unify_(...)`	Unify `(...)` with structure argument.
Procedural control	
`call Label, N`	Call a predicate.
`execute Label`	Jump to a predicate.
`proceed`	Return from a predicate.
`allocate`	Create local stack frame.
`deallocate`	Remove local stack frame.
Selecting a clause (conditional branching)	
`switch_on_term V,C,L,S`	Four-way branch on `r(0)`’s tag.
`switch_on_constant N, Tbl`	Hash table lookup of an atomic term in `r(0)`.
`switch_on_structure N, Tbl`	Hash table lookup of a functor in `r(0)`.
Backtracking (choice point management)	
`try_me_else Label` `try Label`	Create a choice point.
`retry_me_else Label` `retry Label`	Change retry address.
`trust_me_else fail` `trust Label`	Remove top-most choice point.

2.3.4. An example of WAM code

Figure 2.7 gives the Prolog definition and the WAM instructions for the predicate `append/3`. The mapping between Prolog and WAM instructions is straightforward: the `switch` instruction branches to the right clause depending on the type of the first argument, the choice point (`try`) instructions link the clauses together, the `get` instructions unify with the head arguments, and 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 decision which mode to use is made in the `get` instructions, which set a mode flag. For example, if `get_list r(0)` 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.

Choice point handling 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 a stack in memory. It is compiled before the first clause in a predicate. It continues execution with the next instruction and backtracks to label L. (The `try L` instruction is identical to `try_me_else`, except that it continues execution at L and backtracks to the next instruction.) 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 before all clauses after the first but not including the last. The `trust_me_else fail` instruction removes the top-most choice point from the stack. It is compiled before the last clause in a predicate.

2.3.5. Compiling into WAM

Compiling Prolog into WAM is straightforward because there is almost a one-to-one mapping between items in the Prolog source code and WAM instructions. Figure 2.8 gives a scheme for compiling Prolog to WAM. This compilation scheme generates suboptimal code. One can optimize it by generating `switch` instructions to avoid choice point creation in some cases [73].

The clauses of predicate `p/3` are compiled into blocks of code that are linked together with `try` instructions to manage choice points. 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 optimization*, or LCO (also called *tail recursion optimization*, although it is applicable to all predicates, not just recursive ones) converts a call instruction followed by a return into a jump, i.e. it

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

Prolog definition of `append/3`

```
append/3:
    switch_on_term V1, C1, C2, fail  ; Go to V1 if r(0) is a variable.
                                     ; Go to C1 if r(0) is a constant.
                                     ; Go to C2 if r(0) is a list.
                                     ; Fail if r(0) is a structure.
V1: try_me_else V2                   ; Create a choice point.
C1: get_nil r(0)                     ; Unify r(0) with nil.
    get_value r(1),r(2)              ; Unify r(1) and r(2).
    proceed                          ; Return to caller.

V2: trust_me_else fail               ; Remove choice point.
C2: get_list r(0)                    ; Start unification of r(0) with a list.
    unify_variable r(3)              ; Load head of list into r(3).
    unify_variable r(0)              ; Load tail of list into r(0).
    get_list r(2)                    ; Start unification of r(2) with a list.
    unify_value r(3)                 ; Unify head of list with r(3).
    unify_variable r(2)              ; Load tail of list into r(2).
    execute append/3                 ; Jump to append/3 (last call optimization).
```

WAM code for `append/3`

Figure 2.7 – Compiling `append/3` into WAM code

reduces memory usage on the environment stack. For recursive predicates, the LCO converts recursion into iteration, since the jump is to the first instruction of the predicate. The WAM implements a generalization of last call optimization called *environment trimming* that allows the environment to become smaller after each call.

3. Going beyond the WAM

Prolog implementations have made great progress in execution efficiency with the development of the WAM [82]. However, these systems are still an order of magnitude slower than implementations of popular imperative languages such as C. To improve the execution speed it is necessary to go beyond the WAM. This section discusses the limits of the WAM and how the four principles of the Aquarius compiler build on the WAM to achieve higher performance.

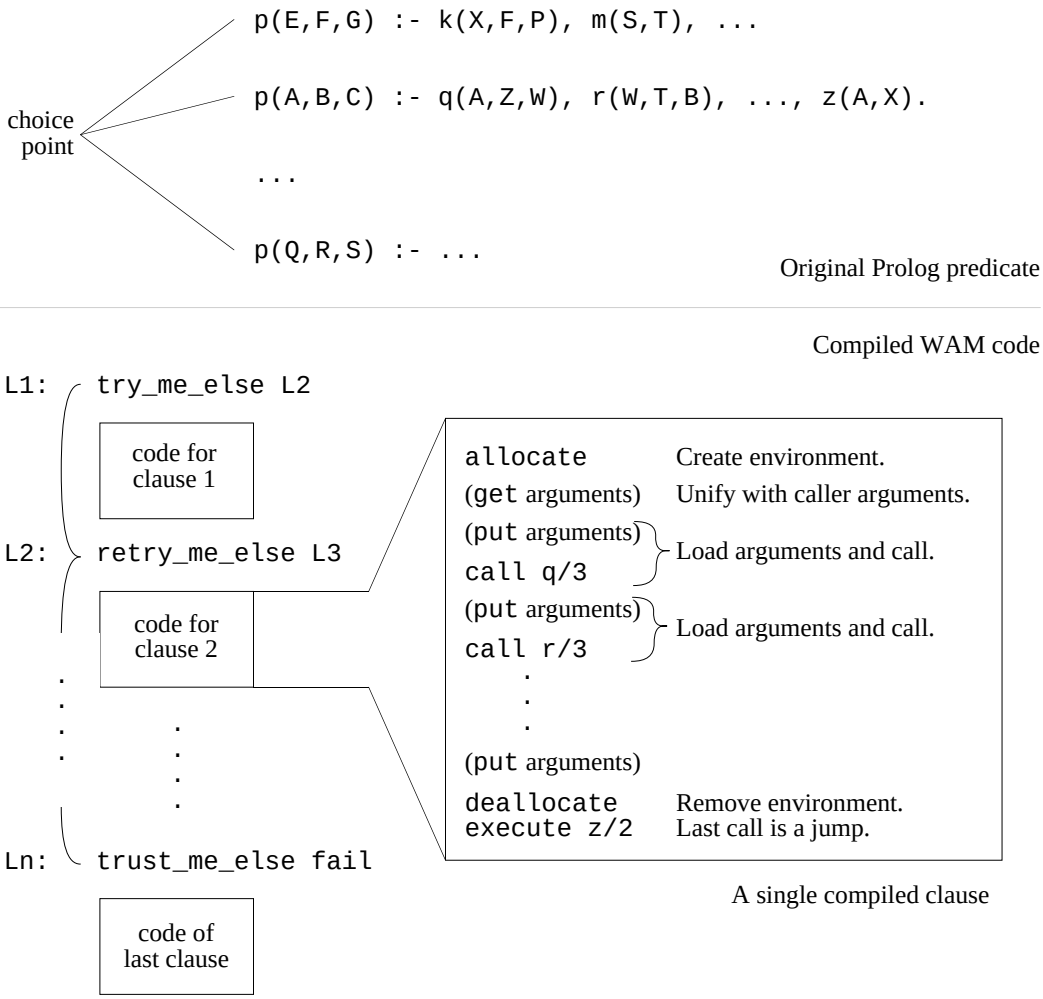

Figure 2.8 – Compiling Prolog into WAM

3.1. Reduce instruction granularity

The WAM is an elegant mapping of Prolog to a sequential machine. Its instructions encapsulate parts of the general unification algorithm. However, these parts are quite large, so that many optimizations are not possible. For example, consider the predicate:

```
p(bar).
```

This is compiled as:

```
get_constant bar, r(0)
proceed
```

The `get_constant` instruction encapsulates a series of operations: dereference `r(0)` (follow the pointer chain to its end), test its type, and do either read mode unification (check that the value of `r(0)` is `bar`) or write mode unification (trail `r(0)` and store `bar` in its cell). All this generality is often unnecessary. For example, if the predicate `p(X)` is always called with a dereferenced atom, then unification reduces to a simple check that the value is correct. The other operations are superfluous.

The Aquarius compiler's execution model, the BAM, is designed to retain the good features of the WAM while allowing optimizations such as this one. It retains data structures and an execution flow similar to the WAM, but it has an instruction set of finer granularity (Chapter 3). The compiler does not use the WAM during compilation, but directly compiles to the BAM. It is of fine enough grain to allow extensive optimization, but it also encodes compactly the operations common in Prolog. For example, it includes an explicit dereferencing instruction, which makes it possible to reduce the amount of dereferencing significantly by only doing it when it is necessary and not in every instruction.

3.2. Exploit determinism

The majority of predicates written by human programmers are intended to give only one solution, i.e. they are deterministic. However, too often they are compiled in an inefficient manner using *shallow* backtracking (backtracking within a predicate to choose the correct clause), when they are really just case statements. This is inefficient since backtracking requires saving the machine state and restoring it repeatedly.

3.2.1. Measurement of determinism

Measurements of Prolog applications support these assertions:

(1) Tick shows that choice point references constitute about half (45-60%) of all data references [69].

(2) Touati and Despain show that at least 40% of all choice point and fail operations can be removed through optimization [70].

The latter result is especially interesting because it attempts to quantify how often shallow backtracking is

optimizable. It considers a choice point to be *avoidable* if between the access of a choice point and its removal by a cut there are no calls to non-built-in predicates, no returns, and only binding of variables that do not have to be restored on backtracking. Avoidable choice points do not have to be created because they are removed immediately. For a set of medium-sized programs, on average the following percentages of choice point creations are avoidable: 57% of the ones removed by cut, 43% of the ones removed by trust, and 48% of the ones restored by fail. The variance of these numbers is large, but the potential for optimization when these situations do occur is significant. The Aquarius compiler is able to take advantage of these optimizations and more, e.g. due to the *factoring* transformation (Chapter 4) it is able to compile the `partition/4` predicate in Warren's quicksort benchmark [30] into deterministic code. The optimizations are synergistic, that is, doing them makes other improvements possible:

(1) Less stack space is needed on the environment/choice point stack. Choice points and environments are both stored on this stack, which means that often a clause's environment is hidden underneath a more recently created choice point. When this happens the last call optimization is not able to recover space. If fewer choice points are created, then last call optimization is effective more often.

(2) There are fewer memory references to the heap because binding a variable is postponed until a clause is chosen.

(3) There is less trailing because it is only needed for bindings that cross a choice point.

(4) Garbage collection is more efficient, since the creation of fewer choice points means that there are fewer starting points for marking.

3.2.2. Ramifications of exploiting determinism

The goal of compiling deterministic predicates into efficient conditional branches affects a large part of the compiler. Many of the transformations done in the compiler are intended to increase the amount of determinism that is easily accessible. This includes formula manipulation, factoring, head unraveling, the determinism transformation (all in Chapter 4), the determinism compiler (Chapter 5), and the determinism optimization (Chapter 6).

Through these transformations the compiler creates a decision graph to index the arguments of a predicate. Type information derived by dataflow analysis is exploited to simplify the graph. The graph is created in an architecture-independent way through the concept of the *test set* (Chapter 4). Intuitively, a test set is a set of Prolog predicates that are mutually disjoint (only one can succeed at any given time) and that correspond to a multi-way branch in the architecture.

3.3. Specialize unification

The WAM unification instructions (`get` and `unify`) are complex. They operate in two modes (read mode and write mode) depending on the type of the object being unified, they dereference their arguments, and they trail variable bindings. It is better to compile unification directly into simpler instructions.

In the Aquarius compiler, unification is compiled into the simplest possible BAM code taking the type information into account (Chapter 5). Often it is possible to reduce a unification to a single load or store. The use of uninitialized variables (see below) to simplify variable binding greatly improves the generated code.

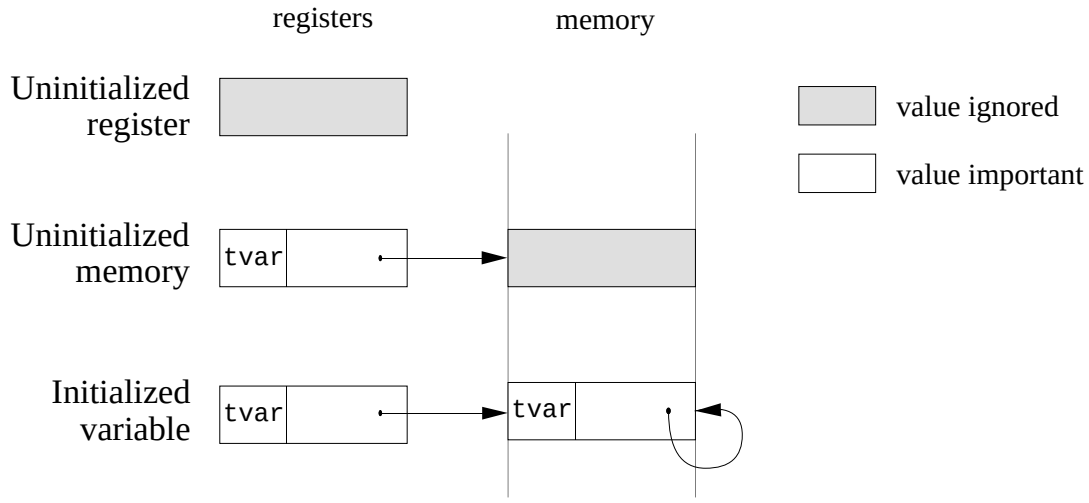

Figure 2.9 – Three categories of unbound variables

3.3.1. Simplifying variable binding

A major source of inefficiency in WAM implementations is that logical variables are often created as unbound (i.e. as self-referential pointers) and then unified soon afterwards. Creating and unifying does much unnecessary work; it would be faster just to reserve a memory location and then write to it. The Aquarius compiler defines such a representation, called *uninitialized variables*. Conceptually, uninitialized variables are defined at two levels:

(1) At the logical level, an uninitialized variable is an unbound variable that is not aliased, i.e. there are no other variables bound to it. The dataflow analyzer (Chapter 4) uses this definition to derive uninitialized variable types.

(2) At the implementation level, an uninitialized variable is a location that is allocated to contain an unbound variable, but the location is not given a value. The kernel Prolog compiler (Chapters 4, 5, and 6) uses this definition to compile uninitialized variables efficiently.

The location containing an uninitialized variable can either be a register or a memory word, resulting in two kinds of uninitialized variables, namely uninitialized register and uninitialized memory variables. The first are registers whose contents are ignored. The second are pointers to memory locations whose contents are ignored. Standard unbound variables are called initialized variables; they are pointers to locations pointing to themselves. Figure 2.9 illustrates the three categories of unbound variables.

Table 2.3 – The cost of uninitialized variables

Type of variable	Cost (VLSI-BAM cycles)			
	For Unification		For Backtracking	
	Creation	Binding	Trailing	Detrailing
Uninitialized Register	0	0	0	0
Uninitialized Memory	1	1	0	0
Initialized Variable	2	5	2	0 or 4

The dataflow analyzer derives both uninitialized register and uninitialized memory types. It is often able to determine that an argument is uninitialized; for a representative set of programs it finds that 23% of all predicate arguments are uninitialized. Of these, two thirds have uninitialized memory type and one third have uninitialized register type.

Table 2.3 gives the minimum run-time costs on the VLSI-BAM processor for the three categories of unbound variables. Costs are given for unification support (creation and binding) and for backtracking support (trailing and detrailing). Binding an initialized variable is expensive because the variable must be dereferenced before the new value can be stored in the memory cell. Binding an uninitialized memory variable reduces to a single memory store operation. Binding an uninitialized register variable is free if it is created in the register that needs it. The cost of detrailing (restoring a variable to an unbound state on backtracking) is zero for uninitialized variables. For initialized variables it depends strongly on the effectiveness of the compiler in generating deterministic code. It is 0 cycles if the variable does not have to be unbound on backtracking, and 4 cycles otherwise.

3.4. Dataflow analysis

The Aquarius compiler implements a dataflow analyzer that is based on abstract interpretation. It translates the program to one in which predicate arguments range over a finite set of values. Each of the values corresponds to an infinite set of values (i.e. a *type*) in the original program. The analyzer derives a small set of types—uninitialized, ground (the argument contains no unbound variables), nonvariable (the argument is not an unbound variable) and recursively dereferenced (the argument is dereferenced, i.e. it is accessible without pointer chasing, and if it is compound, then all its arguments are recursively dereferenced). These types have been chosen carefully to be useful during compilation.

Dataflow analysis by itself is not enough. The rest of the system must be able to use the information derived by the analysis. The techniques to exploit determinism and specialize unification in the Aquarius compiler have been developed in tandem with the analyzer for this purpose. In addition, the fine instruction granularity of the BAM is designed to support these optimizations.

4. Related work

First a survey is given of work that is related to the four principles of the Aquarius compiler. Then an overview is given of Prolog implementations that are interesting in some way.

4.1. Reduce instruction granularity

Tamura et al [39, 65] have done fundamental work at IBM Japan in reducing the grain size of compiled operations for Prolog. Their compilation is done in three steps. The first step is to compile Prolog into WAM. In the second step the intermediate code is translated into a directed graph. Each WAM instruction becomes a subgraph containing simple operations such as case selection on tags, jumps, assignments, and dereferencing. The graph is optimized through rewrite rules. Case selections based on a tag value, never-selected cases, redundant tests, case statements with only one branch, and unreachable instructions are eliminated. Known values are propagated. These rewrites are applied several times and the resulting graph is then translated back into intermediate code. In the third step the intermediate code is translated into a PL.8 program which is sent to a high-quality PL.8 optimizing compiler [3]. Performance results are given for a few small programs and are quite good. There are several problems in their approach. They still use the WAM as an intermediate language, and compiling is prohibitively slow because their system is experimental. Without compile-time hints their performance drops significantly.

4.2. Exploit determinism

Significant improvements over the WAM are possible to avoid choice point creation in deterministic predicates. The WAM indexes on only the first argument and saves all registers in choice points. Turk [72] describes several optimizations that reduce the time necessary to restore machine state when backtracking. In [74], I describe a compilation scheme that attempts to take advantage of the fact that most Prolog predicates are deterministic. Choice point creation and moves to and from choice points are minimized. Clauses are compiled with multiple entry points and predicates are compiled as decision trees. The techniques used in the Aquarius system are inspired by this work. Carlsson [15] measures the performance improvement of a scheme for creating choice points in two parts, saving only a small part of the machine state first, and postponing saving the remainder until later in the clause when it can be determined that the head unification and any simple tests have succeeded. Implemented in the SICStus Prolog system, this reduces execution time by 7-15% on four large programs.

Recently there have appeared several commercial Prolog-like languages (Trilogy and Turbo Prolog)

that generate efficient code for programs annotated with type and determinism declarations. In this regard Trilogy [79] is noteworthy because it gives a logical semantics to programs written in a Pascal-like notation. Typed predicates that are annotated as being deterministic are compiled into efficient native code. The achievement of Trilogy is reassuring; since many predicates in standard Prolog are intended to be executed in a deterministic way, with some analysis it should be possible to obtain the same efficiency for standard Prolog.

Several systems have generalized the first argument indexing of the WAM. BIM_Prolog [4] can index on any argument when given appropriate declarations. SEPIA [29] incorporates heuristics to decide which predicate arguments are important for deterministic selection. It uses the first ‘‘indexable’’ argument of a predicate. If there are several possibilities it first uses the argument where it is more likely that fewer clauses will be selected.

Several papers describe fast implementations of the cut operation. Bowen et al [9] implement cut by adding a register that holds the address of the most recent choice point before entering the predicate. This register is updated by each `call` and `execute` instruction. Cut is implemented by moving this register to the WAM's choice point register `r(b)`. Mariën and Demoen [46] implement cut in a similar fashion. These schemes suffer from having to do an additional register move for each procedure call, unless a different call instruction is used for predicates with and without cut. The scheme implemented in the Aquarius compiler does not slow down procedure calls and does not need an additional register.

4.3. Specialize unification

Significant improvements over the WAM are possible for unification. Turk [72] describes several optimizations related to compilation of unification, to reduce the overhead of explicitly maintaining a read/write mode bit and remove some superfluous dereferencing and tag checking. Mariën [44] describes a method to compile write mode unification that uses a minimal number of memory operations and avoids all superfluous dereferencing and tag checking. In [75], I build on this work by introducing a simplified notation and extending it for read mode unification, but my scheme suffers from a large code size expansion. The Aquarius system modifies this technique to limit the code size expansion at a slight execution time

cost. Meier [48] has developed a technique that generalizes Mariën's idea for both read and write mode and achieves a linear code size, also with a slight execution time cost. This technique is implemented in the SEPIA system [29].

Beer [5] has suggested the use of a simplified representation of Prolog variables for which binding is much faster. He introduces several new tags for this representation, which he calls *uninitialized variables*, and keeps track of them at run-time. He shows that both dereferencing and trailing are reduced significantly. This idea was a strong influence on the Aquarius compiler. At the Prolog level, logical semantics are preserved, but at the code level there is now a coherent integrated use of destructive assignment for values that fit in a register. My scheme is different from Beer's—it uses the same tag for both uninitialized and standard Prolog variables. The analyzer finds uninitialized variables at compile-time and the compiler determines when it is safe to use destructive assignment to bind them.

4.4. Dataflow analysis

R. Warren et al [84] have done the most comprehensive work measuring the practicality of global dataflow analysis in logic programming. Their paper describes two dataflow analyzers: (1) MA^3, the MCC And-parallel Analyzer and Annotator, and (2) Ms, an experimental analysis scheme developed for SB-Prolog. MA^3 derives aliasing and ground types and keeps track of the structure of compound terms, while Ms derives ground and nonvariable types. The paper concludes that both dataflow analyzers are effective in deriving types and do not increase compilation time by too much. My dataflow analyzer differs from both MA^3 and Ms in three ways. First, the analyzer works over a different domain. Second, it avoids problems with aliased variables by deriving only limited type information for them. Third, it is integrated into a compiler which has been developed to take full advantage of the types it derives.

For correctness, it is imperative to consider the effects of variable aliasing on dataflow analysis. Aliasing occurs when two variables are bound to terms that have variables in common. Finding accurate aliasing information is an important topic in current research [18, 36]. However, aliasing complicates the implementation of dataflow analysis. My analyzer considers only unaliased variables as candidates for unbound variable types. Measurements of the analyzer show that unaliased variables occur often enough

to make the analysis worthwhile. This conservative treatment of aliasing simplifies the implementation, since it is not necessary to explicitly represent and propagate aliasing information. Of course, it also reduces the effectiveness of the analysis. Thus aliasing needs to be studied further.

Mariën et al [45] have performed an interesting experiment in which several small Prolog predicates (recursive list operations) were hand-compiled with several levels of optimization based on information derivable from a dataflow analysis. The analysis was done by hand at four levels: The first level derives unbound variable and ground modes. The second level also derives recursively defined types. The third level also derives lengths of dereference chains (pointer chains that must be followed at run-time). The fourth level also derives liveness information for compound data structures and is used to determine when they are last used so that their memory may be recovered (compile-time garbage collection). Execution time measurements show that each analysis level improves speed over the previous level. This experiment shows that a simple analysis can achieve good results on small programs.

4.5. Other implementations

This section gives an overview of interesting Prolog implementations that are related to this dissertation in some way. Most existing implementations of Prolog, both on general-purpose and special-purpose machines, are based on the Warren Abstract Machine (WAM) or are derived from it. The general-purpose and special-purpose approaches are presented separately. The first subsection describes some important software implementations and their ideas. The second subsection summarizes some important architectures and their innovations.

4.5.1. Implementing Prolog on general-purpose machines

As far as I know, the earliest WAM compiler was my PLM compiler, completed and published in August 1984 [73].‡ The compiler was interesting as it was itself written in Prolog, unlike many later Prolog compilers. The first commercial implementation of the WAM was Quintus Prolog, announced in November 1984.

‡ The PLM compiler is still available from us, but is now obsolete and not recommended for current research work. Our research group expects to release soon a complete Prolog system based on the Aquarius compiler.

Among the highest performance commercial implementations available today are IBM Prolog, Quintus Prolog [58], BIM_Prolog [4], and ALS Prolog [2]. There are three significant implementations of Prolog available today that were developed at research institutions: SICStus Prolog [63], SEPIA [29], and SB-Prolog [83]. All of these systems are based on extensions of the WAM (except possibly IBM Prolog, of which I have little information) and compile to WAM-like instructions which are either emulated on the target machine or macro-expanded to native code. Some of these systems (e.g. SB-Prolog and IBM Prolog) are able to compile special cases of deterministic programs into efficient code.

4.5.1.1. Taylor's system

Independently of this research, Andrew Taylor is implementing a high performance Prolog compiler for the MIPS processor [67]. The compiler includes a dataflow analyzer that explicitly represents type, aliasing, dereference chain lengths, and trailing information [66]. His preliminary results indicate that it is of comparable performance to the compiler presented in this dissertation. Running a set of small benchmark programs on the MIPS R2030 processor, the system is 24 times faster than compiled SICStus Prolog version 0.6 and the code size is similar to that of the KCM.

4.5.1.2. IBM Prolog

IBM Prolog accepts mode declarations, implements more general indexing than the WAM, does a limited global analysis (however, it does not derive any types), and generates high performance native code. It is able to compile some kinds of deterministic programs with conditional branches.

4.5.1.3. SICStus Prolog

SICStus Prolog was developed at the Swedish Institute of Computer Science in Stockholm. A back-end module was written for it by Mats Carlsson which generates native code avoiding the superfluous memory references of a naive WAM translation [14, 44]. It is comparable in performance to Quintus Prolog when no built-in predicates are used.

4.5.1.4. SB-Prolog

SB-Prolog was developed at SUNY in Stony Brook. It recognizes a special case of the general techniques for extracting determinism discussed in this dissertation: it recognizes when arithmetic tests that are each other's opposites appear, and compiles a conditional branch. It also incorporates a simple partial evaluator which is used for macro expansion and a simple dataflow analysis scheme has recently been developed for it [84].

4.5.2. Implementing Prolog on special-purpose machines

In the past, because the WAM was regarded as the best way to implement Prolog, the performance gap between special-purpose architectures and general-purpose architectures was large. Much of the effort in high performance Prolog implementation was put into architecture design, and in particular in hardware support for the WAM instructions. This dissertation shows that a better understanding of Prolog execution narrows the performance gap. The implications of this development for the future of special-purpose architectures are discussed in the VLSI-BAM paper [34] and summarized in this section.

4.5.2.1. PLM

The first special-purpose Prolog architecture that was built is the PLM (Programmed Logic Machine), due to Dobry et al [26-28]. Its design was inspired by a proposal of Tick & Warren [68]. The PLM implements the WAM in microcode with a 100 ns clock cycle. It was built on wire-wrap boards and ran a few small programs in 1985. Spin-offs of this project included the VLSI-PLM single-chip implementation [60] and the Xenologic X-1, a commercial coprocessor for Sun workstations.

Several papers have compared the number of cycles needed by the PLM to that of general-purpose architectures. These ratios are valid measurements of the effect of the PLM's architectural support for WAM implementation. Mulder & Tick [51] and Patt & Chen [54] have compared the performance of the PLM [28], a microcoded implementation of the WAM, to a macro-expanded WAM on the MC68020 processor. They find that the MC68020 needs 3 to 4 times the number of cycles as the PLM to execute the WAM. Patt and Chen find that static code size on the MC68020 is about 20 times the PLM.

4.5.2.2. SPUR

Borriello et al [8] have implemented a macro-expanded WAM on the SPUR processor (Symbolic Processing Using RISCs). They find that the SPUR takes about 2.0 times the number of cycles as the PLM and that static code size is about 14 times the PLM. These numbers include local optimizations implemented by Chen and Nguyen [20] that improve the original numbers by about 10%.

4.5.2.3. PSI-II and PIM/p

In the context of the FGCS (Fifth Generation Computer System) project, researchers of ICOT (the Japanese Institute for New Generation Computer Technology) have designed and built several sequential and parallel architectures for logic programming [64, 71]. One of the more interesting sequential machines is the PSI-II (Personal Sequential Inference machine II) [52] a microcoded implementation of the WAM which executes at speeds similar to the PLM. The processing elements of the PIM/p (Parallel Inference Machine) architecture are currently the highest performance sequential logic machines at ICOT. They execute at two to three times the speed of the PLM.

4.5.2.4. KCM

Benker et al [6] describe a special-purpose Prolog machine, the KCM (Knowledge Crunching Machine), which is based on an extended WAM. Its instruction set consists of two parts: a general-purpose instruction set, and a microcoded Prolog-specific instruction set. It has a cycle time of 80 ns and executes in about 1/3 the number of cycles of the PLM. Its code size is about three times greater. The KCM project was done together with the development of a Prolog system and environment called SEPIA (see previous section). About 60 KCM machines were constructed and delivered to the ECRC member companies.

4.5.2.5. VLSI-BAM

Holmer et al [34] describe a single-chip microprocessor with extensions for Prolog, the VLSI-BAM (VLSI Berkeley Abstract Machine). It is a pipelined load-store processor with a cycle time of 33 ns. It takes about 1/3 the number of cycles to run programs as the PLM and its code size is about three times

greater, results similar to the KCM. However, they are achieved largely through the effort of the compiler. The goal of the BAM project is to find the minimal extensions to a general-purpose architecture to support a high performance Prolog implementation. The rationale for the VLSI-BAM architecture is that existing general-purpose architectures are designed to execute imperative languages like C and do not have adequate support for Prolog. The compiler described in this dissertation was developed simultaneously with the architecture, and interaction between the two designs has significantly improved both.

The BAM project has determined that a small amount of architectural support (5% increase in chip area) gives a large performance boost (50% performance increase) for programs that use Prolog-specific features. The support does not interfere with the general-purpose architecture, so it is possible for future general-purpose machines to incorporate this support for high performance symbolic computing. The support is designed specifically to support the logical variable, dynamic typing, unification, and backtracking. A language that uses any of these features can benefit from it.

Chapter 3

The Two Representation Languages

1. Introduction

This chapter defines the two languages used by the compiler to represent programs: kernel Prolog, a simplified form of Prolog, and the Berkeley Abstract Machine (BAM), a low-level instruction set and execution model that is close to a standard sequential processor. Kernel Prolog is an internal language that is not accessible to the user. BAM is the output language of the compiler.

2. Kernel Prolog

The first representation language in the compiler is kernel Prolog, a simplified, canonical form of Prolog. The syntax of kernel Prolog is given in Figure 3.1. This should be compared with the definition of full Prolog syntax given in Chapter 2. The control flow of kernel Prolog is simpler, a set of internal primitives is defined that are only used inside the compiler, and a case statement is defined. Kernel Prolog does not have nested disjunctions, if-then-else, cut, negation, or arithmetic expressions. Each predicate is represented as a single term `(H:-D)` containing a head H with distinct variable arguments and a body D that is a single disjunction (an OR choice). Each alternative of the disjunction is a conjunction, i.e. an AND sequence of goals. Unifications in the head of the original predicate are represented as explicit unifications in the arms of the disjunction. Disjunctions, negations, and if-then-else forms in the original predicate are converted into dummy predicates. Cut and arithmetic expressions are converted into simpler internal built-in predicates.

For example, the predicate:

```
a(b).
a(X) :- ( 0 is X mod 2 -> e(X) ; f(X) ).
```

is represented as follows in kernel Prolog:

```
predicate((H:-D)) :- head(H), disjunction(D).

head(H) :- goal_term(H).

disjunction(fail).
disjunction((C;D)) :- conjunction(C), disjunction(D).

conjunction(true).
conjunction((G,C)) :- goal(G), conjunction(C).

goal(G) :- case_goal(G).
goal(G) :- internal_goal(G).
goal(G) :- external_goal(G).

case_goal('$case'(Name,Ident,CB)) :- test_set(Name, Ident), case_body(CB).

case_body('$else'(D)) :- disjunction(D).
case_body(('$test'(T,D);CB)) :- test(T), disjunction(D), case_body(CB).

external_goal(G) :- goal_term(G), \+case_goal(G), \+internal_goal(G).

term(T) :- var(T).
term(T) :- goal_term(T).

goal_term(T) :- nonvar(T), functor(T, _, A), term_args(1, A, T).

term_args(I, A, _) :- I>A.
term_args(I, A, T) :- I=<A, arg(I, T, X), term(X), I1 is I+1, term_args(I1, A, T).

% Predicates defined in tables:
internal_goal(G) :- (Defined in Table 3.1).
test_set(Name, Ident) :- (Defined in Table 4.11).
test(T) :- (Defined in Table 4.11).

% Built-in predicates needed in the definition:
functor(T, F, A) :- (Term T has functor F and arity A).
arg(I, T, X) :- (Argument I of compound term T is X).
var(T) :- (Argument T is an unbound variable).
nonvar(T) :- (Argument T is a nonvariable).
```

Figure 3.1 – Syntax of kernel Prolog

```
a(X) :- ( X=b, true
        ; '$d'(X), true
        ; fail
        ).

'$d'(X) :- ( '$cut_load'(Z), '$d2'(X, Z), true
           ; fail
           ).

'$d2'(X, Z) :- ( '$mod'(X,2,0), '$cut'(Z), e(X), true
               ; f(X), true
               ; fail
               ).
```

All predicates that start with the character `'$'` are created internally. Cut is implemented with the two built-ins `'$cut_load'(X)` and `'$cut'(X)`. The arithmetic expression `0 is X mod 2` is replaced by a call to an explicit arithmetic built-in `'$mod'(X,2,0)`. The if-then-else is replaced by a call to the dummy predicate `'$d'(X)`. All dummy predicates are given unique names.

Kernel Prolog has many advantages over standard Prolog. The scope of variables is not limited to a single clause, but is extended over the whole predicate. Many optimizations are easier to do—for example, dataflow analysis and determinism extraction. Compilation to BAM code and register allocation are simplified.

The following two sections describe the internal predicates of kernel Prolog and how standard Prolog is converted to kernel Prolog.

2.1. Internal predicates of kernel Prolog

The kernel Prolog form of a program contains predicates that are not part of standard Prolog and that are invisible to the user. The internal predicates always begin with the character `'$'`. They are of three kinds:

(1) **Internal built-in predicates** (Table 3.1). These are classified into three categories depending on their use: (1) implementation of cut, (2) type checking, and (3) arithmetic. They are expanded into BAM instructions before being output, so the user never sees them.

(2) **A case statement**. This control structure is designed to express deterministic selection in Prolog. Chapter 4 describes how the case statement is created. It is translated directly into conditional

Table 3.1 – Internal built-ins of kernel Prolog

Built-in	Description
`'$cut_load'(X)`	Load the choice point register `r(b)` into X.
`'$cut'(X)`	Make the choice point pointed to by X the new top of the choice point stack.
`'$name_arity'(X,Na,Ar)`	Test that X has functor Na and arity Ar. This only does a check; it never binds X.
`'$test'(X,T)`	General type-checking predicate that tests whether the type of X is in the set T, where T $\subset$ {unbound variable, nil, non-nil atom, negative integer, nonnegative integer, float, cons, structure}.
`'$equal'(X,Y)`	Test that X and Y are identical simple terms.
`'$add'(S1,S2,D)`	Integer addition D $\leftarrow$ S1+S2.
`'$sub'(S1,S2,D)`	Integer subtraction D $\leftarrow$ S1–S2.
`'$mul'(S1,S2,D)`	Integer multiplication D $\leftarrow$ S1*S2.
`'$div'(S1,S2,D)`	Integer division D $\leftarrow$ S1/S2.
`'$mod'(S1,S2,D)`	Integer remainder D $\leftarrow$ S1 mod S2.
`'$and'(S1,S2,D)`	Bitwise integer ‘‘and’’ D $\leftarrow$ S1 $\wedge$ S2.
`'$or'(S1,S2,D)`	Bitwise integer ‘‘or’’ D $\leftarrow$ S1 $\vee$ S2.
`'$xor'(S1,S2,D)`	Bitwise integer exclusive-or D $\leftarrow$ S1 $\oplus$ S2.
`'$sll'(S1,S2,D)`	Logical left shift D $\leftarrow$ S1<<S2.
`'$sra'(S1,S2,D)`	Arithmetic right shift D $\leftarrow$ S1>>S2.
`'$not'(S,D)`	Bitwise integer negation D $\leftarrow$ not S.

branches in the BAM code and has the following syntax:

```
'$case'(Name,Ident,CaseBody)
```

where:

```
CaseBody = ( '$test'(Test,Code)
           ; ...
           ; '$else'(Code)
           ).
```

`CaseBody` is a disjunction of `'$test'` goals, terminated with an `'$else'` goal. `Code` is any valid kernel Prolog disjunction. `Name` and `Ident` identify the test set, and `Test` is a Prolog predicate (Table 4.11). `Test` is the test that is valid along the branch. For example, for the hashing function it will be the goal `X=a` where `a` is the atom or structure used in that direction.

(3) **‘‘Dummy’’ predicates**. Kernel Prolog does not allow control structures (i.e. disjunctions, if-then-else, and negation) in clauses, but only calls. The control structures are transformed into calls to dummy predicates, which are predicates that exist only inside the original predicate. Dummy predicates are created with unique names that are derived from the predicate they are contained in.

2.2. Converting standard Prolog to kernel Prolog

The first stage of compilation is a sequence of five source transformations that converts raw input clauses into kernel Prolog. An input predicate in standard Prolog is transformed into a tree that contains a kernel Prolog form of the original predicate and a set of dummy predicates in kernel form created during the transformation. Care is taken to put the predicate in a form that maximizes opportunities for determinism extraction. The five transformations are:

(1) **Standard form transformation**. Convert the raw Prolog input to a convenient standard notation. This does several housekeeping tasks: it properly terminates conjunctions (with `true`) and disjunctions (with `fail`), and it converts negation-as-failure into if-then-else.

(2) **Head unraveling**. Rewrite the head of each clause as a new head and a list of unification goals such that all the arguments of the new head are distinct variables and the head unifications are unification goals.

(3) **Arithmetic transformation**. Compile arithmetic expressions to internal arithmetic built-ins.

(4) **Cut transformation**. Implement cut by converting all uses of cut and if-then-else to internal cut built-ins.

(5) **Flattening**. At this point all complex control has been converted to disjunctions. Convert nested disjunctions to dummy predicates.

2.2.1. Standard form transformation

The standard form of a clause is intended to simplify its syntax so that traversing it is as simple as possible. The standard form satisfies the rules in Table 3.2. These rules are ignored in the presentation of most of the examples in this dissertation because they make the examples less readable (although they are always satisfied in the compiler).

2.2.2. Head unraveling

Unraveling the head of a clause consists of rewriting it as a new head and putting a series of unification goals in the clause's body so that all the head's arguments are distinct variables and all the head

Table 3.2 – Standard form of a clause	
Rule	Description
1	Conjunctions and disjunctions are right associative.
2	Conjunctions have no internal `true` and are terminated by `true`.
3	Disjunctions have no internal `fail` and are terminated by `fail`.
4	Single goals inside disjunctions are considered as conjunctions (and therefore rule 2 applies).
5	There is no negation (it is converted to if-then-else).
6	Arguments of if-then-else are considered as conjunctions (and therefore rule 2 applies).
7	(A->B) as a goal in a conjunction is converted to (A->B;`fail`).
8	The first argument of all unify goals is a variable.

unifications are unification goals in the body.

If this is not done correctly then much opportunity for later optimization is lost. From the predicate's type formula, the compiler knows which head arguments are nonvariable and which head arguments are unbound. Unification goals are created that satisfy two constraints:

(1) Maximize the number of nonvariable arguments that are unified together. Put these unifications first in the unraveled clause.

(2) Minimize the number of unification goals that contain unbound variables. Put these unifications last in the unraveled clause.

For example, consider the clause:

```
:-mode((a(A,B,C):-nonvar(A),nonvar(B),var(C))).
a(A,A,A) :- atomic(A), ...
```

The type declaration says that the first two arguments are nonvariables and the third argument is an unbound variable. The argument A appears three times in the head. Therefore there are three ways to unravel this clause: `(a(X,Y,Z):-X=Y,X=Z)`, `(a(X,Y,Z):-Y=X,Y=Z)`, and `(a(X,Y,Z):-Z=X,Z=Y)`. Considering the mode declaration, the head is transformed into the first of the three unraveled versions:

```
a(A,B,C) :- A=B, A=C, atomic(A), ...
```

The first unification `A=B` is of two nonvariables. The second unification `A=C` is of a nonvariable and an unbound variable. This satisfies both constraints.

```
expression((X is Expr), Code) :- expr(Expr, X, Code, []).

expr(V, V) --> {var(V)}, !.
expr(A, A) --> {integer(A)}, !.
expr(A+B, C) --> expr(A, Ta), expr(B, Tb), ['$add'(Ta,Tb,C)].
expr(A-B, C) --> expr(A, Ta), expr(B, Tb), ['$sub'(Ta,Tb,C)].
expr(A*B, C) --> expr(A, Ta), expr(B, Tb), ['$mul'(Ta,Tb,C)].
expr(A/B, C) --> expr(A, Ta), expr(B, Tb), ['$div'(Ta,Tb,C)].
```

Figure 3.2 – Compiling an arithmetic expression

2.2.3. Arithmetic transformation

The `is/2` predicate is translated into internal three-argument arithmetic built-ins (Table 3.1). Figure 3.2 gives a simplified but fully functional version of the algorithm used to compile expressions. It handles arbitrary expressions containing the four basic arithmetic operations. For example, the call:

```
expression(X is 23*(Y+Z), Code)
```

gives the code:

```
Code = ['$add'(Y,Z,T), '$mul'(23,T,X)]
```

The full algorithm handles all the arithmetic primitives of Table 3.1 and does partial constant folding.

2.2.4. Cut transformation

The cut operation modifies control flow by removing all choice points created since entering the predicate containing the cut, including the choice point of the predicate itself. Cut is implemented by means of a source transformation. It requires no support from the architecture except the ability to access and modify the register `r(b)`, which points to the most recent choice point.

The cut transformation is given in Figure 3.3. A call to the built-in `'$cut_load'(`*X*`)` is put at the entry of a predicate containing a cut. This built-in moves the `r(b)` register to *X*, which marks the top of the choice point stack on entry to the predicate. The argument *X* is passed to the predicate's body. Each occurrence of cut in the body is replaced by a call to the built-in `'$cut'(`*X*`)`. This built-in loads `r(b)`

---

procedure cut_transformation;
var P' : list of clause;
begin
  for each predicate P in the program **do begin**
    if P contains a cut **then begin**
      /* At this point P = [C_1 , ... , C_n] (list of clauses) and C_i = (H_i :- B_i) */
      Add the argument X to all H_i in P;
      Replace each occurrence of ‘‘!’’ in P by `'$cut'(`X`)`;
      P' := P;
      Add the predicate P' to the program;
      H := (new head with same functor and arity as all H_i);
      H' := (H with the additional argument X);
      P := [(H :- `'$cut_load'(`X`)`, H')]
    end
  end
end;

Figure 3.3 – The cut transformation

---

from X, which restores the original top of the choice point stack. For example, consider the predicate:

```
p :- q, !, r.
p :- s.
```

This is transformed into:

```
p :- '$cut_load'(X), p'(X).

p'(X) :- q, '$cut'(X), r.
p'(X) :- s.
```

Compilation then continues in the usual manner. This method is simple and efficient. Variations of it have been implemented in other Prolog systems [4, 13, 45]. This method differs from these variations in that the compiler does not always store the value of `r(b)` on the environment stack, but puts it in a predicate argument X. It is stored in an environment only if the clause is compiled with an environment.

2.2.5. Flattening

At this point, all the complex control in a predicate (disjunctions, if-then-else, and negation-as-failure) has been translated to disjunctions. Flattening replaces the disjunctions by calls to dummy predicates. For example, the definition:

```
a(X,Y) :- ( b1(X,A) ; b2(X,B),t(B) ), d(Y,A).
```

is transformed into:

```
a(X,Y) :- '$flatten_a/2_1'(X,A), d(Y,A).

'$flatten_a/2_1'(X,A) :- b1(X,A).
'$flatten_a/2_1'(X,A) :- b2(X,B), t(B).
```

Compilation then continues in the usual manner and the dummy predicate `'$flatten_a/2_1'(X,A)` is compiled as in-line code. The dummy predicate is created with a unique name derived from the name of the original predicate. The argument list of the dummy predicate is the intersection of the set of variables used inside the disjunction and the set of variables used outside it. In this example the argument list is the intersection of `{X,Y,A}` and `{X,A,B}`, which is `{X,A}`.

3. The Berkeley Abstract Machine (BAM)

The foundation of the efficiency of the compiler is its execution model, the BAM. The BAM has been designed to support all compiler optimizations and to make the system easily retargetable to the VLSI-BAM and general-purpose machines. The design evolved by interaction with the development of the compiler, the architecture design of the VLSI-BAM processor, and the requirement of portability to other architectures. The BAM was developed in tandem with the VLSI-BAM processor, but the two instruction sets are quite different. The VLSI-BAM is constrained by its hardware implementation; the BAM evolved by looking at the requirements of Prolog and is designed to allow a great deal of low-level optimization.

The Aquarius compiler uses a simple output language and not an existing high-level language such as C or an existing low-level language such as an assembly for a particular machine. There are several reasons for this:

(1) Choosing an existing language requires choosing representations for tags and data structures, and writing frequently used Prolog-specific operations as subroutines. This is undesirable for two reasons: First, the VLSI-BAM is one of the target machines and its architecture has a more abstract representation for tags and Prolog-specific operations than general-purpose processors. Second, these representations are not necessarily the best for all machines.

(2) Choosing an existing high-level language is unsatisfactory for the VLSI-BAM processor since the only compiler for it is currently the Aquarius compiler.

(3) An unpredictable factor is introduced when doing performance evaluations. The performance on different machines varies depending on the sophistication of the implementation of the existing language. It is not always easy to determine the performance of the existing language from inspection of its source code.

The syntax and semantics of the BAM is presented at several levels of detail, from a discussion of its features in English down to a detailed formal specification of its semantics in Prolog. The body of the dissertation defines the data types of the BAM, gives an overview of its instruction set, and justifies the choice of instructions. Appendices B and C give formal specifications of BAM syntax and semantics, and

Appendix D gives a concise but complete English description of BAM semantics.

This section has four parts. The first part presents the data types of the BAM. The second part summarizes the BAM instruction set. The instruction set consists of four parts: simple instructions (tagged load-store architecture), complex instructions (Prolog-specific operations), pragmas (embedded information to allow better translation to a real machine), and user instructions (intended to allow the complete run-time system to be written in BAM). The third part justifies the complex instructions. The fourth part justifies the instructions needed to implement unification by showing how they are constructed from a unification algorithm given a few simple assumptions about the architecture.

3.1. Data types in the BAM

The data types of the BAM are classified into two groups: the types used during execution and the types used to represent instructions (Table 3.3). The BAM has four data types that are used during execution: words, natural numbers, symbolic labels, and mappings. These are denoted as the set of all words **W**, the set of natural numbers **N**, the set of mappings **M**, and the set of symbolic labels **L**. A word is a pair `T^N` where T is the *tag* and N is the *value*. A natural number is a nonnegative integer. A mapping (not shown in Table 3.3) is a correspondence between a set of objects and their values (which are often words). A symbolic label marks a position in the program.

Several definitions in Table 3.3 require some clarification. Sets are denoted by bold capital letters, variables by capital letters, and constants by lower case letters. Addressing modes are defined recursively, with a base case consisting of registers and atomic terms, and a recursive case consisting of three parts: tag insertion (`T^X`), indirection (`[X]`), and offset (`(X+N)`). The BAM uses only a subset of the infinite set of addressing modes defined here. Of all the internal registers of the BAM, only the argument registers `r(I)`, the heap pointer `r(h)`, and the backtrack pointer `r(b)` are visible in the instruction set. Appendix B gives a precise definition of instruction syntax including the addressing modes that are actually used. The meaning of the instructions is defined informally in section 3.2 and formally in Appendix C.

A term can be of arbitrary size. A term that fits completely in a register is called *simple*. All other terms are called *compound*. A register cannot store all possible terms, but it can contain encoded informa-

Table 3.3 – Types in the BAM

Types used during execution		
Name		Definition
Word	**W**	= { `T^N` | T ∈ **T**$_p$ ∧ natural(N) } ∪ **A**
Symbolic label	**L**	= { `fail` } ∪ { `F/N` , `l(F/N,I)` | atom(F) ∧ natural(N) ∧ natural(I) }
Natural number	**N**	
Atomic term	**A**	= { `tatm^V` | atom(V) ∨ (V=(F/N) ∧ atom(F) ∧ natural(N))} ∪ { `V` | integer(V) } ∪ { `tflt^V` | float(V) }

Types used to represent instructions		
Name		Definition
Tag	**T**	= {`tvar`, `tlst`, `tstr`, `tatm`, `tint`, `tpos`, `tneg`, `tflt`} = **T**$_p$ ∪ **T**$_a$
Pointer tag	**T**$_p$	= {`tvar`, `tlst`, `tstr`}
Atomic tag	**T**$_a$	= {`tatm`, `tint`, `tpos`, `tneg`, `tflt`}
Condition	**C**	= {`eq`, `ne`, `lts`, `les`, `gts`, `ges`}
Equality condition	**C**$_e$	= {`eq`, `ne`}
Arithmetic operation	**E**	= {`add`, `sub`, `mul`, `div`, `mod`, `and`, `or`, `xor`, `sll`, `sra`}
State register	**R**$_s$	= {`r(h)`, `r(b)`, `r(e)`, `r(hb)`, `r(pc)`, `r(cp)`, `r(tmp_cp)`, `r(tr)`}
Argument register	**R**$_a$	= { `r(I)` | natural(I) }
Permanent register	**R**$_p$	= { `p(I)` | natural(I) }
Addressing mode	**X**	= **A** ∪ **R**$_a$ ∪ **R**$_p$ ∪ {`r(h)`, `r(b)`} ∪ { `T^X` | T ∈ **T**$_p$ ∧ X ∈ **X** } ∪ { `[X]` | X ∈ **X** } ∪ { `X+N` | X ∈ **X** ∧ natural(N) }
Instruction	**I**	(The set of BAM instructions is defined in section 3.2 and Appendix B)

tion about a term. The *tag* of a term stored in a register is the information about the term that is independent of the term's location in memory and can be obtained without doing a memory reference. The *value* of a term in a register tells where to find the rest of the term. A register is partitioned into two fields which contain the tag and the value of a term.

The encoding of information in tags is designed to simplify common operations. It is similar to the encoding used in the WAM (Figure 2.5). Atoms are represented as immediate values with a `tatm` tag. Integers are represented as themselves, and are considered to have `tint`, `tneg`, or `tpos` tags for the conditional branches that look at tags. Unbound variables are represented as pointers with a `tvar` tag that point to themselves or another unbound variable. Structures and lists are represented as pointers with tags `tstr` or `tlst`. They point to a contiguous block of their arguments on the heap. The main functor and arity of a structure are stored there encoded in a single word. The main functor and arity of a list (cons cell) are not stored since they are known implicitly.

The BAM defines five mappings to represent and access all data structures used during execution (Table 3.4). These mappings are the Register Set, the Heap, the Trail, the Code Space, and the Label Map. An infinite number of argument and permanent registers is assumed to exist. Of all registers, only the heap

Table 3.4 – Run-time data structures of the BAM	
Name	Definition
Register Set	$(\mathbf{R}_s \cup \mathbf{R}_a \cup \mathbf{R}_p) \rightarrow \mathbf{W}$
Heap	$\mathbf{W} \rightarrow \mathbf{W}$
Trail	$\mathbf{N} \rightarrow \mathbf{W}$
Code Space	$\mathbf{N} \rightarrow \mathbf{I}$
Label Map	$\mathbf{L} \rightarrow \mathbf{N}$

pointer `r(h)` and the backtrack pointer `r(b)` are made explicit in the instruction set. The others are implicit in its execution. Environments and choice points are represented as register sets that are stored in registers `r(e)` and `r(b)`, respectively. Prolog terms are stored in registers, on the heap, and on the trail. Compound terms are stored on the heap as sequences of words in the same manner as is done in the WAM (Figure 2.5). For all types except atoms, the value field of a word is a natural number that indexes into the heap, and therefore points to terms on the heap. For atoms, the value field is the symbolic atom itself. The correspondence between tags and Prolog data types is given in Table 3.5.

Table 3.5 – Correspondence of tags with Prolog data types	
Tag	Data type
`tvar`	An unbound variable or a general pointer.
`tstr`	Pointer to a structure—a compound term with a functor and fixed number of arguments.
`tlst`	Pointer to a cons cell—a compound term consisting of two parts, a head and a tail.
`tatm`	An atom.
`tpos`	A nonnegative integer.
`tneg`	A negative integer.
`tint`	An integer.
`tflt`	A floating point number.

The following descriptions clarify the correspondence between BAM types and Prolog types:

(1) The value corresponding to a pointer tag is an index into an array of words. This is normally implemented as an address.

(2) The value corresponding to a `tatm` tag is a symbol that uniquely identifies an atom or the main functor of a structure. It is a Prolog atom or a Prolog structure of the form F/N where F is a Prolog atom representing the functor and N is a nonnegative integer representing the arity. For correctness, the assembler and run-time system must guarantee an exact correspondence between this symbol and the contents of the run-time symbol table, so that the built-ins `name/2`, `functor/3`, `arg/3`, and `=../2` all work correctly.

(3) The value corresponding to a `tpos` or `tneg` tag is a nonnegative integer that represents the absolute value of the integer represented by the word.

(4) The value corresponding to a `tint` tag is an integer that represents the value of the integer represented by the word.

(5) The value corresponding to a `tflt` tag is a floating point number that represents the value of the number represented by the word.

Nothing is assumed about how these types are represented on a real machine. When the BAM is targeted to a real machine then the representation of types on the machine must be defined. The representation of types changes with different target machines, different versions of the system, and even different programs. The Implementation Manual [31] discusses how to port the BAM. Symbolic labels are pointers to code. Since mappings can be of any size, they are pointers to data stacks in memory. The representation of a word depends on the encoding used to represent tags on the machine, the word size of the machine, and on the encoding of Prolog atoms into unique bit patterns. For the VLSI-BAM processor, all four types are mapped into 32 bits and words consist of 4 bit tags and 28 bit values.

Table 3.6 – Notation for arguments of BAM instructions

Argument	Type
`X, Y, Z`	Addressing modes, elements of **X**. Most instructions use a subset of all possible addressing modes.
`L, L1, L2, L3`	Branch destinations, elements of **L**.
`N`	A natural number, element of **N**.
`A`	A Prolog atom, element of **A**.
`Tag`	A tag value, element of **T**.
`Eq`	An equality condition, element of $\mathbf{C}_e$.
`Cond`	A condition, element of **C**.
`Op`	An arithmetic operation, element of **E**.
`RegList`	A list of registers used in choice point management. `RegList` $\in \{ [\alpha_0, \alpha_1, ..., \alpha_n] \mid n \in \mathbf{N}, \alpha_i \in \{i, \text{no}\} \}$.

3.2. An overview of the BAM

The BAM uses types and data structures similar to the WAM. It has registers and stacks similar to the WAM and uses a similar execution strategy. However, the instruction set is completely different. The BAM has a load-store instruction set that is extended with tagged addressing modes and a few primitive Prolog-specific instructions. A summary of the addressing modes and instructions is given in Tables 3.6

through 3.10. All instructions use only a subset of the addressing modes given in Table 3.3. The instruction set includes:

- **Simple instructions** (Table 3.7). These are simple register-transfer level operations for a tagged architecture. They include move, push, conditional branch, and arithmetic. These instructions are used to implement many cases of unification and many built-in predicates.
- **Complex instructions** (Table 3.8). There are five frequently-used operations defined as single instructions: dereferencing (following a pointer chain to its end), trailing (saving a variable's address so it can be restored on backtracking), general unification (when the compiler cannot simplify the general case), choice point handling (saving and restoring state for backtracking), and environment handling (creating and removing local stack frames).
- **Embedded information** (Table 3.9). This allows a better translation to the assembly language of the target machine. This information is expressed in two ways: (1) with pragmas, which resemble instructions but are not executable, and (2) by extending instructions with additional arguments. An example of (1) is the tag pragma, which gives the tag of a load or a store, e.g.:

```
pragma(tag(r(1),tvar)). % Register r(1) contains a tvar tag.
move([r(1)],r(0)).      % Load register r(0) from register r(1).
```

  By giving the tag at compile-time, this avoids tag masking on a general-purpose processor and allows the load to be done in a single cycle. An example of (2) is:

```
unify(r(0),r(1),?,nonvar,fail). % Register r(1) is nonvariable.
```

  This gives no information about `r(0)` but says that `r(1)` is nonvariable. This allows the unification to be done more efficiently because no check has to be done whether `r(1)` is unbound.
- **User instructions** (Table 3.10). The BAM language is extended with several instructions, registers, and tags that are never output by the compiler, but are intended for use only by a BAM assembly programmer. This allows the non-Prolog component of the run-time system to be written completely in BAM assembly. These instructions are described in Appendix D.

Table 3.7 – Simple instructions

Instruction	Meaning
`equal(X,Y,L)`	Branch to L if X and Y are not equal.
`move(X,Y)`	Move X to Y.
`push(X,Y,N)`	Push X on stack with stack pointer Y and post-increment N.
`Op(X,Y,Z)`	Perform the arithmetic operation Op on X and Y and store the result in Z. Trap if an operand or the result is not integer.
`adda(X,Y,Z)`	Full-word non-trapping add of a word X and an offset Y, giving a word Z.
`pad(N)`	Add N to the heap pointer.
`switch(Tag,X,L1,L2,L3)`	Three-way branch; branch to L1, L2, L3 depending on whether the tag of X is `tvar`, Tag, or any other value.
`test(Eq,Tag,X,L)`	Branch to L if the tag of X is equal or not equal to Tag.
`hash(T,X,N,L)`	Look up X in a hash table of length N located at L. If X is in the table then branch to the label in the table, else fall through. T ∈ {`atomic`, `structure`}.
`pair(E,L)`	A hash table entry. E is either an atom or a pair functor/arity.
`jump(Cond,X,Y,L)`	Jump to L if the arithmetic comparison of X and Y is true. Trap if an operand is not integer.
`jump(L)`	Jump unconditionally to L.
`label(L)`	L is a branch destination.
`procedure(Name/Arity)`	Mark the beginning of a procedure.
`call(Name/Arity)`	Call the procedure Name/Arity.
`jump(Name/Arity)`	Jump to the procedure Name/Arity.
`return`	Return from a procedure call.
`simple_call(Name/Arity)`	Non-nestable call used to interface with routines written in BAM assembly.
`simple_return`	Non-nestable return used for routines written in BAM assembly.

3.3. Justification of the complex instructions

The execution of Prolog requires five complex operations: dereferencing, trailing, unification, backtracking, and environment management. These operations are represented as single instructions in the BAM. In the WAM, dereferencing, trailing, and unification are done implicitly by many instructions even when they are not needed. Making them explicit allows the compiler to minimize their use as much as possible by doing them only when they are really needed.

The complex instructions could be expanded into sequences of simple instructions; however, this expansion is not done at the BAM level but is delayed to the machine level. There are two reasons for this:

(1) Some machines may implement part or all of a complex instruction directly. Expanding it into simple instructions is therefore premature since it would make this harder to detect. For example, the VLSI-BAM processor has support for some complex instructions (e.g. dereferencing, trailing, and unification).

Table 3.8 – Complex instructions

Instruction	Meaning
`deref(X,Y)`	Dereference X and store result in Y.
`trail(X)`	Push X on the trail stack if the trail condition is satisfied.
`unify(X,Y,Tx,Ty,L)`	General unification of X and Y, branch to L if fail. Trailing is done by this instruction. The extra parameters `Tx`, `Ty` ∈ {`?`, `var`, `nonvar`} give information to improve the translation. They are not needed for correctness.
`unify_atomic(X,A,L)`	Unify X with the atom A and branch to L if fail. No trailing is done by this instruction.
`allocate(N)`	Create an environment of size N on the local stack.
`deallocate(N)`	Remove the top-most environment from the local stack.
`choice(1/N,RegList,L)`	Create a choice point containing the registers listed in `RegList` and set the retry address to L.
`choice(I/N,RegList,L)` (1<I<N)	Restore the argument registers listed in `RegList` from the current choice point, and modify the retry address to L.
`choice(N/N,RegList,fail)`	Restore the argument registers listed in `RegList` from the current choice point, and pop the current choice point from the choice point stack.
`fail`	Restore the machine state (except the argument registers) from the most recent choice point, restore to unbound all variables on the trail that were bound and trailed since the creation of this choice point, and transfer control to the retry address.
`move(r(b),X)`	Move the backtrack pointer to X. This must be done at the entry of any predicate containing a cut.
`cut(X)`	Make the choice point pointed to by X the new top of the choice point stack.

Table 3.9 – Embedded information (pragmas)

Instruction	Meaning
`pragma(align(X,N))`	The contents of location X are a multiple of N.
`pragma(tag(X,Tag))`	The contents of location X have tag Tag.
`pragma(push(term(N)))`	A term of size N is about to be created on the heap.
`pragma(push(cons))`	A cons cell is about to be created on the heap.
`pragma(push(structure(N)))`	A structure of arity N is about to be created on the heap.
`pragma(push(variable))`	An unbound variable is about to be created on the heap.
`pragma(hash_length(N))`	A hash table of length N is about to be created.

(2) For best performance, optimizations should be done at all levels. The BAM level makes certain optimizations easy, e.g. the determinism optimization in Chapter 6. Keeping the complex operations as single instructions allows them to be optimized directly. For example, if a variable is dereferenced twice then the second dereference can be removed. This is much harder to detect if the dereference instruction is expanded into a loop.

It is best to avoid assumptions about the characteristics of the target machine. In the cases where such assumptions would be useful, the BAM uses pragmas to give the information without compromising the

Table 3.10 – User instructions

Instruction	Meaning
`ord(X,Y)`	Extract the value of X and move it to Y.
`val(T,X,Y)`	Create the word Y from the tag T and the value X.
`jump_reg(R)`	Jump to address stored in register R.
`jump_nt(Cond,X,Y,L)`	Jump to L if the full word comparison of X and Y is true. Never trap.
`Op_nt(X,Y,Z)`	Perform the full word arithmetic operation Op (except multiply and divide) on X and Y and store the result in Z. Never trap.
`trail_bda(X)`	Push address X and the value stored there on the trail stack if the trail condition is satisfied. This is a special trail instruction for backtrackable destructive assignment.

machine independence. The translator is free to use or ignore this information.

3.4. Justification of the instructions needed for unification

This section constructs the BAM instructions that contain the required instructions and addressing modes to support unification. It turns out that both simple and complex instructions are necessary to support unification. The instructions are constructed starting from an algorithm for unification and a very general intermediate language. The algorithm is decomposed into specialized instructions depending on the form of the data known at compile-time.

The two starting points are (1) an algorithm for unification (a specification of a unification algorithm is given in Appendix C), and (2) a very general instruction set. The method proceeds in a top-down manner by decomposing the unification algorithm into specialized instructions depending on information about the form of the data known at compile-time (Figure 3.4).

This method is inspired by Kursawe [41] and Holmer [32]. Kursawe applies partial evaluation and specialization in a top-down manner starting from a Prolog program and obtains an instruction set resembling the WAM. Holmer describes several techniques for the automatic design of instruction sets, of which decomposition is one. To go beyond the WAM it is necessary to make assumptions about the architecture, a step that Kursawe does not take. The design of the BAM starts with a general instruction set that does make these assumptions.

The choice of what general instruction set to start with is important. It is not useful to start with an instruction set that has too little expressive power, for example one with a limited set of addressing modes,

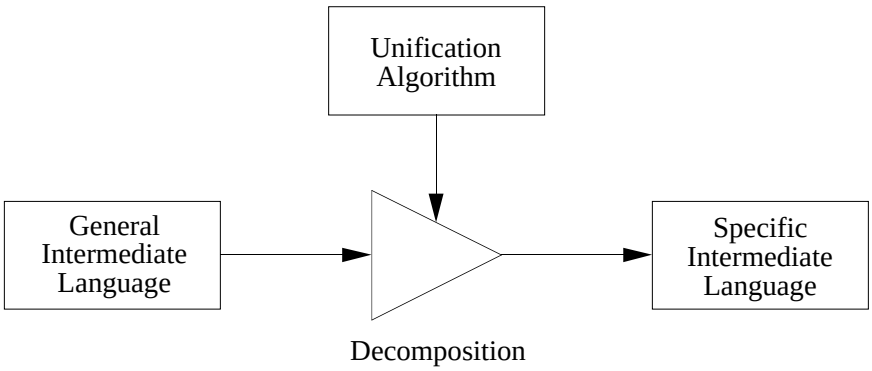

Figure 3.4 – Decomposition of unification

because the required addressing modes are not yet known. Prematurely decomposing complex instructions into simple ones side-steps the results.

The following assumptions are made:

(1) The architecture is sequential and of Von Neumann design with multiple registers.

(2) The basic data element is a *word*, which is large enough to contain an address. A register holds one word.

(3) The instructions have three parts:

- An action. Some sample actions are data movement (`move`, `push`), conditional branching (`equal`), and general unification (`unify`). Other important actions are multi-way branching (`switch`) and several Prolog-specific operations (`deref`, `trail`).
- A set of arguments. Unification acts on two operands, so typically two arguments are sufficient.
- A set of destination addresses. Depending on the outcome of the action, control continues at one of the destinations. The size of the set and the meaning of its members depends on the action. The address of the next instruction in the instruction stream is an implicit member of

the set.

(4) Arguments are referenced with multiple addressing modes. An infinite set of addressing modes are defined in Table 3.3. The instructions derived in this section will need only finite subset. For clarity, Table 3.11 gives some abbreviations useful for this subset.

Table 3.11 – Useful abbreviations

Notation	Meaning
`Disp`	a positive heap displacement (bounded by the size of a term).
`Offset`	a nonnegative offset into a structure (bounded by the arity).
`Imm`	an immediate value; an atom or a numeric constant.
`Var`	a variable local to a clause, i.e. `r(I)` or `p(J)`.
`Arg`	denotes `Var` or `[Var+Offset]`.

Construction of the instruction set proceeds in the following steps. The data representation has already been fixed (section 3.1). The existence of two forms of unification (read mode and write mode) and the need for dereferencing and a three-way branch is shown. The instructions required for read mode and write mode are constructed. Finally, the effects of variable representation (in registers or on the environment) on the instruction set are discussed.

3.4.1. The existence of read mode and write mode

The compilation of the unification $T_1 = T_2$, where T_1 and T_2 are two arbitrary terms, is reduced to the compilation of $V = T$ where at compile-time V is a variable and T is any term. At run-time there are two values of V that result in different actions of the unification algorithm:

(1) V is an unbound variable, in which case T is constructed on the fly and bound to V (this is called *write mode*). To satisfy the standard definition of unification, when T is bound to V a check needs to be done (the *occur check*) that T does not contain V. Following Prolog implementation convention, this check is ignored for efficiency reasons.

(2) V is a nonvariable term, in which case it is checked that the form of V matches T, and the algorithm is invoked recursively for the term's arguments (this is called *read mode*).

3.4.2. The need for dereferencing

Unifying two unbound variables makes one point to the other. Doing this several times leads to pointer chains, with the common value of all the variables in a single location at the end of the chain. To get a variable's value, the pointer chain is followed to its end, an operation known as *dereferencing*. It can be provided as an addressing mode or as a separate instruction. Making it an instruction avoids repeated dereferencing. Therefore the following instruction is added:

```
deref(Var1,Var2)
```

First `Var1` is moved to `Var2`. Then the tag of `Var2` is checked. If it is an unbound variable (`tvar`) it reads memory and a loop is entered replacing `Var2` by the referenced value while its tag is `tvar` and its pointer part is different from `Var2`. A two-argument dereference is chosen over a single-argument dereference because it allows a more compact representation of write-once variables (Chapter 5).

It is assumed in what follows that V and T are dereferenced when necessary, in particular that both the `trail` and `unify` instructions are always given dereferenced arguments.

3.4.3. The need for a three-way branch

The code for a unification $V = T$ consists of three parts: (1) a check whether V is an unbound variable or a nonvariable for choosing between write mode and read mode unification, (2) the instructions for read mode unification, and (3) the instructions for write mode unification.

The tag field is available directly for the check of (1). The check has three possible results: the tag of V matches a known tag (read mode), the tag is an unbound variable tag (write mode), or the tag is neither (failure). This implies the following three-way branch:

```
switch(Tag, Var, VarLbl, TagLbl, FailLbl)
```

If the tag of `Var` is `tvar` (an unbound variable) then jump to `VarLbl`. If the tag of `Var` matches `Tag` then jump to `TagLbl`. Otherwise jump to `FailLbl`. The failure address is explicit instead of implicit to allow the implementation of fast incremental shallow backtracking.

3.4.4. Constructing the read mode instructions

The general case of read mode unification is $V = T$, where at compile-time V is a variable or an argument of a compound term, and T is a term. The first argument of each instruction is the value of V. Two locations are possible for its value:

```
Var             V is a variable
[Var+Offset]    V is an argument of a compound term
```

The abbreviation `Arg` is used to denote one of these two addressing modes (Table 3.11). The second argument and the action are determined by the compile-time knowledge of T. The possibilities are:

(1) T is partially or wholly known at compile-time. The possible information known about T is:

- T is an unbound variable that has not yet been initialized, e.g. because it is the first occurrence in the clause. V is moved directly to T.
- T is an unbound variable. V is stored to T's location in memory.
- T is atomic. Unification reduces to a check that T and V have the same atomic value. If the values do not match the unification fails.
- T is compound. Unification reduces to a check that V has the correct functor and arity, followed by a unification of its arguments with T's arguments. If V's arguments are loaded into registers then the unification can be compiled recursively. It follows that arbitrarily deep nesting of addressing modes is not necessary if one instruction is added:

```
move([Var+Offset], Var)
```

(2) Nothing is known about T at compile-time. The unification of V and T requires a general unification.

The following table of primitive instructions summarizes the action and both arguments:

Action	Argument *V*	Argument *T*	Explanation
`move`	`Arg`	`Var`	*T* is an unbound variable that has not yet been initialized.
`move`	`Arg`	`[Var]`	*T* is an unbound variable that has been initialized.
`equal`	`Arg`	`Var`	*T* is atomic or compound and its main functor is not known at compile-time.
`equal`	`Arg`	`Tag^Imm`	*T* is atomic or compound and its main functor is known at compile-time.
`unify`	`Arg`	`Var`	Nothing is known about *T* at compile-time.

The instructions `equal` and `unify` both can fail, so they have a failure address as third argument. The `equal` instruction compares its arguments and jumps to `FailLbl` if they are not equal.

General unification (`unify`) is the most complex instruction. If the unification fails it jumps to `FailLbl`. This instruction can be implemented using only the other instructions. However, it seems that one additional instruction is useful: a multi-way branch with a different destination for each possible tag value. If there are many possible tags this implies the existence of a jump table in memory, so that the instruction must do a memory reference before it can branch. Instead of using this instruction, another approach is to use a multilevel tree based on the three-way branch. Both approaches are viable since general unification is used rarely in real programs. According to measurements done by Holmer for several large programs [33], general unification takes about 4% of the total execution time of the VLSI-PLM [61]. More than 95% of these calls have arguments that are not compound terms of the same type and therefore do not need the recursive algorithm.

3.4.5. Constructing the write mode instructions

The general case of write mode unification is $V = T$, where V is known to be an unbound variable at run-time and T is a term. Assume that the term T is created on a stack (called the *heap*) with a minimal number of move instructions. This assumption forces us to derive the form that a compound term has on the heap. The following are the possible values of words of a compound term:

```
Var               a variable (assumed initialized)
Tag^Imm           a simple subterm of T
Tag^(r(h)-Disp)   a pointer to a compound subterm of T
```

These are the source addressing modes for the move instructions. A variable `Var` does not have to be

dereferenced when it is stored on the heap because its value is not read. The destination of the move instruction is a location on the heap. This location can be addressed either by a displacement addressing mode offset from the heap pointer `r(h)`, i.e. `[r(h)-Disp]`, or by an auto-increment addressing mode, i.e. a push instruction. The BAM uses the auto-increment addressing mode, for these reasons:

(1) Preliminary studies using exhaustive search [32] show that with the VLSI-BAM microarchitecture the optimal way to create structures in write mode is by means of the idiom ‘‘load register, load register, double-word push’’, i.e. two registers are loaded and then pushed in a single instruction.

(2) Instruction encoding is compacter, i.e. a push does not need a displacement field.

(3) In the VLSI-BAM architecture the push instruction is given a displacement field anyway. This allows efficient implementation of uninitialized variables. For example, a cons cell whose cdr is uninitialized can be created with a single push that has a displacement of 2.

(4) In the VLSI-BAM architecture the use of a push instruction allows a cache optimization: when pushing a dirty line it is not necessary to flush the line first [17]. This optimization was first done in the PSI-II architecture [52].

To summarize, to create a term on the heap it is sufficient to choose from the following set of three instructions (where `r(h)` is the stack pointer and `1` is the increment):

```
push(Var, r(h), 1)
push(Tagˆ Imm, r(h), 1)
push(Tagˆ (r(h)-Disp), r(h), 1)
```

It is also necessary to bind the term to *V*. This requires us to consider the form an unbound variable can take. There are two possibilities:

(1) *V* has not yet been initialized, e.g. because it is the first occurrence in the clause. The term is moved directly to *V*.

(2) *V* has been initialized; it points to a location in memory. The term is stored in this location.

These two possibilities result in the following two instructions:

```
move(A, Var)      store directly to a variable
                  (variable is not initialized)
move(A, [Var])    store to variable's location
                  (variable is initialized)
```

The addressing mode of the argument `A` depends on whether the term is compound or simple, and if it is simple, whether it is an atom or a variable. This results in three possible values for `A`:

```
Var               a simple term (variable)
Tag^Imm           a simple term (nonvariable)
Tag^r(h)          a compound term (on the heap)
```

In addition to the above instructions, it is also necessary to initialize the first occurrence of a variable. One way to do this is:

```
move(tvar^(r(h)-Disp), Var)
push(Var, r(h), 1)
```

With these instructions it is possible to create a term of size n on the heap in n pushes, a great improvement over the WAM, which requires $n+f-1$ stores, $f-1$ dereference operations, and $f-1$ trail checks, where f is the number of functors in the term. This idea was first proposed by André Mariën [44].

3.4.6. Representation of variables

Assume that the execution model represents variables local to a clause in an environment, or stack frame. There is a dedicated register `r(e)`, called the *environment pointer*, that points to the current environment in the environment stack. Variables local to a clause are stored either in registers or in an environment, so the notation `Var` denotes one of the following two addressing modes:

```
r(I)              a variable in a register
p(J)              a variable on the environment stack
```

where `p(J)` is implemented as an offset into the environment, i.e. as `[r(e)+J']` for some `J'`. This implies that double indirection is possible: the addressing mode `[Var+Offset]` is `[p(J)+Offset]` when `Var` is an environment variable. The double indirection is avoided by including one instruction:

```
move(p(J), r(I))
```

Table 3.12 – Data movement instructions for unification

Read mode	Write mode
`move(Arg, Var)`	`push(Var, r(h), 1)`
`move(Arg, [Var])`	`push(Tagˆ Imm, r(h), 1)`
	`push(Tagˆ(r(h)-Disp1), r(h), 1)`
`equal(Arg, Var, F)`	
`equal(Arg, Tagˆ Imm, F)`	`move(Var1, Var2)`
	`move(Tagˆ Imm, Var)`
`unify(Arg, Var, F)`	`move(Tagˆ(r(h)-Disp), Var)`
	`move(Tagˆ r(h), Var)`
	`move(Var1, [Var2])`
	`move(Tagˆ Imm, [Var])`
	`move(Tagˆ r(h), [Var])`

Table 3.13 – Control flow and other instructions for unification

`switch(Tag, Var, VarLbl, TagLbl, F)`	three-way branch
`jump(Lbl)`	join read and write mode paths
`deref(Var1, Var2)`	dereference a pointer chain

3.4.7. Summary of the unification instructions

This section summarizes the BAM instructions necessary to support unification. Tables 3.12 and 3.13 present the instructions. They use only a small finite subset of the addressing modes of Table 3.3. The following typical instructions illustrate the meaning of the notation:

`move(tatmˆ axe,r(3))`	Move the atom `axe` into register `r(3)`.
`move([r(3)+5],r(4))`	Move the word located at address `r(3)+5` into `r(4)`.
`equal(r(2),tatmˆ cat,F)`	If `r(2)` is equal to the atom `cat` then fall through, else jump to label `F`.
`unify(p(2),p(3),F)`	Unify the term located in `p(2)` with the term located in `p(3)`. Jump to label `F` if the unification fails.
`switch(tatm,r(3),V,T,F)`	If `r(3)`'s tag is `tvar` then jump to label V. If `r(3)`'s tag is `tatm` then jump to label T. Otherwise, jump to label F.

Chapter 4

Kernel transformations

1. Introduction

Four optimizing transformations are done on the kernel Prolog representation of programs: formula manipulation, factoring, global dataflow analysis, and determinism extraction. The goal of the transformations is to reduce a single metric: The total execution time of all unifications in the program. This metric is approximated by the number of unifications and by the size of the terms being unified. The chapter first describes the representation of types as logical formulas in the compiler. This is followed by a description of each of the four transformations:

(1) **Formula manipulation**. The compiler implements a set of primitive transformations to replace Prolog code and types (both are represented as logical formulas) with simpler versions that have identical semantics. The simplicity of a formula is defined as the number of goals in the formula. These transformations are done whenever there is a possibility that the code is too complex, i.e. upon reading in a program and after other transformations such as the determinism transformation (see below).

(2) **Factoring**. This transformation groups sets of clauses in a predicate together if they have head unifications in common. This reduces the number of head unifications and shallow backtracking steps.

(3) **Global dataflow analysis**. This stage analyzes the program, annotates it with types, and restructures it. The analyzer uses abstract interpretation to determine the types of predicate arguments.

(4) **Determinism transformation**. This stage rewrites the program to make its determinism explicit, i.e. it replaces shallow backtracking by conditional branching. Many of the other transformations in this chapter are chosen to make this transformation possible more often. The transformation converts the predicate into a series of nested case statements. Sometimes this is only partially successful; certain branches of the case statements may still retain disjunctions (OR choices) that could not be converted into deterministic code.

To improve readability, the examples in this chapter are given in standard Prolog notation. It is understood that they are represented internally in kernel Prolog.

2. Types as logical formulas

Throughout the compiler, type information about variables is represented with logical formulas. During compilation, any information learned is added to the formula, and deduction based on the formula simplifies the generated code. It is a simple and powerful approach to avoid doing redundant operations at run-time. For example, if a variable is dereferenced once, then it should never be dereferenced again. Types in the compiler are defined as follows:

> **Definition T**: Given a predicate f/n with main functor f and arity n, a *type* of f/n is a term ($f(A_1, A_2, \cdots, A_n)$:- `Formula`) where the $A_1, A_2, \cdots, A_n$ are n distinct variables and `Formula` is a logical formula (i.e. a Prolog term).

For example, the type `(range(A,B,C):-integer(A),var(B),integer(C))` says that the first and third arguments of `range/3` are integers and the second argument is an unbound variable. The compiler recognizes all Prolog type-checking predicates in the type formula. Appendix A gives a table of the types recognized by the compiler. In addition to these types, several other types are recognized that do not correspond to Prolog predicates. These types introduce distinctions between objects that depend on the implementation and are indistinguishable in the language, for example, the difference between an integer and a dereferenced integer, and the difference between an unbound variable that is not aliased to any other and an unbound variable that may be aliased. The following types are recognized that do not exist as Prolog predicates:

Internal Type	Description
`uninit(X)`	X is an uninitialized memory argument.
`uninit_mem(X)`	X is an uninitialized memory argument.
`uninit_reg(X)`	X is an uninitialized register argument.
`unbound(X)`	X is of one of the types `uninit_mem(X)`, `uninit_reg(X)`, or `var(X)`.
`deref(X)`	X is dereferenced, i.e. it is accessible without following any pointers.
`rderef(X)`	X is recursively dereferenced, i.e. it is dereferenced, and if it is compound then all its arguments are recursively dereferenced.

These types should not be given by the programmer since incorrect code or a significant loss of efficiency may result if they are used incorrectly. For example, declaring an argument of a predicate to be of uninitialized register type, i.e. the argument is an output that is passed in a register, may cause a large increase in stack space used by the program if that predicate is the last goal in a clause, because last call optimization is not possible in that instance. The safe approach is to leave the use of these types up to the compiler.

The use of logical formulas to hold information during compilation can be contrasted with the use of a symbol table in a compiler for an imperative language.† Representing types as logical formulas has two advantages over a symbol table: (1) They are more flexible during compiler development. The kind of information stored in a symbol table must be known when the compiler is designed. Formulas can contain kinds of information that are not known during the compiler's design. (2) They lend themselves to powerful symbolic manipulation such as deduction. Improving the deductive abilities leads to better code without having to change any other part of the compiler. The disadvantage of this representation is that its manipulation is slow. Future versions of the compiler could use a representation that is faster in the common cases.

Type formulas are used in the following ways in the compiler:

(1) Representing type information known about a set of variables. For example, the formula `(var(X),atom(Y))` means that X is an unbound variable and Y is an atom. The user manual (Appendix A) lists the types recognized by the compiler.

(2) Using a primitive form of deduction to simplify the generated code. For example, assume the formula is `(list(X),var(Y),deref(Z),...)`. To compile a run-time check that X is a non-variable, the compiler first checks whether this formula implies `nonvar(X)`. This is true because `list(X)` implies `nonvar(X)`, so no run-time check is necessary.

(3) Updating the type formula when new information is learned. After compiling a goal, the formula is updated to represent the new knowledge that is gained. For example, after executing the arithmetic

† Of course, both the assembler and the run-time system use standard symbol tables.

expression `X is A+B` it is known that X is an integer, so the formula is extended with `integer(X)`.

In most cases, logical formulas are immutable, e.g. when a variable X is known to be a list (represented as `list(X)`), that fact remains true forever. This is not true for all types. The types used to denote unbound variables (e.g. `var(X)` and `uninit(X)`) become false as a result of an instantiation. This is also true of the standard order comparisons (e.g. `X@<Y`, `X@>Y`, and so forth) and the types `deref(X)` and `rderef(X)`. The compiler is careful to take this into account when updating the type formula.

Table 4.1 – Primitives to manipulate logical formulas and Prolog formulas

Primitive	Description
F_1 implies F_2	Implication: Succeeds if it can determine that there does not exist an assignment to variables in F_1 and F_2 that causes both F_1 and not(F_2) to succeed.
F_2 := simplify(F_1)	F_2 is a simplification of F_1.
F_2 := subsume(F, F_1)	F_2 is a simplification of F_1, given that F is true.
F_2 := update_formula(F, F_1)	F_2 is the result of removing information contradicted by F from F_1 and adding F to F_1.

3. Formula manipulation

The compiler implements a set of primitive transformations to manipulate formulas. They are summarized in Table 4.1, where F, F_1, and F_2 are logical formulas. Each of these primitives has two versions: a pure logical and a Prolog version. The logical version is used to manipulate types (see previous section). It assumes the formula has a purely logical meaning, i.e. that the operational concepts of execution order of goals, number of solutions, and backtracking behavior are not important. The Prolog version is used to manipulate kernel Prolog code. It assumes the formula must keep Prolog's operational semantics.

Implication is implemented to work well with most combinations of Prolog predicates that are used in type declarations. The following examples all return with success:

Table 4.2 – Examples of simplification			
Formula	Simplified formula		Comments
	logical	Prolog	
`(true ; true)`	`true`	`(true ; true)`	The Prolog version is unchanged unless the compiler option `same_number_solutions` is disabled.
`(p,fail)`	`fail`	`(p,fail)`	The Prolog version is unchanged unless the compiler can deduce that `p` has no side effects (read / write or assert / retract).
`(!,p ; q)`	`(p ; q)`	`(!,p)`	Cut is logically identical to `true`, but it must be retained since it modifies backtrack behavior in the entire clause containing it.

`atom(X)` implies `nonvar(X)`
`X<Y` implies `integer(X)`
`X<5` implies `X<10`
`uninit(X)` implies `deref(X)`
`functor(X, _, 0)` implies `atomic(X)`
`(X==a; X==b)` implies `atom(X)`

Simplification is done on standard Prolog, on kernel Prolog, and on type formulas. Table 4.2 gives some examples to illustrate the difference between logical and Prolog semantics. A single function simplify(F) handles both logical and Prolog semantics (Figure 4.1). For conciseness, the definition of simplify(F) uses the compound terms `(A,B)`, `(A;B)`, `(A->B)`, and `(\+(A))` both as selectors (to choose the branch of the case statement) and constructors (in the calls to simp_step(F)). Tables 4.3 and 4.4 define part of the definition of simp_step(F), the primitive simplification step. The complete definition contains about 50 rules. The functions subsume(F, F_1) and update_formula(F, F_1) are implemented in a similar way.

```
function simplify(F : formula) : formula;
begin
    case  /* decompose the formula */
    F = (A,B)   : return simp_step( (simplify(A),simplify(B)) );  /* and */
    F = (A;B)   : return simp_step( (simplify(A);simplify(B)) );  /* or */
    F = (A->B)  : return simp_step( (simplify(A)->simplify(B)) );  /* implies */
    F = \+(A)   : return simp_step( \+(simplify(A)) );  /* negation */
    otherwise   : return simp_step(F);
    end
end;
```

Figure 4.1 – Simplification of a formula

Table 4.3 – Simplification rules (part of simp_step's definition)		
Rule		Condition to apply this rule
Input formula	Output formula	
`(true,A)`	`A`	*(none)*
`(A,true)`	`A`	*(none)*
`(true;A)`	`true`	semantics(prolog) $\wedge$ no_side_effects(A) $\wedge$ diff_sol $\wedge$ no_bind(A)
`(true;A)`	`true`	semantics(logical)
`(A,fail)`	`fail`	semantics(prolog) $\wedge$ no_side_effects(A)
`(A,fail)`	`fail`	semantics(logical)
`(fail,A)`	`fail`	*(none)*
`(fail;A)`	`A`	*(none)*
`(A->true;B)`	`A`	semantics(prolog) $\wedge$ succeeds(A) $\wedge$ deterministic(A)
`(A->true;B)`	`A`	semantics(logical) $\wedge$ succeeds(A)
`A`	`fail`	semantics(prolog) $\wedge$ fails(A) $\wedge$ no_side_effects(A)
`A`	`fail`	semantics(logical) $\wedge$ fails(A)

Table 4.4 – The conditions for applying simplification rules	
Condition	Description
semantics(S)	Simplify according to semantics S where S $\in$ {prolog,logical}.
no_side_effects(A)	Formula A does not have side effects when executed.
deterministic(A)	Formula A gives only one solution when executed.
no_bind(A)	Formula A does not bind any variables.
diff_sol	Relax semantics of Prolog to allow a different number of solutions.
succeeds(A)	Formula A always succeeds when executed.
fails(A)	Formula A always fails when executed.

4. Factoring

Factoring is based on the operation of finding the most-specific-generalization, or MSG, of two terms. Factoring collects groups of clauses whose heads can be combined in nontrivial fashion using the MSG operation. The advantage of factoring is that it reduces the number of unifications performed during execution. Figure 4.2 defines the MSG in terms of unification. Given two terms T_1 and T_2, consider the set M of all terms that unify with both of them. The MSG of T_1 and T_2 is the unique element T_m of M which unified with any other element U of M gives T_m. Intuitively, this says that T_m contains the maximal common information of T_1 and T_2.

The MSG (also called anti-unification) is the dual operation to unification. Given two terms, unification finds a term that is a more instantiated case of each of the two, i.e. the most general common instance of the two. The MSG is a term of which each of the two is a more instantiated case. For example, consider the two compound terms `s(A,x,C)` and `s(A,B,y)`. Unifying these two terms results in `s(A,x,y)`. The MSG of the two terms is `s(A,B,C)`. Unification may fail, i.e. the most general unifier

function msg(T_1 , T_2 : term) : term;
var
    M : set of term;
    T_m , U : term;
begin
    M := { T | T unifies with T_1 and T unifies with T_2 };
    Find $T_m \in M$ such that $\forall\ U \in M$: unify(U , T_m) = T_m;
    return T_m
end;

Figure 4.2 – The most specific generalization

is the empty set. Finding the MSG never fails. In the worst case, the generalization of the two terms is an unbound variable, which represents the set of all terms. For example, consider the two atomic terms `x` and `y`. Unifying these two results in failure, whereas the MSG is an unbound variable.

Another way of viewing the MSG operation is as an approximation to the union of two sets. Every term corresponds to a set by instantiating the variables in the term to all possible ground values. In general, the union of two of these sets does not correspond to any term. The MSG finds the smallest superset of the union that is represented by a term. A similar property holds of unification: it finds the largest subset of the intersection that is represented by a term.

For all arguments of the predicate, the factoring transformation finds the largest contiguous set of clauses whose MSG is a compound term. This set is used to define a dummy predicate and the definition of the original predicate is modified to call the dummy predicate. The algorithm is given in Figure 4.3. As an example of factoring, consider the predicate:

```
h([x|_]).
h([y|_]).
h([]).
```

The lists in the heads of the first two clauses are combined: the MSG of `[x|_]` and `[y|_]` is `[_|_]`. The result after factoring is:

---

procedure factoring;
var
    M : term;
    C_i , C'_i : clause;
    π , P_π : list of clause;
    a , i , p , q : integer;
begin
    for each predicate P in the program **do begin**
        /* At this point P = [C_1 , C_2 , ... , C_n] (list of n clauses) */
        /* and C_i = (H_i :- B_i) (Each clause has head H_i and body B_i) */
        for a := 1 **to** arity(P) **do begin**
            Partition P such that each contiguous group π = [C_p , C_{p+1} , ... , C_q] ($1 \le p \le q \le n$) satisfies exactly one of the two properties:
            1. Either $p = q$ (π contains only one clause), or
            2. π is the largest group for which $M = \underset{i=p}{\overset{q}{\text{MSG}}}$ (argument a of H_i) is compound.
            for each contiguous group π **do if** $p < q$ **then begin**
                /* Create the dummy predicate P_π */
                for i := p **to** q **do begin**
                    C'_i := C_i;
                    Remove M from H'_i;
                    Add all variables in M as arguments to H'_i
                end;
                P_π := [C'_p , ..., C'_q];
                /* Create the call to the dummy predicate */
                H := (new head with same functor and arity as P and M in argument a);
                H_π := (new head with same functor and arity as P_π);
                for i := 1 **to** arity(P) **do if** $i \neq a$ **then begin**
                    Make argument i of H and H_π identical
                end;
                Replace π in P by the single clause C_π = (H :- H_π)
            end
        end
    end
end;

Figure 4.3 – The factoring transformation

```
h([A|B]) :- h'(B, A).
h([]).

h'(B, x).
h'(B, y).
```

Factoring reduces the number of unifications done at run-time in two ways: (1) compound terms are only created once during predicate execution, instead of being repeated for each clause (e.g. the list `[A|B]` in the example), and (2) the arguments of compound terms become predicate arguments, which more often

allows the determinism transformation to convert shallow backtracking into deterministic selection (e.g. the value of the second argument of the predicate `h'` determines the correct clause directly without any superfluous unifications). The following heuristic is used:

> **Factoring Heuristic:** For each argument in a predicate, factor the largest set of contiguous clauses whose MSG is a compound term. Repeat this operation until no more factoring is possible.

This heuristic needs refinement in some cases to avoid superfluous choice point creation which may slow down execution. The savings of multiple structure creation (how many fewer unifications are done) should be weighed against how much deterministic selection is possible in the dummy predicates.

If the compiler option `same_order_solutions` is enabled (the default) then the operational semantics is that of standard Prolog, i.e. the order of solutions returned on backtracking is identical to that of standard Prolog. Disabling the option relaxes the semantics of standard Prolog by also factoring non-contiguous clauses whose MSG is a compound term. This may change the ordering of solutions on backtracking. This option allows experimentation with variations of standard Prolog semantics.

To illustrate how factoring can reduce the amount of shallow backtracking, consider the following predicate, which is part of a definition of quicksort:

```
partition([Y|L],X,[Y|L1],L2) :- Y=<X, partition(L,X,L1,L2).
partition([Y|L],X,L1,[Y|L2]) :- Y>X,  partition(L,X,L1,L2).
partition([],_,[],[]).
```

The first argument of the first two clauses can be factored, resulting in:

```
partition([Y|L],X,L1,L2) :- partition'(L,X,L1,L2,Y).
partition([],_,[],[]).

partition'(L,X,[Y|L1],L2,Y) :- Y=<X, partition(L,X,L1,L2).
partition'(L,X,L1,[Y|L2],Y) :- Y>X,  partition(L,X,L1,L2).
```

(In the compound term `[Y|L]` the rightmost variable L is kept in the same argument position and the other variable Y is put at the end of the goal.) The transformation results in only a single unification of `[Y|L]` instead of two in the original definition. In the dummy predicate the comparisons `Y=<X` and `Y>X` use arguments of the predicate, not arguments of a compound term. This makes it possible to compile `partition/4` with a conditional branch instead of with shallow backtracking.

5. Global dataflow analysis

It is difficult to obtain information about a program by executing it in its original form, since the range of possible behaviors is potentially infinite, and even simple properties of programs may be undecidable. To get around this problem, the idea of *abstract interpretation* is to transform the program into a simpler form which allows practical analysis. After the analysis the inverse transformation gives information about the original program. The fundamentals of a general method based on this approach and its mathematical underpinning are explained by Kildall [37] and Cousot & Cousot [23]. Marriott and Sondergaard [47] give a lucid explanation of the basic ideas. This method has been studied extensively and developed into a practical tool for Prolog [18, 21, 24, 25, 49, 50, 53, 66, 67, 76, 84].

The four sections that follow summarize the relevant parts of the theory of abstract interpretation, present my application of it to Prolog, describe the analysis algorithm in detail, and discuss the integration of the algorithm into the body of the compiler. In Chapter 7 an evaluation is done of the effectiveness of the algorithm.

5.1. The theory of abstract interpretation

The transformed program should mimic the original faithfully. This is made rigorous by introducing the concept of *descriptions* of data objects. Let E be the powerset, i.e. the set of all subsets, of a set of data objects, and D be a partially ordered set of descriptions. Then an *abstract interpretation* is defined by the following conditions:

(1) $E_P : E \to E, \quad D_P : D \to D$

(2) $\alpha : E \to D, \gamma : D \to E$

(3) α and γ are monotonic.

(4) $\forall d \in D : d = \alpha(\gamma(d))$

(5) $\forall e \in E : e \leq \gamma(\alpha(e))$

(6) $\forall d \in D : E_P(\gamma(d)) \leq \gamma(D_P(d))$

The operator E_P in the first condition describes a single step of the execution of the program P as a state transformation. Symbolic execution of the transformed program is described by the operator D_P. Except for the conditions given above, the choice of E_P and D_P is completely free. The choice is guided by several trade-offs, for example: (1) speed versus precision of the analysis, (2) complexity versus confidence in the correctness of the analysis.

As an example of E_P (from Cousot & Cousot [23]), consider a program in an imperative language represented as a graph where each node is a simple statement such as an assignment or a conditional. Let an *environment* be defined as a correspondence between each variable in the program and a possible value. Then for each edge of the graph a set of possible environments (called a *context*) is given. Initially they are all unknown. An application of E_P transforms all contexts to their new values reached after one execution step.

For Prolog, a natural choice is to identify E_P with the standard operator $T_P : 2^{B_P} \rightarrow 2^{B_P}$ which describes its procedural semantics. In this case E is 2^{B_P}, where B_P is the Herbrand universe of the program P, i.e. the set of all ground goals† that can be constructed using predicates, functors, and constants of the program. T_P does a single ''forward chaining'' step to find the conclusions that can be inferred from a given set of ground goals. Formally, T_P maps any $I \subseteq \mathrm{B_P}$ into $T_P(I) = \{ A \in \mathrm{B_P} : \mathrm{A} :- \mathrm{A_1}, \cdots, \mathrm{A_n}$ is a ground instance of a clause in P and $\{ A_1, \cdots, A_n \} \subseteq \mathrm{I} \}$. In other words, an application of T_P transforms a subset of B_P into a new subset containing the new goals inferred from the program's clauses given the old goals. The *meaning* of a program P is defined as $\mathrm{lfp}(T_P)$ (where lfp = the least fixpoint operator). This is the set of all ground goals that can be derived from the program clauses. For example, consider the following program:

```
nat(0).
nat(s(X)) :- nat(X).
```

which states that `nat(X)` is true if X is zero or X is the successor of a natural number. The program's meaning is:

---

† These are called ''atoms'' in mathematical logic. To avoid confusion with the atom data type in Prolog, this dissertation uses the Prolog terminology.

```
{ nat(0), nat(s(0)), nat(s(s(0))), nat(s(s(s(0)))), ... }
```

which represents the set of natural numbers.

The second and third conditions introduce the operators α (the abstraction function) and γ (the concretization function). The operator $\alpha : E \to D$ determines the description corresponding to a particular set of data objects. The operator $\gamma : D \to E$ determines the set of data objects corresponding to a particular description.

The fourth and fifth conditions ensure that α and γ behave correctly with respect to each other. Condition four means that in going from descriptions to data objects and back no information is lost. Condition five means that in going from a data object to a description and back that the resulting set of data objects includes the original data object. The sixth condition is known as the *safeness criterion*. It is necessary to ensure that the symbolic execution (through D_P) mimics the execution of P accurately (through E_P). In other words, the abstract interpretation gives descriptions that include all the data objects that the execution of the original program gives.

To illustrate what the conditions mean consider the abstract domain of signs of real numbers. The data objects are real numbers. Let there be three possible signs for numbers: + (positive), – (negative), and 0 (zero). The set of descriptions D describes the possible states of a set of reals, so it contains all combinations of the three signs:

$$D = \{ \{\}, \{0\}, \{+\}, \{-\}, \{+, -\}, \{-, 0\}, \{+, 0\}, \{+, -, 0\} \}$$

According to the second condition α maps a set of reals onto its signs, and γ maps a set of signs onto a set of reals. For example:

$$\alpha(\{-5\}) = \{-\}$$

$$\alpha(\{-3, 5\}) = \{+, -\}$$

$$\gamma(\{+\}) = \{ r \in \mathbf{R}, r > 0 \}$$

The fourth condition says that going from a sign to a set of reals and back will give the same sign. The fifth condition says that going from a set of reals to a sign and back will give a set of reals that includes the

original set. So for example:

$$\{+\} = \alpha(\gamma(\{+\}))$$

since $\gamma(\{+\})$ = the set of positive reals, whose sign is $\{+\}$, and:

$$\{5\} \subset \gamma(\alpha(\{5\}))$$

since $\alpha(\{5\}) = \{+\}$, and $\gamma(\{+\})$ is the set of positive reals, which contains 5. In order to explain condition six, consider the equation 27×37. Here E_P is multiplication of reals, and D_P is the corresponding operation in the abstract domain of signs. The multiplication corresponds to $\{+\} \times \{+\}$ in the abstract domain. The result of the abstract multiplication should be $\{+\}$, since $27 \times 37 = 999$, which is positive. Condition six is a formalization of this requirement.

Dataflow analysis is done by transforming the original program over the domain E described by E_P to a new version over the domain D described by D_P. Then $\gamma(\text{lfp}(D_P))$ (lfp = the least fixpoint operator) gives a conservative estimate of the required information. Much work has been done in discovering useful domains D for particular applications and efficient algorithms for finding fixpoints of D_P [10, 53].

5.2. A practical application of abstract interpretation to Prolog

The implementation of abstract interpretation presented in this dissertation uses a very different E_P from the one suggested in the previous section by the formal definition of Prolog's procedural semantics. The choice of E_P used in the Aquarius compiler closely follows execution on a machine. Consider a program with n predicates P_i. The *data objects* are the n-tuples $(T_1, T_2, \cdots, T_n)$ where each T_i is a functor of same name and arity as P_i and the arguments of T_i are terms constructed using data functors and atomic terms in the program and possibly containing unbound variables. E is the powerset of these data objects. The *descriptions* are the n-tuples $(L_1, L_2, \cdots, L_n)$ where each L_i is a functor of same name and arity as P_i and the arguments of L_i are constrained to be on a given finite lattice. D is the set of these descriptions. A *lattice* is a partially ordered set in which every nonempty subset has a least upper bound (denoted as the lub) and a greatest lower bound (denoted as the glb). Each of the elements of the lattice corresponds to a set of possible values in the original program. This lattice is called an *argument lattice*, since it is used to represent the possible values of a predicate argument. A *predicate lattice* (such as L_i) is the Cartesian

product of the lattices of all the predicate's arguments.

The operator E_P that mirrors execution of the program corresponds to a single resolution step. It is a transformation of a set of data objects and an execution state to another set of data objects and a new execution state, following Prolog's depth-first execution semantics, that is, its left-to-right execution of goals in a clause, and its top-to-bottom selection of clauses in a predicate. The operator D_P that mirrors execution of the program over the descriptions is similar, except that the arguments are lattice values.

If the conditions of abstract interpretation hold, then the least fixpoint of the symbolic execution over the lattice is a conservative approximation to the global information, in other words the set of values that a variable can have during execution is a subset of what is derived in the analysis.

The three sections that follow describe the lattice used by the analysis algorithm. The first section introduces and defines the lattice elements and the types with which they correspond. The next section gives an example to show how to derive the types. The last section summarizes the properties of the types that are used by the algorithm.

5.2.1. The program lattice

Dataflow analysis for Prolog differs from that of statically typed languages because it does not check types, but it infers them. The most important information that can be deduced about an argument is whether it is used as an input or an output argument of a predicate, i.e. the *mode* of the argument. After the mode is determined, it is useful to find its *type*, i.e. the set of values that it can have. The remainder of this chapter refers only to the type of an argument, in the assumption that this implies the mode as well. I have experimented with four lattices of varying complexity in the analyzer, and the lattice that is currently implemented has been chosen to give the most information while keeping analysis fast.

During the analysis the algorithm maintains two lattices for each predicate in the program. These lattices correspond to the *entry* and *exit* types of the predicate, i.e. the value of the variable valid upon entering the predicate and upon successful exit from the predicate. The lattice describing the entire program is the Cartesian product of the predicate lattices.

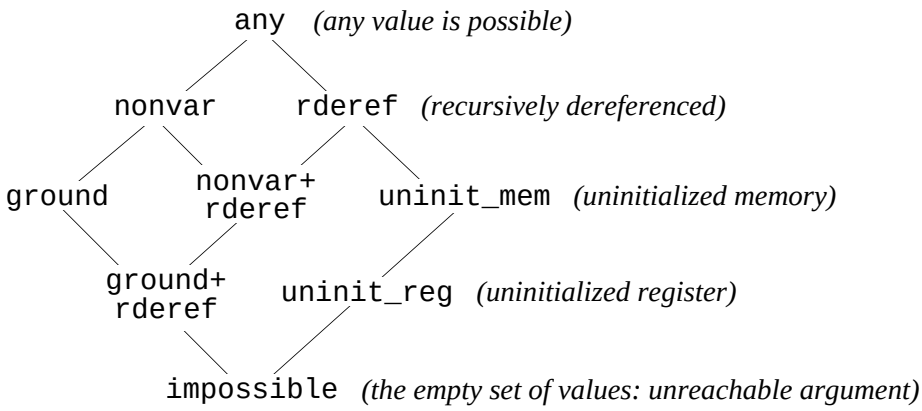

Figure 4.4 – The argument lattice

The argument lattice of the entry and exit types in the current analyzer is shown in Figure 4.4. In this lattice, `any` (the top element) denotes the set of all values, `impossible` (the bottom element) denotes the empty set (i.e. this predicate is unreachable during execution), `uninit` denotes the set of uninitialized variables (unbound variables that are not aliased; see Chapter 2), `ground` denotes the set of values that are ground (i.e. the term contains no unbound variables), `nonvar` denotes the set of nonvariables, `rderef` denotes the set of values that are recursively dereferenced (i.e. the term is dereferenced, which means that it is accessible without any pointer chasing, and if it is compound then all its arguments are recursively dereferenced), and `ground+rderef` denotes the set of values that are both ground and recursively dereferenced.

5.2.2. An example of generating an uninitialized variable type

This section gives a simple example of the generation of uninitialized variable types to give an idea of what abstract interpretation does and to illustrate the argument lattice. Uninitialized variables are generated whenever the analyzer deduces that an unbound variable cannot be aliased to another. For example, consider the following program fragment:

```
pred(...) :- ..., goal(Z), ...
goal(X) :- X=s(Y), goal(Y).
```

If Z is the first occurrence of that variable in the `pred(...)` clause then it is considered a candidate uninitialized variable. This is possible because it is certainly not aliased to any other variable. In the definition of `goal(X)`, if X is uninitialized then the argument Y of the structure `s(Y)` may be considered uninitialized as well. This Y is passed on as an argument to `goal(Y)`. Therefore both calls of `goal(X)` are with an uninitialized argument, so it is consistent to give the argument X an uninitialized variable type.

It may happen that elsewhere in the program there is a call of `goal(X)` where X is not uninitialized (for example it may be a compound term, or it may be aliased). In that case, the assumption that X is uninitialized is invalidated. This may invalidate assumptions about other arguments of other predicates, so it is necessary to propagate this information. For correctness, it is necessary to iterate until the least fixpoint is reached. At that point symbolic execution of the program does not change any of the derived types.

5.2.3. Properties of the lattice elements

The example given above already gives an inkling of the relevant properties of ground, uninitialized, and recursively dereferenced variables that simplify the analysis. Here is a more complete list of these properties:

- The property of being ground, uninitialized, or recursively dereferenced propagates through explicit unifications. The propagation is bidirectional:

  (1) If X is ground, uninitialized, or recursively dereferenced, then after executing an explicit unification with a compound term (e.g. `X=s(A,B)`), all of its variables (e.g. A and B) are ground, all of the new variables (e.g. A and B) are uninitialized, or all of the new variables are recursively dereferenced.

  (2) In the other direction, if all the variables in the compound term are ground, then X is ground. If all the variables are recursively dereferenced, then X is recursively dereferenced if it was

previously uninitialized.

- The property of being ground is independent of aliasing. For example, if X is ground, then it remains ground after executing the unification `X=Y`. This is not true of recursively dereferenced or uninitialized variables.
- An uninitialized variable is not aliased to any other variable. Lattice calculations for uninitialized variables do not affect each other.

5.3. Implementation of the analysis algorithm

Previous sections have introduced the ideas underlying the algorithm, the program lattice used by the algorithm, an example of how types are derived, and the properties of the lattice elements. This section gives a more complete explanation of the algorithm. The presentation starts with an overview of the data representation. It then describes the algorithm, and finally it gives a detailed example of analysis.

Table 4.5 – The components of the variable set VS

Name	Description
S	The set of variables encountered so far in the clause. This set is important because any variable encountered in a goal that is not in this set is known not to be aliased to any other, i.e. it is a new variable, and therefore it is both uninitialized and dereferenced.
G	The set of variables that are ground. These variables are bound to terms that contain no unbound variables.
N	The set of variables bound to a nonvariable term. This set is a superset of G.
U	The set of variables that are uninitialized. A variable becomes uninitialized if it is unbound and known not to be aliased to any other variable. The symbolic execution enforces this constraint. This set is disjoint with N.
D	The set of variables that are recursively dereferenced. A variable is recursively dereferenced if it is bound to a term that is dereferenced, i.e. it is accessible without any pointer chasing, and if it is compound then all its arguments are recursively dereferenced. This set is a superset of U.

5.3.1. Data representation

During analysis the types are represented in two ways:

(1) **As lattice elements**. For each predicate, there are two structures containing a lattice element in each

argument. These structures represent the entry and exit types of the predicate. For example, the predicate `concat/3` has two structures which could have the values:

```
entry: concat(any,ground,uninit)
 exit: concat(ground,ground,any)
```

This says that upon entering `concat/3` the second argument is ground and the third argument is uninitialized. When the predicate is exited the first two arguments are ground.

(2) **As sets of variables**. Type information can also be stored as a set for each type that contains the variables of that type.

These two different representations each have their advantages. The lattice representation makes it easy to calculate the lub (least upper bound). The variable set representation makes it easy to symbolically execute a clause, i.e. to propagate and update information about variables' types through the clause. Functions are provided to convert between the two representations (Figure 4.7). For the lattice in Figure 4.4, there are five sets of variables which are updated during the symbolic execution of a predicate. Conceptually they are part of a 5-tuple VS = (S, G, N, U, D) that holds the current type information (Table 4.5).

5.3.2. Evolution of the analyzer

The current analyzer was preceded by three simpler versions. The lattice of the first analyzer represented only entry types and had three elements: `impossible`, `uninit`, and `any`. The second analyzer added the `ground` type in the entry lattice and an exit lattice of the same structure. The third analyzer added the `rderef` type to these lattices. The current (fourth) analyzer added the `nonvar` type. Despite not using a representation for variable aliasing, the third and fourth analyzers are able to derive many nontrivial `rderef` and `nonvar` types. The added types are independent, i.e. each version of the analyzer does no better than previous versions on types that previous versions also derive.

The choice of what lattice types to add was done by inspecting the compiled code of programs and by deciding what types were easy to derive in the context of the structure of the existing analyzers. Types were added that are present in many programs. Measurements show that having an exit lattice and doing back propagation (see below) are essential features to derive good `ground`, `rderef`, and `nonvar`

types. A numerical evaluation of the efficiency of the analysis (the percentage of arguments for which types are derived) and the effect of analysis on execution time and code size is given in Chapter 7.

For the next version of the analyzer the added types `rlist` (recursive list, i.e. the term is either nil or a cons cell whose tail is a recursive list), `integer`, and `((nonvar+deref) or uninit)` (the term is either a dereferenced nonvariable or uninitialized) are contemplated.

```
type  varset = (set, set, set, set, set);  /* 5-tuple */

var   Program  : set of predicate;
      L_entry  : mapping predicate → lattice;
      L_exit   : mapping predicate → lattice;
      P  : predicate;

procedure analysis;
var   E  : set of predicate;
      VS  : varset;
begin
      E := { P | arity(P) = 0 } ∪ (declared entry points);
      Initialize L_entry with the types of the declared entry points;
      Initialize L_exit to impossible for all predicate arguments;
      while E ≠ ∅ do begin
            for each predicate P ∈ E do begin
                  VS := lattice_to_varset(L_entry[P], P);
                  VS := update_exit(VS, predicate_analyze(P, VS), P)
            end;
            E := { P | L_entry[P] has changed or ∃ G∈P : L_exit[G] has changed}
      end
end;
```

Figure 4.5 – The analysis algorithm: top level

5.3.3. The analysis algorithm

The analysis algorithm is presented at three levels of detail. An English-language description is given of the basic ideas. A detailed pseudocode definition (Figures 4.5 through 4.7) describes the complete algorithm at a high level of abstraction. Appendix G gives the implementation in the compiler.

The algorithm maintains entry and exit lattice elements for each predicate argument in the program. Analysis proceeds by traversing the call graph starting from a set of entry points that have known types. The entry points include all predicates of arity 0 and any entry declarations given by the programmer (Appendix A). The traversal is repeated until there are no more changes in the lattice values, that is, until a

```
function predicate_analyze(P : predicate; VS : varset) : varset;
var   F : formula;
      VS_i : array [1 .. n] of varset;
      G_e : goal;
      i , j : integer;
begin
      /* At this point P = [ C_1 , ... , C_n ] (list of n clauses) */
      for each non-active clause C_i ∈ P do begin /* Symbolic execution of clause C_i */
            /* At this point C_i = (G_i1 , ... , G_in_i) (conjunction of n_i goals) */
            VS_i := VS;
            for j := 1 to n_i do begin /* Symbolic execution of goal G_ij */
                  if (G_ij is a unification) then begin
                        VS_i := symbolic_unify(VS_i , G_ij) /* Figure 4.8 */
                  end else if G_ij ∈ Program then begin /* G_ij is defined in the program */
                        L_entry[G_ij] := lub(L_entry[G_ij], varset_to_lattice(VS_i , G_ij));
                        if non-exponentiality constraint then begin
                              VS_i := update_exit(VS_i , predicate_analyze(G_ij , VS_i), G_ij)
                        end
                  end else begin /* G_ij is not defined in the program */
                        F := varset_to_type(VS_i , G_ij);
                        G_e := entry_specialize(G_ij , F);
                        VS_i := update_exit(VS_i , exit_varset(G_e), G_ij)
                  end
            end;
            VS_i := back_propagate(VS_i , C_i) /* To obtain more precision */
      end;
      return ⋂_{i=1}^{n} VS_i /* Merge the exit values of all VS_i */
end;
```

Figure 4.6 – The analysis algorithm: analyzing a predicate

fixpoint is reached. With suitable conditions (i.e. all type updating is *monotonic* and types are propagated correctly) this fixpoint is the least fixpoint and the resulting types give accurate information about the original program. When a goal is encountered during a traversal three things are done: (1) the goal's entry lattice type is updated using the current value of VS, (2) if the goal's definition is part of the program then the definition is entered, and (3) upon return, the new value of VS is used to update the goal's exit lattice type. A correct value of VS is maintained at all times during the traversal of a goal's definition.

The definition of the algorithm in Figures 4.5 through 4.7 leaves out some details but is a faithful description of the analysis. The two conditions *non-active* and *non-exponentiality* are explained in the next section. The following sections describe what happens in symbolic execution of a predicate (including back propagation) and symbolic execution of a goal.

```
function update_exit(VS1 , VS2 : varset; G : goal) : varset;
var   VS : varset;
begin
      /* Calculate new VS from old VS1 and exit VS2 */
      VS.nonvar := VS1.nonvar ∪ VS2.nonvar;
      VS.ground := VS1.ground ∪ VS2.ground;
      VS.rderef := (VS1.rderef ∩ VS1.ground) ∪ VS2.rderef;
      VS.sofar := VS1.sofar ∪ vars(G);
      VS.uninit := VS1.uninit − vars(G);
      /* Calculate new exit lattice */
      Lexit[G] := lub(Lexit[G], varset_to_lattice(VS, G));
      return VS
end;

function lub(L1 , L2 : lattice) : lattice;
return (least upper bound of L1 and L2);

function lattice_to_varset(L : lattice; G : goal) : varset;
return (varset corresponding to L using variables of G);

function varset_to_lattice(VS : varset; G : goal) : lattice;
return (lattice corresponding to VS using variables of G);

function back_propagate(VS : varset; C : clause) : varset;
return (improved exit varset from VS using unification goals of C);

function varset_to_type(VS : varset; G : goal) : formula;
return (type formula corresponding to VS using variables of G);

function entry_specialize(G : goal; F : formula) : goal;
return (specialized entry point of G when called with type F);

function exit_varset(G : goal) : varset;
return (exit varset stored for the known goal G);
```

Figure 4.7 – Utility functions needed in the analysis algorithm

5.3.4. Execution time of analysis

This section shows that the average analysis time for programs that contain only linearly recursive predicates (i.e. no clauses contain more than one recursive call) and that have bounded arity is proportional to the size of the program. The analysis time $T_{analysis}$ is proportional to the time of each iteration T_{iter} and the number of iterations N_{iter} needed to reach the least fixpoint:

$$T_{analysis} = O(T_{iter} \cdot N_{iter})$$

For programs that contain only linearly recursive predicates, the time of each iteration is:

$$T_{iter} = O(S \cdot A)$$

where S is the total number of goals in the program and A is the maximum number of times a predicate is traversed. (Programs with non-linearly recursive predicates are discussed below.) This is true because the algorithm traverses each clause at most once in an iteration. It assumes that the symbolic execution of a goal whose definition is not traversed is a constant time operation. A predicate is traversed only if the current entry type is worse than the previous worst entry type. The number of times this situation can occur is bounded by the depth of the entry lattice of the predicate, which is proportional to the maximal arity in the program. Therefore:

$$S = \sum_{i=1}^{n} \sum_{j=1}^{n_i} length(C_{ij})$$

$$A = O(\max_{i=1}^{n} arity(P_i))$$

where the program contains n predicates, and each predicate P_i contains n_i clauses C_{ij}. The arity of a predicate is denoted by $arity(P_i)$ and the number of goals in a clause is denoted by $length(C_{ij})$. The number of iterations is trivially bounded by the depth of the program lattice:

$$N_{iter} = O(D_{total})$$

where D_{total} is given by:

$$D_{total} = \sum_{i=1}^{n} 2 \cdot 4 \cdot arity(P_i)$$

In this equation, 2 counts the entry and exit lattices, $arity(P_i)$ is the number of arguments in the predicate lattice, and 4 is the depth of each argument lattice. This bound on N_{iter} is wildly pessimistic. For most real programs N_{iter} is bounded by a small constant. All the benchmark programs satisfy $N_{iter} \leq 7$ (Chapter 7). However, there exist pathological predicates P_x for which $N_{iter} = \theta(arity(P_x))$. For example, consider the program:

```
main :- a(9,a,_,_,_,_,_,_,_,_,_).

a(0,_,_,_,_,_,_,_,_,_,_).
a(N,A,A,C,D,E,F,G,H,I,J) :- N1 is N-1, a(N1,A,C,D,E,F,G,H,I,J,A).
```

The analyzer requires 10 passes to determine that all arguments of `a/11` are ground and dereferenced upon exit.

To summarize these results, the worst-case and average case total execution times of analysis for programs without non-linearly recursive predicates are:

$$T_{analysis,worst} = O(A \cdot S \cdot D_{total})$$

$$T_{analysis,ave} = O(A \cdot S)$$

If the arity is bounded, then the average execution time of analysis is proportional to the program's size.

For programs that contain non-linearly recursive predicates this result needs to be amended. There is a trade-off between precision and execution time of the analysis. If not enough predicates are traversed then analysis information is lost. If too many predicates are traversed then analysis time becomes too long. Two constraints are used to prune the traversal of the call graph:

(1) The *non-active* constraint. A clause that is in the process of being traversed is called an *active* clause. During recursive calls of predicate_analyze, the algorithm maintains a set of the active clauses and will not traverse an active clause twice.

(2) The *non-exponentiality* constraint. Traverse a predicate (i.e. call predicate_analyze) only if one of two conditions hold: (a) The entry type has changed since the last traversal of the predicate, or (b) At least one of the predicate's clauses is active.

Condition (a) is understandable: it is needed to ensure that an updated type is propagated correctly. The rationale for condition (b) is more subtle. If it did not hold, then the exit types derived by the analysis would be significantly worse because the base case of a recursive predicate may not be reached during the traversal. Running the analyzer both with and without this condition shows this to be true for most programs.

The problem with condition (b) is that it leads to an analysis time that is exponential in the number of non-linearly recursive clauses in a predicate. For many programs this is not serious. However, it occurs often enough that it should be solved. One of the benchmark programs, the nand benchmark, has this problem. A better condition is needed to replace condition (b). It must (1) ensure that the base case of all recursive predicates is reached (for good exit types), and (2) not result in time exponential in the number of non-linearly recursive predicates.

5.3.5. Symbolic execution of a predicate

The heart of the dataflow analysis algorithm is the symbolic execution of a predicate (Figure 4.6). Each clause of the predicate is traversed from left to right. During the traversal the type information is kept in the variable set VS. Symbolic execution of the predicate consists of four steps:

(1) For each clause of the predicate, translate the lattice entry type of the predicate into the variable set VS, and start traversing the clause.

(2) Symbolically execute each goal in the clause and update VS.

(3) At the end of each clause, *back propagation* improves VS by deducing information that only becomes available at the end of the clause. For example, consider the clause:

```
a(X) :- X=[Y|L], b(Y, L).
```

If both Y and L are in the ground set G of VS at the end of the clause then this is also true of X because of the unification `X=[Y|L]`. Back propagation is used to improve the exit types for ground, recursive dereference, and nonvariable types. Measurements show that it is a necessary step to get good exit types.

(4) At the end of the predicate, combine the variable sets of all clauses by intersecting their corresponding components. Convert the result back to the lattice representation and update the exit type for the predicate.

5.3.6. Symbolic execution of a goal

Symbolic execution of a goal is done in three ways, depending on whether the goal is a unification, the goal is defined in the program, or the goal is not defined in the program.

5.3.6.1. Unification goals

Symbolic execution of unification is defined by the function symbolic_unify(VS, X=T) in Figure 4.8, which converts VS = (S, G, N, U, D) into VS′ = (S′, G′, N′, U′, D′). These equations use the utility functions of Table 4.6. For each component of VS, any equation in Figure 4.8 with a true condition can be

Table 4.6 – Utility functions of a term T	
Notation	Definition
$vars(T)$	The set of variables in the term T.
$dups(T)$	The set of variables that occur at least twice in the term T.
$new(T) = vars(T) - S$	The set of all variables in T that have not occurred before.
$old(T) = vars(T) \cap S$	The set of all variables in T that have occurred before.
$deref(T) = vars(T) - (S - U)$	The set of all variables in T that are candidates to be recursively dereferenced. This is the same as $new(T) \cup (vars(T) \cap U)$, i.e. $new(T)$ supplemented with the variables in T that are uninitialized.

$$S' = S \cup vars(X=T)$$

$$G' = \begin{cases} G \cup vars(T) & \text{if } X \in G \\ G & \text{otherwise} \end{cases}$$

$$N' = \begin{cases} N \cup G' \cup \{X\} & \text{if nonvar(T) or (var(T) and } T \in N) \\ N \cup G' & \text{otherwise} \end{cases}$$

$$U' = \begin{cases} U \cup new(T) - old(T) - \{X\} - dups(T) & \text{if } (X \notin S \text{ or } X \in U) \\ U - vars(X=T) & \text{otherwise} \end{cases}$$

$$D' = \begin{cases} D \cup deref(T) \cup \{X\} & \text{if } (X \notin S \text{ or } X \in U) \text{ and } old(T) \subset (D \cup U) \\ D \cup deref(T) & \text{if } (X \notin S \text{ or } X \in U) \text{ or } X \in (D \cap G) \\ D \cup deref(T) - \{X\} & \text{if } dups(T) = \varnothing \text{ and } X \in D \text{ and } old(T) \subset U \\ D \cap G & \text{otherwise} \end{cases}$$

Figure 4.8 – Symbolic unification VS′ := symbolic_unify(VS, X=T)

used. In practice, if more than one condition is satisfied, an equation giving more information (i.e. the resulting set is larger) is used first. These equations are listed first. For example, the first equation of D′ gives a larger set, so it is preferred over the others. If both X and T are variables, then the algorithm switches X and T is to see if one of the more desirable equations is satisfied before attempting one of the lesser equations.

Table 4.7 – Conditions for the lattice entry type	
Name	Condition
C_{ground}	*vars*(X) $\subset$ G
C_{var}	var(X)
C_{old}	X $\in$ (*dups*(P) $\cup$ S $-$ U)
C_{rderef}	(*vars*(X) $\cap$ S) $\subset$ D
C_{nonvar}	(X $\in$ N)

Table 4.8 – Calculation of the lattice entry type					
C_{ground}	C_{var}	C_{old}	C_{rderef}	C_{nonvar}	Lattice value
yes	-	-	yes	-	`ground+rderef`
yes	-	-	no	-	`ground`
no	no	-	yes	-	`nonvar+rderef`
no	no	-	no	-	`nonvar`
no	yes	no	-	-	`uninit`
no	yes	yes	yes	yes	`nonvar+rderef`
no	yes	yes	yes	no	`rderef`
no	yes	yes	no	yes	`nonvar`
no	yes	yes	no	no	`any`

5.3.6.2. Goals defined in the program

Symbolic execution of a goal with a definition is done by symbolically executing the definition. Information is kept about the part of the call graph that has already been traversed, so that analysis will not go into an infinite loop. The function varset_to_lattice(VS, P) is defined by Tables 4.7 and 4.8. For each argument X of P, first determine the values of the five conditions in Table 4.7. Then use these conditions to look up the lattice value for the argument in Table 4.8.

5.3.6.3. Goals not defined in the program

Examples of goals that are not defined in the program being analyzed are built-ins and library predicates. Symbolic execution of these goals is done in two parts. First, entry specialization replaces the goal by a faster entry (section 5.4.1). Second, the type declarations that the programmer has given for the entry are used to continue the analysis. If there are none, then worst-case assumptions are made.

5.3.7. An example of analysis

The following program is interesting because it is mutually recursive:

Table 4.9 – Analysis of an example program

	`incl_2(A,B,C)`			`incl_3(A,B,C,D)`			
	A	B	C	A	B	C	D
Start							
entry	`impossible`	`impossible`	`impossible`	`impossible`	`impossible`	`impossible`	`impossible`
exit	`impossible`	`impossible`	`impossible`	`impossible`	`impossible`	`impossible`	`impossible`
After pass 1							
entry	*rderef*	*uninit*	*uninit*	*uninit*	*rderef*	*rderef*	*uninit*
exit	*nonvar*	*rderef*	*nonvar*	*rderef*	*any*	*ground*	*nonvar*
After pass 2							
entry	`rderef`	`uninit`	`uninit`	`uninit`	`rderef`	`rderef`	`uninit`
exit	`nonvar`	`rderef`	`nonvar`	`rderef`	`any`	*nonvar*	`nonvar`
After pass 3							
entry	`rderef`	`uninit`	`uninit`	`uninit`	`rderef`	`rderef`	`uninit`
exit	`nonvar`	`rderef`	`nonvar`	`rderef`	`any`	`nonvar`	`nonvar`

```
main :- incl_2([A,B], C, D).

incl_2([], C, [C]).
incl_2([A|E], C, D) :- incl_3(C, A, E, D).

incl_3(C, A, E, [A|D]) :- incl_2(E, C, D).
```

The predicates `incl_2/3` and `incl_3/4` are extracted from a definition of set inclusion. Three analysis passes are necessary to reach the fixpoint (Table 4.9). The entries that have changed with respect to the previous pass are in *italics*. The final types are given in Table 4.10. Most of the correct types are determined after the first pass. A single exit type of `incl_3/4` is corrected in the second pass. This is necessary because the third argument of `incl_3/4` is the same as the first argument of `incl_2/3`.

Table 4.10 – Final results of analysis

`incl_2(A,B,C)`	
entry type:	`rderef(A),uninit(B),uninit(C)`
exit type:	`nonvar(A),rderef(B),nonvar(C)`
`incl_3(A,B,C,D)`	
entry type:	`uninit(A),rderef(B),rderef(C),uninit(D)`
exit type:	`rderef(A),nonvar(C),nonvar(D)`

5.4. Integrating analysis into the compiler

Deriving type information is only the beginning. The analyzer must be integrated into the compiler to take advantage of the type information. The dataflow analysis module itself does four source transformations (Figure 4.9) before passing the result to the next stage, which does determinism extraction. The

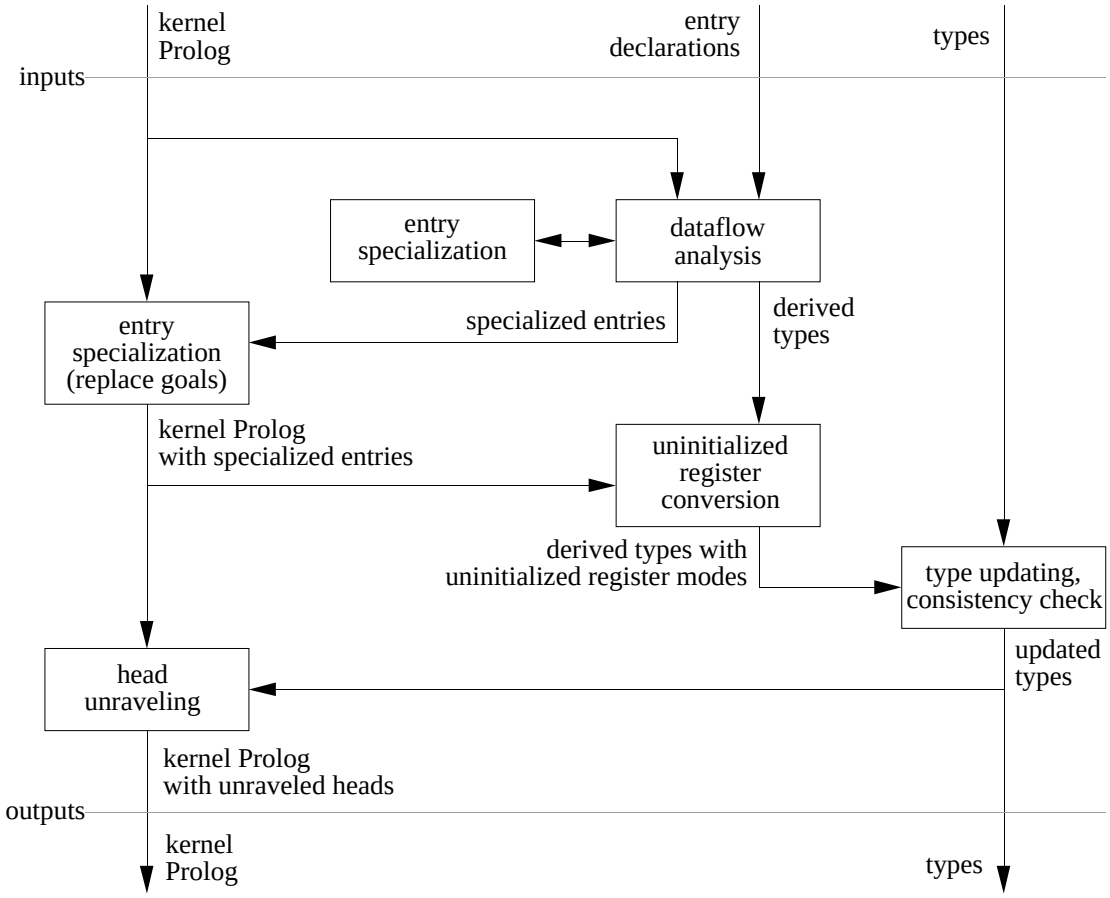

Figure 4.9 – Integrating analysis into the compiler

following source transformations are done in the dataflow analysis module:

(1) **Entry specialization**. Determine a fast entry point for each occurrence of a call whose definition is not in the program being analyzed and continue analysis with this entry point.

(2) **Uninitialized register conversion**. Convert uninitialized memory types to uninitialized register types when it results in a speedup. It is done when an argument can be returned in a register without giving up last call optimization.

(3) **Head unraveling**. Unravel the heads of all clauses again in the light of the derived type information. For example, the head `a(A,A,A)` can be unraveled in three different ways, namely

`(a(A,B,C):-A=B,C=B)` or `(a(A,B,C):-A=C,B=C)` or `(a(A,B,C):-B=A,C=A)`. If both A and B are nonvariables and C is unbound, then the first or third possibilities allow the compiler to do argument selection. Unraveling is already done during the conversion to kernel Prolog, but it must be done again after dataflow analysis since the new types may allow it to be done better.

(4) **Type updating**. Supplement the type declarations given by the programmer (if any) by the derived types. All inconsistencies are reported and compilation continues with the corrected types.

The first three of these transformations are discussed in more detail in the following sections.

5.4.1. Entry specialization

During analysis, a fast entry point is determined for each call whose definition is not in the program being analyzed (i.e. each *dangling* call). For example, the call `sort(A,B)` is replaced by the entry point `'$sort *2'(A,B)` if B is uninitialized. Analysis continues with the types of the fast entry point. The program is unchanged until the end of analysis, so the determination of the fast entry point is repeated in each analysis iteration whenever a dangling call is encountered. This mechanism is intended to speed up execution of built-in predicates and library routines, but it is also available to the programmer.

The fast entry point is determined by calculating the type formula corresponding to the variable set *VS* with the function varset_to_type(*VS*, *G*) (Figures 4.6 and 4.7). This type formula is used to traverse the modal entry tree for the goal. The modal entry tree is a data structure that contains a set of entry points and the types that each requires (Appendix A). Entry specialization is also done in the clause compiler, and a detailed example of the use of the modal entry tree is given in Chapter 5 (section 3.4).

5.4.2. Uninitialized register conversion

Often an uninitialized memory type can be converted to an uninitialized register type. The compiler uses four conditions to guide the conversion process. Define a *survive* goal as one that does not alter any temporary registers (except for arguments with uninitialized register type, which are outputs). A goal that potentially alters temporary registers is a *non-survive* goal. The compiler maintains a table of survive goals. With these definitions the four conditions for a predicate *P* are:

(1) All arguments of P with uninitialized memory type are *candidates* to be converted.

(2) A candidate argument of P must occur at most once in the body of each clause of P. In each clause where it occurs, the argument must be in the last non-survive goal or any survive goal beyond it.

(3) For each clause of P, if the last goal G is a non-survive goal, then the candidate argument of P must be in the same argument position in G as in the head of P. This is necessary to avoid losing the opportunity for last call optimization (LCO): if the argument positions are different then a move instruction is needed between the last call and the return. If the last goal is a survive goal then the condition is unnecessary because it is not as important to retain LCO: a survive goal can never be mutually recursive with the predicate it is part of.

(4) Often the last goal G has candidate arguments that are not candidate arguments of P, so they have to be initialized when returning from G. This has two disadvantages: P loses LCO and P must allocate an environment (which may not exist otherwise). The solution to this problem involves a trade-off: is it better to have LCO in P and fewer uninitialized register arguments in G, or to have no LCO in P and more uninitialized register arguments in G? The compiler recognizes a class of predicates G for which the first is true: Define a *fast* predicate as one whose definition contains only built-ins and survive goals. If G is fast then reduce the set of G's candidate arguments to include only those that are candidate arguments of P.

A transitive closure is done until all four conditions are satisfied. These conditions can be relaxed slightly in several ways. However, even with the existing conditions it is possible to convert about one third of all uninitialized types into uninitialized register types (Chapter 7). The third and fourth conditions are not needed for correctness, but only for execution speed. The third condition ensures that LCO is not lost. The fourth condition speeds up the chat_parser benchmark by 1% and was added after code inspection discovered cases where the use of uninitialized registers slows down execution.

5.4.3. Head unraveling

This transformation repeats the head unraveling transformation (Chapter 3) with the information gained from dataflow analysis. This increases the opportunities for determinism extraction. For example,

before analysis the clause:

```
a(X,X,X).
```

is transformed to the following kernel Prolog by making the head unifications explicit (i.e. ‘‘unraveling’’ the head unifications):

```
a(X,Y,Z) :- X=Y, X=Z.
```

If analysis derives that X is unbound and both Y and Z are nonvariable, then the above expansion hides the determinism by twice unifying an unbound variable with a nonvariable. Unraveling the head unifications again after analysis results in:

```
a(X,Y,Z) :- Y=Z, X=Y.
```

In this version, the nonvariables Y and Z are unified together, better exposing the deterministic check that is done, and the unbound variable X is only unified once.

6. Determinism transformation

This section groups four transformations that expose the determinism inherent in a predicate. The purpose of the first three transformations is to make the determinism in the predicate easily visible, so that the fourth transformation, determinism extraction, is as successful as possible in generating case statements. The following transformations are done in order:

(1) **Head-body segmentation**. By separating the heads of clauses from the clause bodies, this reduces the code expansion caused by type enrichment and determinism extraction.

(2) **Type enrichment**. This adds types to predicates for which global analysis is not able to determine the type. The compiler creates different versions of the predicate assuming different input types. This increases code size, but improves performance since often a predicate is deterministic at run-time even though this could not be detected at compile-time.

(3) **Goal reordering**. This reorders goals in a clause to expose more determinism. Tests (such as arithmetic relations) are moved to the left and predicates guaranteed to succeed (such as unifications with uninitialized variables) are moved to the right.

(4) **Determinism extraction with test sets**. This transformation converts the predicate into a nested case statement that makes its determinism explicit, so that a straightforward compilation to BAM code is possible.

6.1. Head-body segmentation

This transformation reduces the code expansion resulting from enrichment and determinism extraction. A predicate is split into a new predicate and a set of clause bodies. The new predicate contains only the goals of the original predicate that are useful for determinism extraction, i.e. all explicit unifications and tests (including type checking and arithmetic comparisons, see Table 4.11) in each clause starting from the head up to the first goal that is not in this category. The rest of the clause bodies are separated from the predicate. This is done to avoid code duplication in determinism extraction, since the same clause may occur in several leaves of the decision tree.

For example, the predicate:

```
p(A,B) :-
         ( var(A), p(A), q(A,C), t(C,D), u(D,B)
         ; A=b, r(A), s(A)
         ).
```

is transformed into:

```
p(A,B) :-
         ( var(A), '$d1'(A,B)
         ; A=b, '$d2'(A)
         ).


'$d1'(A,B) :- p(A), q(A,C), t(C,D), u(D,B).

'$d2'(A) :- r(A), s(A).
```

The new predicate consists only of those parts of the original predicate that are useful for extracting determinism. The determinism extraction is free to create a decision tree from the new predicate without worrying about duplicating the clause bodies at the leaves of the tree. The separated clause bodies are compiled once only, and the BAM transformation stage (Chapter 6) merges them with the decision tree, thus creating a decision graph.

The decision exactly where to split the clause bodies depends on several factors. All goals in the body are classified into two kinds: goals that are useful for extracting determinism (called ‘‘tests’’), and other goals. Then the split follows these rules: (1) Only those tests all of whose variables are in the head become part of the new predicate. (2) If the length of the clause body is less than a given threshold, then all of it becomes part of the new predicate.

Head-body segmentation interacts with type propagation. It often occurs that a clause body is called from several leaves in the decision tree with different types. In that case, it is compiled with a type that is the intersection of the types of the entry points. A complication arises when one of the leaves considers a variable to be uninitialized, and another leaf does not. In that case, the first leaf jumps to a piece of code to initialize the variable, and only afterwards jumps to the clause body.

6.2. Type enrichment

By looking at the type or the value of one or more arguments it is possible to reduce the set of clauses that have to be tried. Often the dataflow analysis is able to derive sufficiently strong types so that a good selection can be done, i.e. a deterministic predicate can be compiled efficiently. However, if the types given for the predicate are weak then a source transformation is done to enrich them. The enrichment consists of adding a test to check at run-time whether an argument is a variable or a nonvariable, and to branch to different copies of the predicate in each case.

The number of arguments that are enriched is given by the argument S of the compiler option `select_limit(S)`. Define a *good* predicate argument as one that is an argument of a unification not known to succeed always, i.e. in the unification neither argument is known to be unbound. An argument is *known* to be of a given type if the type is implied by the type formula. Whether or not enrichment is done is based on the following heuristic:

> **Enrichment Heuristic 1:** If the number of good arguments known to be nonvariable is less than the selection limit S, then choose the lowest numbered good argument that is not known to be nonvariable. Otherwise choose only the first argument, if it is a good argument and it is not known to be nonvariable.

This heuristic is applied recursively on enriched predicates. The default selection limit is always S=1. This default is justified given that (1) a selection limit S=1 already generalizes the first argument selection

of the WAM, and (2) compilation time and object code size increase rapidly with the selection limit. Even with S=1, the source transformation occasionally results in some duplicate code being generated. This is removed by the BAM transformation stage. When S=1 the heuristic is simpler:

> **Enrichment Heuristic 2:** If there exist no good arguments known to be nonvariable, then choose the lowest numbered good argument that is not known to be nonvariable. Otherwise choose the first argument, if it is a good argument and it is not known to be nonvariable.

This heuristic generalizes the first-argument selection of the WAM, i.e. it always does at least a first argument selection, but depending on the types that the predicate has (often derived from dataflow analysis) and the predicate itself (what kinds of head unifications it does), the amount of selection can be vastly greater. The heuristic may seem complex, but it is a natural way to make a predicate deterministic.

To show how enrichment works, consider the following predicate without type declarations:

```
a(a).
a(b).
```

It is transformed into:

```
a(A) :-    var(A), a_v(A).  % If A is unbound.
a(A) :- nonvar(A), a_n(A).  % If A is nonvariable.

a_v(a).      a_n(a).
a_v(b).      a_n(b).
```

The predicate `a/1` has been enriched with an unbound type (in `a_v/1`) and with a nonvariable type (in `a_n/1`). As another example, consider the definition without any type declarations:

```
member(X, [X|_]).
member(X, [_|L]).
```

In this case the heuristic picks the second argument, since the first one does no useful unifications. After enrichment, the predicate becomes:

```
member(X, L) :-    var(L), member_v(X, L).
member(X, L) :- nonvar(L), member_n(X, L).

member_v(...) :- (same as original definition)

member_n(...) :- (same as original definition)
```

The two tests `var(L)` and `nonvar(L)` determine which of the two dummy predicates to execute, `member_v/2` or `member_n/2`, and are compiled into a single conditional branch. This is a

consequence of the fact that the two tests are mutually exclusive, i.e. if one succeeds then the other fails and vice versa. Both `member_v/2` and `member_n/2` have the same definition as the original predicate, but they have different types for the second argument. The predicate `member_v/2` is compiled assuming the second argument is a variable. The predicate `member_n/2` is compiled assuming the second argument is a nonvariable. Both `member_v/2` and `member_n/2` are also targets of the factoring transformation (section 4).

Type enrichment can introduce a significant increase in code size if it is not handled carefully. In practice, the code size is kept small because: (1) the added types result in significantly smaller code for clause selection in each of the two dummy predicates, (2) before doing enrichment, head-body segmentation separates clause heads from the bodies, so that long clause bodies are not duplicated, and (3) the BAM transformation stage (Chapter 6) removes any remaining duplicate code. In a sense, the definitions are first ‘‘loosened up’’ by head-body segmentation and type enrichment to allow more optimization, and then later ‘‘tightened up.’’

6.3. Goal reordering

This transformation reorders goals in a clause to increase determinism and to reduce the number of superfluous unifications that are done. Goals that are useful in determinism extraction are put as early as possible, and goals that are certain to succeed (such as unifications with uninitialized variables) are put later.

The goals in a clause are classified in four categories: tests (Table 4.11), unifications with unbound variables, unifications with uninitialized variables, and other goals. The goals are reordered so that tests are first (for deterministic selection), followed by unifications with unbound variables (may be affected by aliasing), unifications with uninitialized variables (unaffected by aliasing, so they can safely be put last), and the other goals. The reordering takes into account the fact that unification is commutative, i.e. that unification goals can be permuted in any way without changing the semantics. Some reorderings are better than others because aliasing can worsen the type formula, e.g. if X is unbound (`var(X)`) then after performing the unification `Y=Z` it may not be unbound any more, if it is aliased to Y or Z. The reordering is

constrained so that aliasing does not change the operational semantics.

For example, consider a predicate that has an uninitialized argument:

```
:- mode((a(A,B,C) :- uninit(C))).
a(X, Y, Z) :- Z=[X|L], X<Y, ...
```

The transformation knows that the unification `Z=[X|L]` does not instantiate X or L because Z is unbound and unaliased. Therefore the unification is moved back:

```
a(X, Y, Z) :- X<Y, Z=[X|L], ..
```

This has two advantages: (1) the test `X<Y` is brought forward so that it can be used by determinism extraction, and (2) the unification `Z=[X|L]` is not done if the test `X<Y` fails.

This transformation compensates for the popular programming style which puts all unifications in the head and all tests in the body, e.g. people prefer to write:

```
a([X|L], [X|M]) :- var(X), ...
```

instead of:

```
a([X|L], Z) :- var(X), Z=[X|M], ...
```

The first version does not imply anything about the instantiation pattern of the arguments, whereas the transformed version does.

6.4. Determinism extraction with test sets

The majority of predicates written by human programmers are intended to be executed in a deterministic way. These predicates are in effect case statements, yet they are too often compiled in an inefficient manner, by means of shallow backtracking (i.e. saving the machine state, unification with the clause heads, and repeated failure and state restoration). This section describes the general technique used in the compiler to convert shallow backtracking into conditional branching.

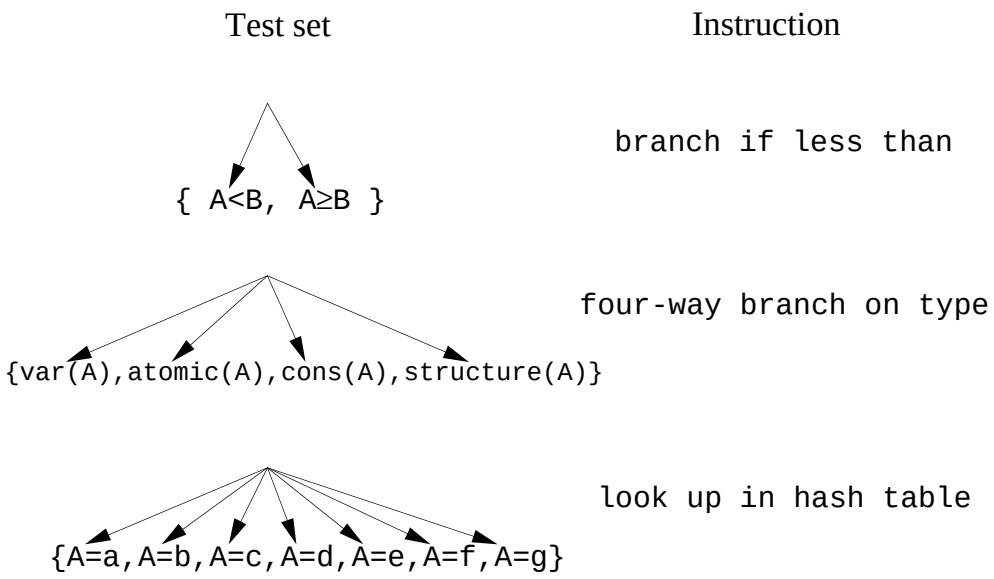

Figure 4.10 – Some examples of test sets

6.4.1. Definitions

Predicates are compiled into code which is as deterministic as possible through the concept of the *test set*. Two definitions are useful:

> **Definition ST:** A goal *G* is a *simple test* with respect to the kernel Prolog predicate *P* and the type formula *F* if it satisfies the following conditions:
>
> - *G* uses only variables that occur in the head of *P*.
> - The implementation of *G* does not change any state in the execution model, i.e. *G* does not cause side-effects (I/O or database operations), *G* does not create choice points, and *G* does not bind any variables.
> - *G* does not always succeed.
>
> **Definition TS:** A set of goals is a *test set* with respect to the kernel Prolog predicate *P* and the type formula *F* if it satisfies the following conditions:
>
> - Each goal in the set is a simple test according to definition ST.
> - With a given set of variable values, at most one goal in the set can succeed.
> - A multi-way branch in which each destination corresponds to the success of one of the goals in the set can be implemented in the target architecture.

The tests in the set need not actually be present in the definition of P. Whether or not a given set of goals is a test set depends on the architecture and the predicate P.

6.4.2. Some examples

Most conditional branches in an architecture correspond to a test set. For example, a branch-if-less-than instruction corresponds to the test set {A<B, A≥B}. More complex conditions such as an n-way branch implemented by hashing can also be represented as test sets. Figure 4.10 shows some examples of test sets. The second and third examples correspond to WAM instructions.

To illustrate the use of test sets, consider the predicate:

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

which is one way to calculate the maximum of A and B. It is compiled as:

```
max(A, B, C) :- if A>B then C=A
                else if A<B then C=B
                else (C=B or C=A)
```

(The Prolog notation is simplified for readability.) The predicate is executed completely deterministically if A>B or A<B; a choice point is created only when A=B. The choice point maintains the operational semantics: since both clauses of the original predicate succeed when A=B, there are two identical solutions.

---

type testset = testset(testset_name, testset_ident, set of goal);

function determinism(*D* : disjunction; *H* : goal; *F* : formula; *Previous* : set of testset) : disjunction;
var *TS* : testset;
TS$_{set}$: set of testset;
begin
if length(*D*) ≤ 1 **then return** *D*;
TS$_{set}$:= find_testsets(*D*, *H*, *F*, *Previous*);
if *TS*$_{set}$ = ∅ **then return** *D*;
TS := pick_testset(*TS*$_{set}$);
return code_testset(*TS*, *D*, *H*, *F*, *Previous*)
end;

Figure 4.11 – The determinism extraction algorithm

```
function find_testsets(D : disjunction, H : goal; F : formula; Previous : set of testset) : set of testset;
var   TS : testset;
      TS_set : set of testset;
      i , j : integer;
begin
      /* At this point D = (C_1 ; ... ; C_n) where D has n choices */
      TS_set := ∅;
      for i := 1 to n do begin
            /* C_i = (G_i1 , ... , G_in_i) where C_i has n_i goals */
            for j := 1 to n_i do
                  if G_ij = "!" then exit inner loop
                  else for all testsets TS from table do begin
                        /* TS = testset(Name , Ident , Tests) from Table 4.11 */
                        if TS ∉ Previous and vars(G_ij) ⊂ vars(H) and bindset(G_ij , F) = ∅ then
                              if ∃ T ∈ Tests : (G_ij implies T and not(F implies T)) then
                                    TS_set := TS_set ∪ {TS}
                  end
      end;
      return TS_set
end;
```

Figure 4.12 – Finding all test sets in a predicate

```
function pick_testset(TS_set : set of testset) : testset;
var   TS : testset;
begin
      pick TS ∈ TS_set such that
            ∀ U ∈ TS_set : goodness(TS) ≥ goodness(U);  /* From Equation (G) */
      return TS
end;
```

Figure 4.13 – Picking the best test set

6.4.3. The algorithm

Given a predicate, the compiler proceeds by first finding all test sets that contain tests that are implied by goals in the predicate. This depends on the type formula that is known for the predicate; for example, the unification `X=a` is only a test if X is nonvariable, i.e. if the type formula implies `nonvar(X)`. Then a ‘‘goodness’’ measure is calculated for each test set, and the test set with the largest goodness is used first. The goodness measure is calculated heuristically; in the current implementation each test set is weighted by an architecture-dependent goodness (which depends on how efficiently it is

```
function code_testset(TS : testset; D : disjunction; H : goal; F : formula; Previous : set of testset) : disjunction;
var   T : goal;
      Choices : disjunction;
begin
      Choices := [ ];
      /* At this point TS = testset(Name, Ident, Tests) */
      for all T ∈ Tests do begin
            D_test := subsume(T, D);
            D_test := determinism(D_test, H, update_formula(T, F), Previous ∪ {TS});
            append '$test'(T, D_test) to Choices;
            D := subsume(not(T), D)
      end;
      D := determinism(D, H, F, Previous ∪ {TS});
      append '$else'(D) to Choices;
      return '$case'(Name, Ident, Choices)
end;
```

Figure 4.14 – Converting a disjunction into a case statement

implemented in the architecture) and by the number of possible outcomes (e.g. hashing with a large number of cases is considered better than a two-way branch). The predicate is converted into a case statement using the best test set. The algorithm is called recursively for each arm of the case statement to build a decision tree. This tree is collapsed into a graph by the BAM transformation stage.

Figures 4.11 through 4.14 give a pseudocode definition of this algorithm. The figures define the function determinism(D, H, F, *Previous*) that performs the determinism extraction. Given a predicate written as a head H and a disjunction D, along with the type formula F that is true for that predicate, the function finds as many test sets as possible in the disjunction and converts them into case statements. It returns a new disjunction that contains these case statements. The parameter *Previous* is used to avoid infinite recursion. It contains all test sets that have already been used to make sure each test set is only used once.

The function find_testsets(D, H, F, *Previous*) returns a list of all test sets in the disjunction (Figure 4.12). It picks a test set if there is a goal in the predicate which implies a test in the test set. It limits the goals to those that do not bind any variables (bindset(G_{ij}, F) = ∅) and those that use only variables that occur in the head ($vars(G_{ij}) \subset vars(H)$). The function pick_testset(TS_{set}) returns the test set with the greatest measure of goodness, as given by Equation (G) (Figure 4.13). The function code_testset(TS, D,

H, F, *Previous*) converts the disjunction D into a case statement when given a test set *TS* (Figure 4.14). It uses the functions subsume(F, F_1) and update_formula(F, F_1), which are defined in section 3.

Table 4.11 – Test sets

Name	Example Test	Example BAM translation
equal	`X==Y` (X or Y is simple at run-time)	`equal(X,Y,Lbl)`
equal(atomic,A) (A is an atom)	`X==A`	`equal(X,A,Lbl)`
equal(structure,F/N) (F/N is name/arity)	`'$name_arity'(X,F,N)`	`equal([X],F/N,Lbl)`
hash(atomic)	`X=A` (A is atomic)	`hash(tatm,X,N,Lbl)`
hash(structure)	`X=S` (S is a structure)	`hash(tstr,X,N,Lbl)`
comparison(Class,Kind) (Class ∈ {eq, lts, gts}) (Kind ∈ {arith, unify, stand})	`X<Y`	`jump(lts,X,Y,Lbl)`
Type (Type ∈ AllTypes)	`var(X)`	`test(eq,tvar,X,Lbl)`
switch(Type) (Type ∈ TagTypes − {var})	`atom(X)`	`switch(tatm,X,L1,L2,L3)`

Table 4.11 lists the test sets currently recognized by the compiler. This includes unification goals, all type checking predicates, and all arithmetic comparisons. For each test set it gives the name, a representative test in the test set (only one is given, although usually there are several others), and the translation of that test into a conditional branch of the BAM instruction set. For the test sets hash(atomic) and hash(structure) the BAM code includes a hash table (not shown) in addition to the hash instruction. The following definitions simplify the table:

TagTypes = {var, atom, structure, cons, negative, nonnegative, float}, i.e. all types that correspond to one tag in the VLSI-BAM architecture.

AllTypes = TagTypes ∪ {atomic, integer, simple, compound}, i.e. it includes types that correspond to more than one tag.

The goodness measure for a test set in a predicate is calculated using the following rule:

$$Goodness = 1000 \cdot D + G \qquad \text{(G)}$$

where D is the number of directions of the test set that occur in the predicate and G is the raw goodness measure of the test set. This rule ensures that the number of useful directions in the testset is most important. The raw goodness is used only when the number of directions is the same. Table 4.12 gives the raw goodness of all test sets in the VLSI-BAM architecture [34], with a brief justification of the ranking. The

Table 4.12 – Raw goodness measure of test sets in the VLSI-BAM

Test set	Rank	Comments
switch(cons) switch(structure)	131	Switch is best because it is fast and it is a three-way branch, so it gives the most information. Switch of compound terms is better than other switches because it makes traversing a recursive term (like a list or a tree) fast.
switch(negative) switch(nonnegative) switch(atom)	130	Switch of atomic terms is worse because it penalizes the case of traversing a recursive term.
switch(integer)	129	Switch of integer is worse because the VLSI-BAM has separate negative and nonnegative (`tpos` and `tneg`) tags, requiring two branches.
var atom cons structure negative nonnegative	120	These test sets are types that correspond directly to tags, and there exist fast two-way branches on tags.
equal	85	This test set requires two instructions—a compare and branch, and also possibly loading its arguments into registers.
equal(atomic,_) comparison(_,_)	80	These test sets each require two instructions—a compare and branch.
integer atomic compound	79	These test sets are types that each correspond to two tags, so they need two tag checks.
equal(structure,_)	60	Equality comparison of a structure's functor & arity needs a memory reference.
simple	50	This test set corresponds to a type that needs five tag checks (four without floating point).
hash(atomic)	41	Hashing is the slowest because it needs to calculate the hash address.
hash(structure)	40	Hashing on a structure is slightly slower than hashing on an atomic term because a memory load is needed to access the main functor of the structure, whereas the atomic term is directly available in the register.

value of the rank is not important; only the relative order is important. Architectures rank the test sets according to how efficiently they are implemented in the architecture. To compile for a different architecture, only the ranking is changed in the compiler. The ranking is modified for other processors by a compiler option. For example, for the MIPS processor, the option `mips` changes the ranking to make the test set `equal(atomic,[])` best, i.e. a comparison with the atom `[]` (nil), because it can be implemented with a single-cycle conditional branch instruction. The MIPS does not have separate tags for negative and nonnegative integers, so the test sets negative and nonnegative are not implemented as efficiently as on the VLSI-BAM. These two test sets have lower ranks.

Chapter 5

Compiling Kernel Prolog to BAM Code

1. Introduction

The previous chapters described the conversion of standard Prolog to kernel Prolog and the optimizing kernel transformations. This chapter shows how the optimized kernel Prolog is compiled to BAM code. The compilation to BAM is performed in two steps for each predicate. In the first step, the control instructions that make up the framework of the predicate are compiled by the predicate compiler. This includes compiling the deterministic case statements into conditional branches and the disjunctions with choice point instructions.

In the second step, the clauses that make up the body of the predicate are compiled by the clause compiler. The clause compiler uses two primitives, the goal compiler and the unification compiler, to compile goals and explicit unifications. The clause compiler also does register allocation, entry specialization (replacing built-in predicates by faster entry points), and performs the write-once transformation (for fast trailing), and the dereference chain transformation (to maintain consistency with the dataflow analysis). These transformations are explained in detail in the sections below.

2. The predicate compiler

In the kernel transformation stage (Chapter 4), determinism extraction attempts to convert each predicate into a series of nested case statements. This is not always successful; sometimes the case statements still retain disjunctions (OR choices) that could not be converted into deterministic code. The predicate compiler compiles both the case statements and the disjunctions into BAM code. The case statements are compiled into conditional branches. The disjunctions are compiled into choice point instructions. The predicate compiler uses two primitives, the determinism compiler and the disjunction compiler, to compile the predicate's case statements and disjunctions.

2.1. The determinism compiler

Compiling a kernel Prolog predicate into deterministic BAM code is done in two steps. First, the determinism transformation (a kernel Prolog transformation, Chapter 4) converts a kernel Prolog predicate into a series of nested case statements. Then the determinism compiler compiles the nested case statements into BAM code. A case statement may contain any test set, and each test set is mapped to a conditional branch. The test sets and their corresponding conditional branches are given in Table 4.11.

2.2. The disjunction compiler

A disjunction (an OR formula) is a list of clauses that encapsulates a choice. The first clause is executed the first time the disjunction is encountered. The remaining clauses are executed in order on backtracking—each time backtracking returns to the disjunction the next clause is tried. This is implemented by code which generates choice points. A choice point encapsulates the state of the abstract machine at the time it is created. Backtracking restores machine state from a choice point to let execution continue from the point at which the choice point was created.

Creating and restoring machine state in choice points is time-consuming. To minimize the size of the choice points (and hence the time required to create them), the choice point management instructions in the BAM are streamlined to perform the least amount of data movement. They save only those registers that are needed in the clauses of the disjunction after the first, and for each clause of the disjunction they restore only those registers that are needed in that clause. Argument registers are restored in the clause itself and not in the `fail` instruction. Therefore the size of the choice point does not have to be stored in the choice point and decoded in the `fail` instruction. A disadvantage is a slightly larger code size.† Consider the following kernel Prolog for a predicate P with n clauses:

`Head :- (` C_1 `;` C_2 `; ...  ;` C_n `; fail).`

A single choice point is created for each invocation of P. The set of registers saved in the choice point is the set of all head arguments that are used in clauses after the first, i.e. C_2 through C_n. Arguments that

† This is less of a problem in the VLSI-BAM since the instruction reorderer merges pairs of single-word loads into double-word loads.

occur only in clause C_1 do not have to be stored in the choice point. The set of registers that is restored for each clause is the set of arguments used in that clause.

Before creating the choice point, the compiler dereferences those arguments that it can deduce will be dereferenced later. This avoids dereferencing the same argument more than once. The set of arguments to be dereferenced is derived by checking the type formula corresponding to each goal in the body of the predicate's definition, and noting whether its arguments have to be dereferenced. For example, arithmetic operations and relational tests are goals that require their arguments to be dereferenced.

To illustrate the compilation scheme, consider the following predicate:

```
p(A,B,C,D) :- ( a(A)
              ; c(C)
              ; d(D)
              ; fail
              ).
```

It is compiled as:

```
procedure(p/4).
        choice(1/3,[2,3],l(p/4,2)).   ; Save registers r(2) and r(3).
        jump(a/1).
label(l(p/4,2)).
        choice(2/3,[2,no],l(p/4,3)).  ; Restore only register r(2).
        move(r(2),r(0)).
        jump(c/1).
label(l(p/4,3)).
        choice(3/3,[no,3],fail).      ; Restore only register r(3).
        move(r(3),r(0)).
        jump(d/1).
```

The choice instructions do all the choice point manipulation: `choice(1/3,...)` creates the choice point, `choice(2/3,...)` modifies the address to return to on backtracking, and `choice(3/3,...)` removes the choice point. Register `r(0)` is not saved in the choice point because it is not needed in clauses beyond the first. The second and third clauses restore only the registers they need. Register `r(1)` is not saved because it is not needed at all.

Each choice instruction contains a list of the registers that it used. The length of the list is the same for all choice instructions in a predicate. For choices after the first, the atom `no` is put in the positions of registers that do not have to be restored. For example, the list `[0,no,5]` means that registers `r(0)` and `r(5)` are restored from the first and third locations in the choice point, and the second location is not

accessed.

In this example a further optimization can be done by merging the move instructions with the choice instructions, i.e.:

```
choice(3/3,[no,3],fail).
move(r(3),r(0)).
```

becomes:

```
choice(3/3,[no,0],fail).
```

This is possible because the value loaded in a register is determined by its position in the list, not by its number, and because register `r(3)` is only used to load `r(0)`.

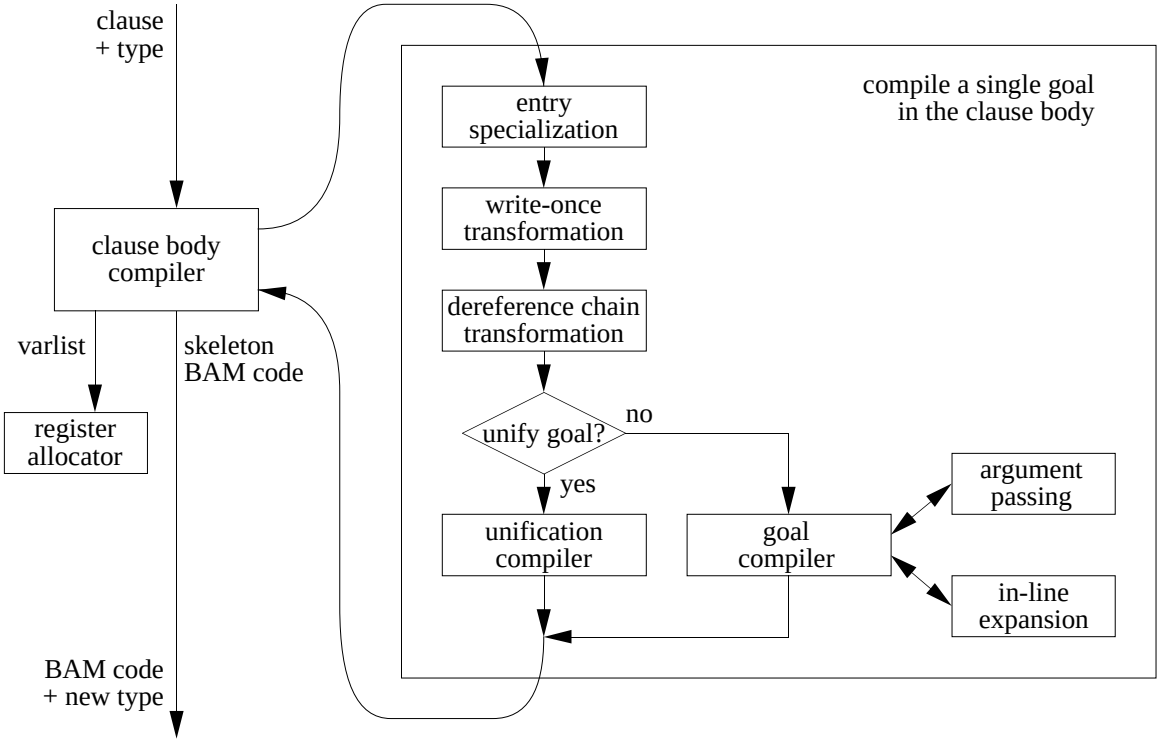

Figure 5.1 – Structure of the clause compiler

3. The clause compiler

The clause compiler converts a clause from kernel Prolog form (with type annotations) to BAM code. The structure of the clause compiler is given in Figure 5.1. After compiling the goals in the body there are two intermediate results: (1) BAM code in which variables have not yet been allocated to registers (*skeleton* code) and (2) a variable occurrence list (the *varlist*), that contains all unallocated variables in the skeleton code. The final BAM code is obtained by passing the varlist to the register allocator.

Each goal in the clause body is compiled in four steps. First, three transformations are performed on the goal: entry specialization, the write-once transformation, and the dereference chain transformation. Then the goal is compiled into BAM code by one of two routines, the unification compiler or the goal compiler, depending on whether the goal is a unification or not.

These are the important blocks in the clause compiler:

(1) **The goal compiler**. Its main task is to handle argument passing. Because of the interaction between the different kinds of unbound variables, initialized and uninitialized, this results in a case analysis. In addition, the goal compiler compiles in-line some built-in predicates and the dummy predicates that were created in the transformation to kernel Prolog.

(2) **The unification compiler**. Its task is, given a type, to compile an explicit unification into the simplest possible code.

(3) **The register allocator**. Its task is to allocate variables to registers in such a way that the number of superfluous move instructions is minimized. It uses a data structure called the *varlist* which is generated by the clause body compiler.

(4) **Entry specialization**. This attempts to replace each goal in the clause by a faster entry point, depending on the types known at the call.

(5) **Write-once transformation**. This transformation is part of a technique for reducing the overhead of trailing.

(6) **Dereference chain transformation**. This transformation is necessary to keep the dataflow analysis and the clause compiler consistent.

The following sections give more details about these each of these blocks. First, an example of a clause compilation is given, with emphasis on the skeleton code, the varlist, and a specification of the register allocator. This is followed by discussions of the goal compiler, the unification compiler, entry specialization, the write-once transformation, and the dereference chain transformation.

3.1. Overview of clause compilation and register allocation

This section gives an example of how a clause is compiled. Consider the following clause with no types:

```
a(A,B) :- b(A,C), d(C,B).
```

Compilation of this clause proceeds in three steps: First the kernel Prolog is compiled to BAM code and a variable occurrence list, or *varlist*. In this example, most of the work in this step is done in the goal compiler. The resulting BAM code is referred to as *skeleton* code since variables have not yet been allocated to registers. The varlist is derived from the skeleton code and contains the list of variables and registers in it. Second, the register allocator uses the varlist to allocate variables to registers. Third, after all predicates and all clauses are compiled, the BAM optimization stage improves the code (Chapter 6). The skeleton code for this clause is:

```
allocate(X).           ; Create an environment (its size is still unknown).
move(r(0),A).          ; Load the head arguments into variables A and B.
move(r(1),B).
move(tvar^r(h),C).     ; Create an unbound variable and put it in C and D.
move(tvar^r(h),D).     ; C may exist beyond a call, D exists between calls.
pragma(push(variable)).
push(D,r(h),1).
move(A,r(0)).          ; Load the parameters of the first call.
move(D,r(1)).
call(b/2).
pragma(tag(C,tvar)).   ; C has an extra link, with a tvar tag.
move([C],r(0)).        ; Extra indirection to remove the extra link.
move(B,r(1)).
call(d/2).
deallocate(X).         ; No last call optimization in the skeleton code.
return.
```

The varlist for this clause is:

```
[pref,r(0),A,          ; Corresponds to move(r(0),A).
 pref,r(1),B,
 C,pref,C,D,D,         ; Corresponds to the unbound variable in C and D.
 pref,A,r(0),
 pref,D,r(1),
 fence,                ; Corresponds to call(b/2).
 C,r(0),
 pref,B,r(1),
 fence]                ; Corresponds to call(d/2).
```

3.1.1. Construction of the varlist

The varlist is constructed to satisfy these conditions:

(1) The only contents of the varlist are unbound variables, temporary registers, and the atoms `fence` and `pref`.

(2) The order of variable occurrences is the same in the skeleton code and the varlist.

(3) The atom `fence` is inserted as a marker at each point where temporary variables do not survive. This corresponds to each `call(..)` instruction in the skeleton code.

(4) Two variables that are preferably allocated to the same register are preceded by the atom `pref` and called a *pref pair*. A pref pair is created when allocating the variables to the same register allows an instruction to be removed. For example, the `move(A,r(0))` instruction can be removed if the variable A is allocated to register `r(0)`.

(5) A variable occurs exactly once in the varlist if and only if it occurs exactly once in the skeleton code. Such a variable is called a void variable. An instruction containing a void variable may be removed.

(6) A variable occurs more than once in the varlist if and only if it occurs more than once in the skeleton code.

3.1.2. The register allocator

The register allocator assigns a register to each variable in the varlist such that there are no conflicts, i.e. a single register never holds two values at the same time. The allocator also calculates the size of the environment (the number of permanent registers) for the `allocate` and `deallocate` instructions. The algorithm is defined in Figure 5.2. It assumes that variables are represented as logical variables, i.e.

```
procedure register_allocator(VL : varlist);
var   V_void , V_temp , V_pref , V_perm : set of variable;
begin
      V_void := { variable Y | Y occurs exactly once in VL };
      ∀ X ∈ V_void do Allocate each X to r(void);
      V_perm := { variable Y | The sequence [Y, ..., fence, ..., Y] occurs in VL };
      ∀ X ∈ V_perm do Allocate each X to a different p(I);
      Environment size := number of elements in V_perm ;
      V_temp := { variable Y | Y occurs more than once in VL };
      V_pref := prefer(VL );
      while V_temp ≠ ∅ do begin
            while ∃ X ∈ V_pref : X is allocatable to r(I) without conflict do begin
                  Allocate X to its preferred register r(I);
                  V_pref := V_pref − {X };
                  V_temp := V_temp − {X };
                  V_pref := prefer(VL )
            end;
            if ∃ X ∈ V_temp then begin
                  Allocate X to the lowest r(I) possible without conflict;
                  V_pref := V_pref − {X };
                  V_temp := V_temp − {X };
                  V_pref := prefer(VL )
            end
      end
end;

function prefer(VL : varlist) : set of variable;
begin
      return { variable Y | The sequence [ pref, Y, _ ] or [ pref, _ , Y ] occurs in VL }
end;
```

Figure 5.2 – The register allocator

that allocating a variable to a register binds that variable in all sets that contain it. It assumes that there are an infinite number of temporary and permanent registers. It uses the following correspondence between variable lifetimes and registers:

(1) A variable that occurs exactly once is allocated to `r(void)`.

(2) A variable occurring on both sides of a `fence` marker (it *crosses* a fence) is allocated to a permanent register `p(I)` (a location in the environment).

(3) A variable that does not cross a fence and that occurs more than once is allocated to a temporary register `r(I)`.

The algorithm is independent of the write-once transformation and the dereference chain transformation.

This is possible because the clause compiler is careful to feed the allocator a varlist that takes the two transformations into account.

In the example of the previous section, the allocator assigns the following values to the variables:

```
A = r(0)
B = p(0)
C = p(1)
D = r(1)
X = 2
```

Since both B and C cross a fence, they are allocated to permanent registers. Both A and D are allocated to their preferred registers. The number of permanent variables, X, is 2.

3.1.3. The final result

The final BAM code output by the compiler after all transformations and optimizations (including the BAM transformations of chapter 6) is:

```
allocate(2).           ; Allocate space for two permanent variables.
move(r(1),p(0)).
move(tvar^r(h),r(1)).  ; Create an unbound variable and put it in r(1) and p(1).
move(r(1),p(1)).
pragma(push(variable)).
push(r(1),r(h),1).
call(b/2).
pragma(tag(p(1),tvar)).
move([p(1)],r(0)).     ; Indirection due to dereference chain transformation.
move(p(0),r(1)).
deallocate(2).
jump(d/2).             ; Last call optimization converts 'call' to 'jump'.
```

3.2. The goal compiler

Given a goal and type information about the goal, this module sets up the arguments to call the goal, does the call, and sets up the return arguments. The main task of the goal compiler is to handle the complexities that arise when supporting combinations of uninitialized and initialized parameters. The following situations are also handled:

(1) **Duplicate variables**. An uninitialized variable that occurs twice in a goal must be initialized before calling the goal.

(2) **Uninitialized register variables**. Passing arguments as uninitialized register variables requires some care. These variables are not passed into a predicate, but are outputs returned in registers.

(3) **Dummy predicates**. Several compiler transformations create new predicates as part of the transformation. These predicates are only called once, so they are compiled in-line.

(4) **Built-in predicates**. Some built-in predicates are translated into in-line code (Table 5.5).

```
function compile_goal(G : goal; F : formula; V_sf : set) : return (Code : list; F_out : formula; V_sf,out : set);
var   V_uninit , V_init : set of variable;
      Initcode , Precode , Call , Postcode : list of instruction;
      A : term;
      g_i , r_i : {ini, mem, reg};
      i : integer;
begin
      /* Initialize all uninitialized variables that are duplicated */
      V_uninit := { X | F implies (uninit_mem(X) or uninit_reg(X)) };
      V_init := ((vars(G) – V_sf) ∪ V_uninit) ∩ dups(G); /* Table 4.6 */
      Initcode := list of ( ∀ X ∈ V_init : Code to initialize the variable X);

      /* Pass arguments to the goal and clean up afterwards */
      Precode := [ ];
      Postcode := [ ];
      for i := 1 to arity(G) do begin
            A := (argument i of goal G);
            g_i := given_flag(A , F , V_sf ); /* Table 5.1 */
            r_i := require_flag(A , G); /* Table 5.2 */
            Append precode[ g_i , r_i ] to Precode ; /* Table 5.3 */
            Append postcode[ g_i , r_i ] to Postcode /* Table 5.4 */
      end;

      /* Call the goal */
      if (G can be expanded in-line) then
            Call := (in-line expansion of G) /* Table 5.5 */
      else if (G is a dummy predicate) then
            Call := (in-line compilation of G's definition)
      else if (G does not alter temporary registers) then
            Call := (a simple_call instruction for G) /* Table 3.7 */
      else
            Call := (a call instruction for G);
      Code := append(Initcode , Precode , Call , Postcode)
end;
```

Figure 5.3 – The goal compiler

The function compile_goal(G, F, V_{sf}) defines the goal compiler (Figure 5.3). Its inputs are the goal (G), a type formula (F), and the set of variables that have a value on input (V_{sf}). Its outputs are a list of BAM instructions (*Code*), the type formula true on output (F_{out}), and the set of variables that have a value on

output ($V_{sf,out}$).

Each goal has three type formulas associated with it: a *Require* type, a *Before* type, and an *After* type. These types are optionally given by programmer input and are supplemented by dataflow analysis. The compiler maintains a table of these types for all predicates including built-ins and internals. The Require type gives the types that the arguments being passed to the goal must have, i.e. the goal compiler is required to make them true in all cases. The Before type gives the types that are true before the call. The After type gives the types that are true after the call returns. No special action is needed by the goal compiler to ensure the validity of the Before and After types.

Compiling a goal is made more complex because the kind of argument needed by the goal may not be the same as the one that is given to it. The goal's *Given* type (which is valid before the goal and given by F in Figure 5.3) must be reconciled with the goal's Require type. The most common Require and Given types are the three varieties of unbound variables: uninitialized memory and register variables and initialized variables. This requires a case analysis with 3×3 cases for each argument of the goal to properly match the Require and Given types.

Table 5.1 – Calculating the Given flag of an argument

Condition on argument A	g_i
nonvar(A)	`ini`
var(A) $\wedge$ (F implies uninit_mem(A))	`mem`
var(A) $\wedge$ (($A \notin V_{sf}$) $\vee$ (F implies uninit_reg(A)))	`reg`
var(A) $\wedge$ ($A \in V_{sf}$)	`ini`

Table 5.2 – Calculating the Require flag of an argument

Condition on argument A	r_i
require(G) implies uninit_mem(A)	`mem`
require(G) implies uninit_reg(A)	`reg`
otherwise	`ini`

Table 5.3 – Calculating the precode from the flags

g_i	r_i	precode[g_i , r_i]
reg	reg	[]
mem	reg	[]
ini	reg	[]
reg	mem	[move(tvar^r(h),*B*),adda(r(h),1,r(h))]
mem	mem	[]
ini	mem	[move(tvar^r(h),*B*),adda(r(h),1,r(h))]
reg	ini	[move(tvar^r(h),*B*),push(*B*,r(h),1)]
mem	ini	[move(*A*,[*A*]),move(*A*,*B*)]
ini	ini	[]

Table 5.4 – Calculating the postcode from the flags

g_i	r_i	postcode[g_i , r_i]
reg	reg	[]
mem	reg	[move(*B*,[*A*])]
ini	reg	unify(*A*,*B*)
reg	mem	[move(*B*,*A*)]
mem	mem	[]
ini	mem	unify(*A*,*B*)
reg	ini	[move(*B*,*A*)]
mem	ini	[]
ini	ini	[]

Require and Given flags r_i and g_i (with values in {ini, mem, reg}) are associated with each goal argument for the Require and Given types. Tables 5.1 and 5.2 define how the Require and Given flags are calculated. The function require(G) in Table 5.2 is a defined predicate in the compiler that returns the Require type for any goal. It knows all about built-in and internal predicates and the results of dataflow analysis.

Duplicate arguments (e.g. A in the call p(A, A)) are treated specially. An argument that is duplicate cannot be uninitialized—it occurs in more than one place, so it is not unaliased any more. The goal compiler initializes these arguments before doing the case analysis.

Table 5.3 gives the precode, i.e. the code that is generated before the call to set up, and Table 5.4 gives the postcode, i.e. the code that cleans up after the call. To enforce the Require type, in seven of the nine cases a different argument B is passed to the call instead of the goal's original argument A. For example, if the Given flag is mem and the Require type is reg, then the compiler must create a new variable B of type uninit_reg(B) to pass to the goal. After the goal returns, the original argument A and the returned argument B are unified together. The new variable B is created for all combinations of Given and

Require flags except (`reg , reg`) and (`mem , mem`). In these two cases no precode or postcode is needed.

To simplify the presentation, Figure 5.3 only does part of what the algorithm implemented in the compiler does. The definition of compile_goal in the figure only handles Require and Given types that are all uninitialized variables. The actual algorithm handles any types. The type formula F and the variable set V_{sf} are updated continuously during the execution of compile_goal. A variable occurrence list is calculated for the register allocator. The actual algorithm handles 12 cases for parameter passing instead of 9—as an optimization, two varieties of Given uninitialized register types are recognized.

Table 5.5 – BAM expansion of internal built-ins

Kernel Prolog	BAM instruction
`'$cut_load'(X)`	`move(r(b),X)`
`'$cut'(X)`	`cut(X)`
`'$name_arity'(X,'.',2)`	`test(ne,tlst,X,fail)`
`'$name_arity'(X,Na,Ar)`	`equal([X],tatm^(Na/Ar),fail)`
`'$name_arity'(X,Na,0)`	`equal(X,tatm^Na,fail)`
`'$test'(X,Types)`	(a sequence of `test` instructions)
`'$equal'(X,Y)`	`equal(X,Y,fail)`
`'$add'(A,B,C)`	`add(A,B,C)`
`'$sub'(A,B,C)`	`sub(A,B,C)`
`'$mod'(A,B,C)`	`mod(A,B,C)`
`'$mul'(A,B,C)`	`mul(A,B,C)`
`'$div'(A,B,C)`	`div(A,B,C)`
`'$and'(A,B,C)`	`and(A,B,C)`
`'$or'(A,B,C)`	`or(A,B,C)`
`'$xor'(A,B,C)`	`xor(A,B,C)`
`'$sll'(A,B,C)`	`sll(A,B,C)`
`'$sra'(A,B,C)`	`sra(A,B,C)`
`'$not'(A,C)`	`not(A,C)`

3.2.1. An example of goal compilation

This section gives a simple example of compilation to show how the goal compiler works in practice. Consider the following predicate in standard Prolog:

```
a(X, Y) :- Y is X+1.
```

This is converted to kernel Prolog:

```
a(X, Y) :- '$add'(X, 1, Y).
```

To compile the call to `'$add'/3` it is necessary to pass parameters in the right way. In particular, it is necessary to pass the output of the addition into variable Y. The built-in `'$add'(A,B,C)` has the

following types associated with it:

Require = `(deref(A),deref(B),uninit_reg(C))`.
After = `(integer(A),integer(B),integer(C),rderef(A),rderef(B),rderef(C))`.

From the Require type, the first two arguments X and 1 of `'$add'/3` must be dereferenced and the third argument Y must be an uninitialized register. The Given types of X and Y depend on the type formula for `a(X,Y)`. Assume first that no type is given for `a(X,Y)`. From Tables 5.1 and 5.2, the Given flag for Y is `ini` and the Require flag for Y is `reg`. From Tables 5.3 and 5.4, the precode in this case is empty and the postcode is a call to unify(A, B) to generate unification code. The compiled BAM code is:

```
procedure(a/2).
    deref(r(0),r(0)).               ; Dereference X.
    add(r(0),1,r(0)).               ; Perform the addition.
    deref(r(1),r(1)).               ; Dereference Y.
    unify(r(0),r(1),nonvar,?,fail). ; Unify Y with the result of the addition.
    return.
```

If `a(X,Y)` has a type then the code can often be simplified. For example, assume that its type is `(deref(X),uninit_mem(Y))`, i.e. X is dereferenced and Y is an uninitialized memory variable. Then the Given flag for Y is `mem`. The compiled BAM code is:

```
procedure(a/2).
    add(r(0),1,r(0)).               ; Perform the addition (X is dereferenced).
    pragma(tag(r(1),tvar)).
    move(r(0),[r(1)]).              ; Bind Y to the result of the addition.
    return.
```

3.3. The unification compiler

This section gives an overview of the compilation of unification, the optimizations that are done, and several examples.

3.3.1. The unification algorithm

Given a unification goal and type information about its arguments, this algorithm generates the simplest possible code to implement the unification. In the general case, the algorithm builds a tree of instructions. Each node of the tree has three branches—one each for read mode and write mode unification, and one for failure. The algorithm generates dereference instructions if necessary and trail instructions to undo

variable bindings when backtracking. It does other optimizations including optimal write mode unification, type propagation, and depth limiting.

Write mode unification of a term generates a block of push instructions that builds the term on the heap. Read mode unification of a term is done sequentially for each of the term's arguments. First it checks the name and arity of the term. Then the arguments are unified. For arguments that are simple terms this consists of a single `move`, `equal`, or `unify` instruction. For arguments that are compound terms the unification algorithm is called recursively.

The function unify(X, Y, F, V_{sf}) defines the unification algorithm (Figure 5.4 and 5.5). Its inputs are the two terms to be unified (X and Y), the type formula true on input (F), and the set of variables that have a value on input (V_{sf}). Its outputs are a list of BAM instructions (*Code*), the type formula true on output (F_{out}), and the set of variables that have a value on output ($V_{sf,out}$).

The algorithm does several tasks that are not shown in the figure since they would unnecessarily complicate the presentation. The instruction list, the type formula, and the variable set are updated continuously during the compilation. Before using the value of a variable, it is dereferenced if necessary. Before binding a value to a variable, it is trailed if necessary. A variable occurrence list (varlist) is calculated for the register allocator (Figure 5.2).

3.3.2. Optimizations

The actual implementation does four optimizations not shown in Figure 5.4 and 5.5. It does optimal write mode unification. It keeps track of terms that are ground and recursively dereferenced to avoid compiling superfluous write mode unifications and dereferences. To reduce code size, it performs the last argument optimization and the depth limiting transformation.

3.3.2.1. Optimal write mode unification

The algorithm is modified to build a compound term in write mode with the least number of move instructions. First the code for building the main functor with empty slots for its arguments is generated. This is followed by the code for building the arguments and filling in the slots with the correct heap offsets.

```
function unify(X, Y: term; F : formula; V_sf : set) return (Code : list; F_out : formula; V_sf,out : set);
begin
    Code := [ ];
    if (var(X) and var(Y)) then begin
        if (F implies (unbound(X) or unbound(Y))) then
            Compile a store instruction
        else
            Compile a call to a general unification subroutine;
        return
    end else if (nonvar(X) and nonvar(Y)) then begin
        Compile a check that X and Y have the same functor and arity a;
        for i := 1 to a do begin
            Append unify(X_i, Y_i, F, V_sf) to Code
        end;
        return
    end if (nonvar(X) and var(Y)) then Swap X and Y
    else if (var(X) and nonvar(Y)) then Do nothing;

    if (X ∉ V_sf) then return unify_write(X, Y, F, V_sf);
    else begin /* At this point X ∈ V_sf */
        if (F implies nonvar(X)) then return unify_read(X, Y, F, V_sf)
        else if (F implies var(X)) then return unify_write(X, Y, F, V_sf)
        else begin
            Compile a three-way conditional branch comparing the tags of X and Y;
            Call unify_read and unify_write to compile the read and write mode branches
        end
    end
end;
```

Figure 5.4 – The unification compiler: the main routine

This technique was proposed as an optimization over the WAM by André Mariën [44]. The examples of unification given later use this technique. The justification of the BAM instructions needed for unification was done with this technique (Chapter 3).

3.3.2.2. Last argument optimization

This is an important optimization that significantly reduces the code size. It can be performed whenever a compound term has a compound term in its last argument. Without this optimization, the tree generated by the algorithm has the same depth as the term that is compiled. For each level in the tree a new block of write mode code is generated. For lists of n elements this results in $O(n^2)$ move instructions. The optimization reduces the code size to $O(n)$ by creating only a single write mode block, and letting all depths of the tree jump into it. This optimization was proposed by Mats Carlsson [14]. The code for write

```
function unify_write(X, Y: term; F : formula; V_sf : set) return (Code : list; F_out : formula; V_sf,out : set);
begin
    /* At this point X is an unbound variable */
    Generate a block of instructions to create the term Y on the heap;
    Bind X to this block (i.e. generate code to dereference X if necessary,
    store a pointer to this block in X, and trail X if necessary)
end;

function unify_read(X, Y: term; F : formula; V_sf : set) return (Code : list; F_out : formula; V_sf,out : set);
begin
    /* At this point Y is a nonvariable and F implies nonvar(X) */
    Code := [ ];
    Compile a check that X contains a structure of same functor and arity as Y;
    for i := 1 to arity(Y) do begin
        Append unify(X_i, Y_i, F, V_sf) to Code
    end
end;
```

Figure 5.5 – The unification compiler: read and write mode unification

mode unification of a nested term is replaced by a single jump instruction to the write mode code block of the outermost term. An example of unification given below uses this optimization.

3.3.2.3. Type propagation

There are two ways in which propagating type information during the compilation of unification improves the code. First, during the unification, the algorithm keeps track of the variables that are ground, uninitialized, and recursively dereferenced. This information is propagated into the arguments of compound terms. The propagation of ground and recursively dereferenced types was added after measurements of the dataflow analyzer showed that these types are numerous.

Second, when a new variable is encountered in a term, then the unification compiler has the choice whether to create it as an initialized variable or as an uninitialized variable. It is not always best to create new variables as uninitialized, since this often makes it impossible to apply last call optimization. To solve this problem it is necessary to look ahead in the clause. The variable is created as uninitialized only if there is a goal later in the clause with this variable in an argument position that must be uninitialized.

3.3.2.4. Depth limiting

Because the unification compiler generates a separate read and write mode branch for each functor in the term that is unified, deeply nested terms result in a code size explosion. The last argument optimization (see above) reduces the code size when the nesting occurs in the last argument. For other cases, a different technique is necessary. The unification compiler replaces a deeply nested subterm by a variable, creates the subterm with write mode unification and does a general unification with the variable. The depth limit is set by the compiler option `depth_limit(N)`, and the default depth is `N=2`. For example, consider the following unification where the complicated term `z(...)` is nested deeply:

```
X=s(t(u(...z(...)...)))
```

It is replaced by a sequence of three unifications:

```
X=s(t(u(...A...))), B=z(...), A=B
```

The variable B does not yet have a value, so the unification `B=z(...)` is executed in write mode. A general unification is performed for `A=B`. Since the size of a write mode unification is linear in the size of the compound term, this considerably shortens the code for deeply nested terms. Measurements were done to determine the effect of this transformation on execution time. In most cases it is insignificant, e.g. for the nand benchmark (Chapter 7), a program that contains deeply nested structures, the difference in execution time between depth limits of two and three is insignificant (i.e. only a few cycles out of several hundred thousand).

3.3.3. Examples of unification

Consider the following sample clause:

```
a(A, s(A,[X|X])).
```

The WAM code for this clause is (assuming the two arguments of the clause are in registers `r(0)` and `r(1)`):

```
procedure  a/2                    ;; the clause has two arguments.
    get_structure  s/2,r(1)       ;; unify r(1) with s(A,[X|X]).
    unify_value  r(0)             ;; unify the first argument with r(0).
    unify_variable  r(3)          ;; load the second argument into r(3).
    get_list  r(3)                ;; unify r(3) with [X|X].
    unify_variable  r(2)          ;; load the first argument into r(2).
    unify_value  r(2)             ;; unify the second argument with r(2).
    proceed                       ;; return to caller.
```

Temporary values are stored in registers `r(2)` and `r(3)`. The execution time of this code averaged over read and write mode is 63 cycles on the Xenologic X-1 processor [85], an implementation of the PLM architecture [28]. The BAM code generated for the same clause is (the pragmas have been left out for clarity):

```
procedure(a/2).
    deref(r(1),r(1)).                          ;; dereference r(1).
    switch(tstr,r(1),l(a/2,3),l(a/2,4),fail). ;; three-way branch.
label(l(a/2,3)).                               ;; write mode for s(A,[X|X]).
    trail(r(1)).                  ; conditionally push r(1) on trail stack.
    move(tstrˆh,[r(1)]).          ; bind s(A,[X|X]) to second argument.
    push(tatmˆ(s/2),h,1).         ; create the term s(A,[X|X]).
    push(r(0),h,1).
    push(tlstˆ(h+2),h,1).
    pad(1).
label(l(a/2,1)).                  ; common code for last arg. opt.
    move(tvarˆh,r(2)).            ; create the two arguments of [X|X].
    push(r(2),h,1).
    push(r(2),h,1).
    return.
label(l(a/2,4)).                               ;; read mode for s(A,[X|X]).
    equal([r(1)],tatmˆ(s/2),fail).; check functor & arity of s/2.
    move([r(1)+1],r(3)).          ; load first argument into r(3).
    deref(r(3),r(3)).
    deref(r(0),r(0)).
    unify(r(3),r(0),?,?,fail).    ; unify first argument with r(0).
    move([r(1)+2],r(0)).          ; load second argument into r(0).
    deref(r(0),r(0)).
    switch(tlst,r(0),l(a/2,6),l(a/2,7),fail). ;; three-way branch.
label(l(a/2,6)).                               ;; write mode for [X|X].
    trail(r(0)).
    move(tlstˆh,[r(0)]).
    jump(l(a/2,1)).               ; jump to common code (last arg. opt.).
label(l(a/2,7)).                               ;; read mode for [X|X].
    move([r(0)],r(2)).
    move([r(0)+1],r(0)).
    deref(r(0),r(0)).
    deref(r(2),r(2)).
    unify(r(0),r(2),?,?,fail).    ; unify arguments of [X|X].
    return.
```

Again, the two arguments of the clause are in registers `r(0)` and `r(1)` and temporary values are stored in registers `r(2)` and `r(3)`. To reduce the code size, the write mode code for `[X|X]` jumps into the

middle of the code for `s(A,[X|X])`. With this optimization the code is 29 BAM instructions long (after translation and instruction reordering, this is 264 bytes on the VLSI-BAM). The WAM code is only 7 instructions long (17 bytes on the PLM) because each instruction encapsulates a choice. WAM instructions for unification assume the existence of a read/write mode bit in the implementation, which collapses the execution tree onto itself.

The code size ratio VLSI-BAM/PLM is large for this example. It was hoped during development that (1) code expansion would be less for other kinds of Prolog code (e.g. calls, parameter passing, backtracking), and (2) dataflow analysis would reduce the complexity of unifications. These intuitions have been borne out (Chapter 7): the static code size in VLSI-BAM bytes measured for large programs is only three times that of the PLM, a microcoded WAM with a byte-coded instruction set.

The execution time of the above code on the VLSI-BAM is 25 cycles (measured with a simulator taking pipeline delays into account and averaged over read and write mode). This is about 40% of the cycles needed for the X-1. This time can be estimated by taking the average execution times of BAM instructions when translated to the VLSI-BAM architecture: `unify` takes 5 cycles, `equal` takes 3 cycles, `switch`, `deref`, `trail`, and `move` from memory take 2 cycles each, `push`, `adda`, and all other `move` instructions take 1 cycle each, and `pad` instructions take 0 cycles because they are collapsed into the pushes. These estimates are only approximately correct because of instruction reordering optimizations performed on VLSI-BAM code.

Through programmer annotation or dataflow analysis it is sometimes possible to know the type of an argument at compile-time. For example, sometimes it is known whether an argument is unbound or bound. Consider the same sample clause again:

```
a(A, s(A,[X|X])).
```

Assume it is known that the second argument is an uninitialized memory variable. This is expressed with the following type declaration:

```
:- mode((a(A,B):- uninit_mem(B))).
```

With this type the clause's code is only 9 BAM instructions long (36 bytes on the VLSI-BAM):

```
procedure(a/2)
    move(tstr^h,[r(1)]).    ;; bind s(A,[X|X]) to second argument.
    push(tatm^(s/2),h,1).   ;; create the term s(A,[X|X]).
    push(r(0),h,1).
    push(tlst^(h+2),h,1).
    pad(1).
    move(tvar^h,r(0)).      ;; create the two arguments of [X|X].
    push(r(0),h,1).
    push(r(0),h,1).
    return.                 ;; return to caller.
```

The execution time of this example is 11 cycles.

3.4. Entry specialization

For each goal in the clause, the clause compiler attempts to replace it with a faster entry point, depending on the types existing at that point. For example, if it is known that the arguments N and A of the predicate `functor(X,N,A)` are atomic then a faster version can be compiled.

Entry specialization is done in both the clause compiler and the dataflow analysis. Doing it in both places is complementary since the analysis only keeps track of a limited set of types: ground, nonvariable, uninitialized, and recursively dereferenced. During clause compilation more information is known, for example, if the goal `X<Y` occurs in a clause, then afterwards it is known that `X<Y` is true. Analysis does not have a representation for this information, but it could be useful for entry specialization.

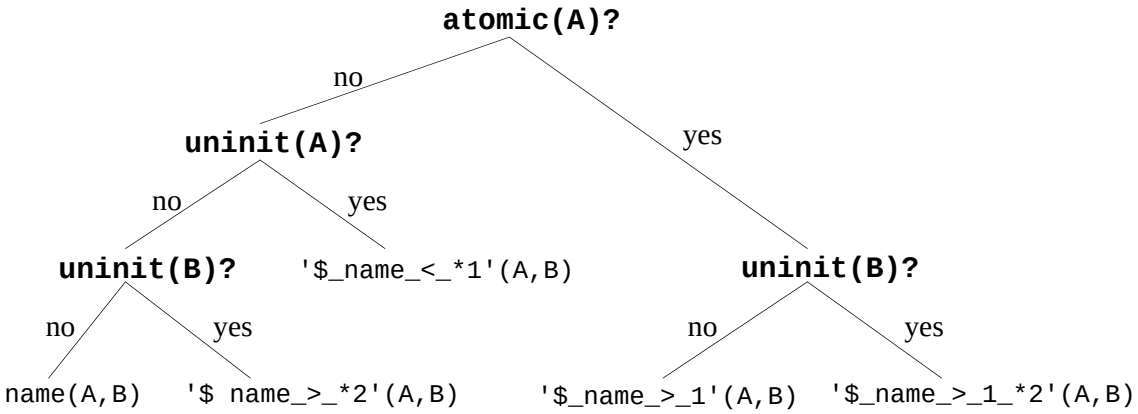

Figure 5.6: Example of a modal entry tree for entry specialization

Entry specialization can be done for any predicate whose definition is not in the program. The system has implemented this for the built-in predicates, but it can be used by the programmer for any library predicate. For each predicate that has faster entry points, a `modal_entry` declaration is given, along with type declarations for the fast entry points. These declaration are used in the dataflow analysis and the clause compiler to replace any call to the predicate with a faster entry point. For example, here is the modal entry declaration for the `name(A,B)` built-in predicate:

```
:- modal_entry(name(A,B),
   mode(atomic(A),
        mode(uninit(B),
             entry('$ name > 1 *2'(A,B)),
             entry('$ name > 1'(A,B))
            ),
   mode(uninit(A),
        entry('$ name < *1'(A,B)),
        mode(uninit(B),
             entry('$ name > *2'(A,B)),
             entry(name(A,B))
            )
       ))).
```

This declaration defines a binary tree, depicted in Figure 5.6. The nodes of the tree are decision points containing a type. If the type is valid then the left subtree is chosen, otherwise the right subtree is chosen. The leaves of the tree are the entry points. If none of the types are valid then the leftmost leaf is chosen, which

usually is the same predicate as the original one. Each of the four fast entry points also has a type declaration:

```
:- mode('$ name > 1'(A,B),     (deref(A),deref(B)),  atomic(A),
                               (list(B),ground(B)), n).
:- mode('$ name < *1'(A,B),    (uninit(A),deref(B)), true,
                               (atomic(A),deref(A),list(B),ground(B)), n).
:- mode('$ name > *2'(A,B),    (deref(A),uninit(B)), true,
                               (atomic(A),list(B),ground(B),rderef(B)), n).
:- mode('$ name > 1 *2'(A,B), (deref(A),uninit(B)), atomic(A),
                               (list(B),ground(B),rderef(B)), n).
```

These declarations are written in a five-argument form that is more general than a standard type declaration (Appendix A): it gives the entry types (both Require and Before) and the exit (After) types for the predicate.

3.5. The write-once transformation

In the BAM all unbound variables are kept on the heap. This makes trail checking significantly faster. However, when combined with the ability to destructively modify the value of permanent variables (e.g. to dereference them and save the dereferenced value in the permanent) it leads to several problems. These problems are all neatly resolved by the write-once transformation.

Putting all unbound variables on the heap means that there are no pointers to the environment/choice point stack; all pointers point to the heap. This reduces trail checking to a single comparison with the heap backtrack pointer `r(hb)` and a conditional push to the trail stack. It is not necessary to do another comparison to decide whether the variable is on the heap or in an environment. In addition, since all unbound variables are created on the heap there are no ‘‘unsafe variables’’ as in the WAM. An unsafe variable is an unbound variable that is created on the environment and that must be moved to the heap (‘‘globalized’’) before last call optimization deallocates its memory.

Modifying the value of a permanent variable (e.g. by dereferencing or binding it) cannot be done without a trail operation. Indeed, consider the case where a permanent dereferences to a nonvariable term. If the dereferenced value overwrites the original value, then both the original value and its address have to be trailed since backtracking has to restore the original value. This is expensive, since it has to be done

every time a permanent is bound or dereferenced.

One solution to this problem is never to store a dereferenced permanent back in the environment. This solves the problem but it is inefficient since a permanent may have to be dereferenced several times in a clause.

A better solution is to allocate a new permanent on the environment whenever the value of an old one needs to be changed. The new permanent gets the new value and the old permanent is unchanged. As a result, all permanent variables are only given values once, so they are called ‘‘write-once’’ permanents. Because it is not changed, the old permanent does not have to be trailed. At the cost of a slightly bigger environment, this completely eliminates the need to trail permanent variables. This allocation scheme is implemented in the clause compiler.

To summarize:

(1) All unbound variables are created on the heap, and unbound permanent variables in an environment always point to the heap.

(2) The trail check is a single comparison with `r(hb)` and a conditional push to the trail stack (2 cycles on the VLSI-BAM).

(3) Permanent variables are only given a single value in a clause. Whenever a permanent would be changed, a new one is allocated and given the modified value.

(4) Register allocation must allocate a different permanent register for each permanent variable in the clause. It is not allowed to use the same register for two variables whose lifetimes do not overlap.

This solution is implemented in the clause compiler by mapping a permanent variable onto a new variable whenever its value would change. The register allocator treats the new variables just like any other, and allocates them to temporary or permanent registers.

The main disadvantage of this technique is that environments are larger. For example, consider a clause of the form:

```
e(A,E) :- a(A,B), b(B,C), c(C,D), d(D,E).
```

where variables are chained from one predicate to the next. In the WAM, it is allowed to allocate

permanent variables such that variables whose lifetimes do not overlap are allocated to the same permanent register. For the above example, this requires just two permanent registers, so the total environment size is four words (it also includes registers `r(e)` and `r(cp)`). Only two permanents are needed no matter how long the chain of body goals is. This method requires trailing of the permanent's values, because backtracking must see the original values. This scheme is consistent with the original implementation of the WAM, i.e. binding permanent variables on the environment and globalizing unsafe variables to ensure correctness.

In contrast, the number of permanent variables needed by the write-once technique increases linearly with the length of the chain. For the above example, this requires four permanent variables, so the total environment size is six words. The total memory usage is increased by less than this amount because no trailing of permanents is needed.

This is an example of a trade-off between memory space and execution time. The extra memory space needed is comparable to the increased size of the trail stack if there is no trail check for permanent variables. Since this is small, I have opted to decrease execution time at the expense of larger environments. By keeping all unbound variables on the heap and by implementing permanent variables as write-once variables, permanent variables can be dereferenced and bound without trailing, and the cost of trailing heap variables is reduced to a single comparison and conditional push.

3.6. The dereference chain transformation

This transformation is needed to maintain consistency between the dataflow analysis and the clause compiler. A new unbound variable (of either initialized type or uninitialized memory type) is created as a pointer to a memory location. Binding the variable stores the new value in the location. However, the register(s) that originally contained the unbound variable still have pointers to the location. One level of indirection is needed to access the value.

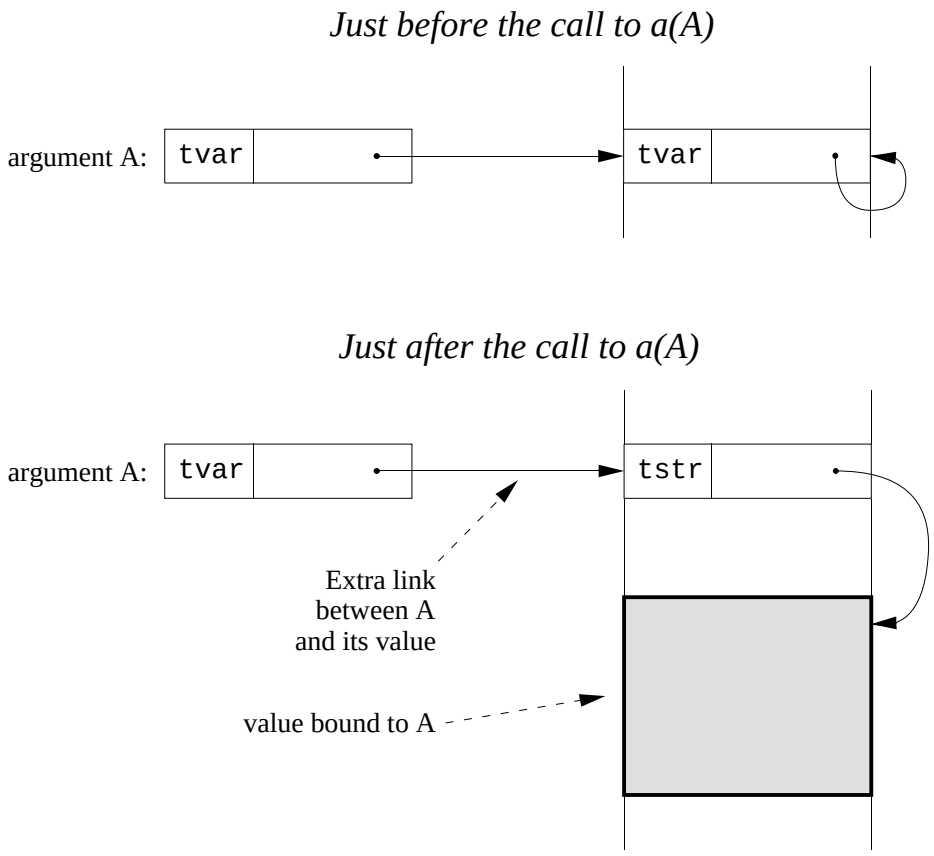

Figure 5.7 – The need for the dereference chain transformation

To see why this is necessary and what it implies, consider the execution of the clause `main` (Figure 5.7):

```
main :- (i) a(A), (ii) write(A).

a(A) :- A=s(t(a),u(b),v(c)).
```

The relevant situation can be seen in the transition from *(i)* (just before the call to `a(A)`) to *(ii)* (just after the call to `a(A)`). At *(i)* a new unbound variable A is created on the heap. At *(ii)* the variable A has been bound to a value. The important point is that A still has a `tvar` tag, and that one indirection is needed to access the `tstr` pointer. The extra link exists because the creation of A and its binding are done in separate steps. This is true for both initialized unbound variables and uninitialized memory variables.

This situation is not a problem unless dataflow analysis determines that A is returned as a dereferenced value. In that case there is a conflict between what the analysis deduces and what the clause compiler thinks is true. There are two ways to solve this problem: either weaken the analysis so that it will not deduce a dereference type in this case, or modify the clause compiler to ensure that the variable is dereferenced by doing an extra indirection whenever the variable is accessed after it is bound. The compiler implements the second solution since dereferencing is a time-consuming operation and it is important to derive as many dereference types as possible. The trade-off between doing an extra indirection for a value that may not be accessed later and doing an extra dereference loop seemed to be a fair one.

The compiler inserts code to do this indirection whenever the variable is accessed after it is bound. In addition to maintaining consistency with the analysis, this speeds up later dereferencing. There is a minor interaction with the register allocator—for correctness, variables that get an extra indirection are not allowed to be pref pairs.

Chapter 6

BAM Transformations

1. Introduction

After compiling the program from kernel Prolog into BAM code, a series of optimizing transformations is performed. The transformations performed are: (1) duplicate code elimination, (2) dead code elimination, (3) jump elimination, (4) label elimination, (5) synonym optimization, (6) peephole optimization, and (7) determinism optimization. This chapter first gives two definitions and then presents the transformations.

2. Definitions

The following two definitions are useful:

> **Definition DB:** A *distant branch* is a branch that always transfers control to an instruction other than the next in the instruction stream.

According to this definition, there are exactly four distant branches in the BAM: fail, return, jump, and switch. All other branches do not satisfy the definition since they can fall through to the next instruction.

> **Definition BB:** A *contiguous block* is any sequence of instructions that terminates with a distant branch.

According to this definition, a contiguous block can start with any instruction and can contain conditional branches with a fall through case. Therefore the code contains a large number of overlapping contiguous blocks. This is useful to get maximum optimization when looking for contiguous blocks that satisfy some property. The individual transformations mentioned in this chapter will usually only look at contiguous blocks satisfying certain constraints, for example, the contiguous blocks that begin with a label.

3. The transformations

Seven transformations (Figure 6.1) are done on the BAM code generated for each predicate by the kernel to BAM compilation stage. A transitive closure is performed on the sequence of seven transformations, i.e. they are applied repeatedly until there are no more changes. Each transformation is carefully

coded to result in code that is better (i.e. faster or shorter) than its input, so the closure operation terminates.

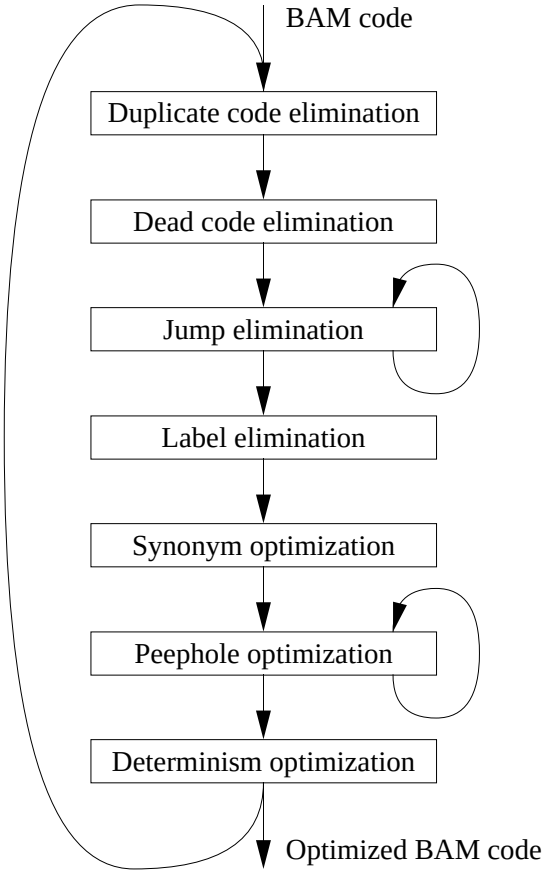

Figure 6.1 – BAM Transformations

3.1. Duplicate code elimination

All duplicate contiguous blocks except the last occurrence are replaced by a jump to the last one. This optimization is also known as *cross-jumping*. It tightens up loose code generated by the type enrichment transformation (Chapter 4). It is implemented by first creating an table indexed by all contiguous blocks that (1) begin with a label, (2) do not contain any other labels (but they are allowed to contain branches), and (3) are not degenerate blocks that consist of only a single jump, return, or fail instruction (but a single switch is allowed). The table contains the label of the last occurrence of the block. All con-

tiguous blocks in the code, including those that do not begin with labels, are looked up in the table and replaced by jumps if they are not the last occurrence. The result of this optimization is to reduce code size at the price of slightly slowing down execution.

3.2. Dead code elimination

All code that is not reachable from the entry point of a predicate is removed. This is done in two steps: First, all the labels that are reachable through any number of branches are calculated by doing a transitive closure. Second, a linear traversal of the code is done and the instructions following a distant branch up to the next reachable label are eliminated.

3.3. Jump elimination

Rearrange contiguous blocks to minimize the number of jump, call, and return instructions. This optimization is a variant of the *jump chaining* optimization. A transitive closure is done on the following replacements:

(1) Replace a jump by the contiguous block it points to if the block is only pointed to by one branch or if the block is shorter than a preset threshold. The threshold can be changed by a compiler directive. The replacement is not done if the block is part of write mode unification or unification with an atom, since these two cases are hurt by the transformation.

(2) Replace a call to a dummy predicate by the code for the predicate if it is straightline code, i.e. its code consists only of non-branches, call instructions, and branches all of whose destinations are `fail`. The predicate's code must be terminated by a `return` or `fail` instruction.

(3) Replace a conditional branch to a conditional branch by a new conditional branch if possible. The only case currently recognized is:

```
test(ne,tvar,V,L).
 ...
label(L).
switch(Tag,V,fail,L2,L3).
```

which causes the test instruction to be replaced by:

```
switch(Tag,V,L1,L2,L3).
label(L1).
```

(4) Replace a branch one of whose destinations is a jump or fail instruction by a new branch identical to the original one except that the destination label is replaced by the destination label of the jump or by fail.

3.4. Label elimination

Remove all labels that are not jumped to by any branch in the code. This is done in two steps: First, the set of all destinations of all branch instructions is collected. Second, the labels not in this set are removed from the code.

3.5. Synonym optimization

This transformation is similar to strength reduction. It does a linear traversal of the code and replaces every addressing mode by the cheapest addressing mode that contains the same value. For example, if `p(1)` and `r(0)` contain the same value, then an occurrence of `p(1)` can be replaced by `r(0)`. The following cost order (from cheapest to most expensive) is used by default and is based on the cost in the VLSI-BAM architecture:

Addressing mode	Reason for cost	Overhead (cycles)
`r(b)`	Promotes creation of `cut(r(b))` which is a no-op	0
`r(I)`	Usable without overhead	0
`Atom`	Requires ldi (load immediate) instruction	1
`Tag^X`	Tagged pointer creation needs lea (load effective address) instruction	1
`p(I)`	Permanent variable needs ld (load) instruction	1
`[r(I)]`	Indirection needs ld (load) instruction	1
`[r(I)+N]`	Offset indirect needs ld (load) instruction	1
`[p(I)]`	Indirect permanent needs 2 ld (load) instructions	2
`[p(I)+N]`	Offset indirect permanent needs 2 ld (load) instructions	2
`r(void)`	Most expensive because it must not be changed	–

The reason given for the cost describes the instructions necessary to implement the addressing mode for the VLSI-BAM. More information on the instruction set of the VLSI-BAM is given in [34]. The addressing mode `r(void)` is created by the register allocator. It corresponds to a void variable, i.e. a variable that occurs only once in a clause and whose value may therefore be ignored. It is made the most

expensive because it must remain unchanged so that peephole optimization can remove the instruction containing it.

The synonym optimization is implemented by maintaining a set of equivalence classes at all points of the program, where each equivalence class is a set of addressing modes whose values are identical. Labels in the code cause the set of equivalence classes to be reset to empty. A future extension of this module could eliminate this restriction by following the labels and performing a transitive closure, resulting in a slight performance gain.

3.6. Peephole optimization

A transitive closure is performed on a peephole transformation with a window of three instructions. The set of patterns was determined empirically by looking at the compiler's output and adding patterns to fix obvious inefficiencies. Each pattern is implemented as a single clause in the optimizer. The patterns are one, two, and three instructions long. However, the window is extended to arbitrary size for one pattern, a generalized last call optimization:

```
call(N/A).
deallocate(I). % Arbitrary number of deallocate instructions.
 ...
deallocate(J).
return.
```

which is transformed to:

```
deallocate(I). % Same sequence as above.
 ...
deallocate(J).
jump(N/A).
```

3.7. Determinism optimization

A choice instruction is removed if it is followed by a sequence of instructions that cannot fail and a cut instruction. This simple-looking optimization significantly increases determinism—many predicates (e.g. Warren's quicksort benchmark) containing a cut become deterministic that would otherwise be compiled with a choice point.

A similar optimization is performed by the simplification transformation of kernel Prolog (Chapter 4). For example, it transforms `(!,p ; q)` into `(!,p)`. The determinism optimization extends simplification—if the goal `s` compiles into instructions that cannot fail then it is able to successfully optimize the BAM code of `(s,!,p ; q)` even when simplification cannot determine that `s` always succeeds.

Consider this predicate, which contains no cut:

```
:- mode((max(A,B,C) :- uninit(C))).  % C is unbound and unaliased.

max(A, B, C) :- A<B, B=C.            % No cut here.
max(A, B, C) :- A=C.
```

It is compiled into the following BAM code (slightly simplified for readability):

```
procedure(max/3).
        deref(r(0),r(0)).
        deref(r(1),r(1)).
        jump(lts,r(0),r(1),l(max/3,1)).  % Conditional branch A<B.
        move(r(0),[r(2)]).               % A<B is false.
        return.
label(l(max/3,1)).
        choice(1/2,[0,2],l(max/3,4)).    % A<B is true.
        move(r(1),[r(2)]).
        return.
label(l(max/3,4)).
        choice(2/2,[0,2],fail).
        move(r(0),[r(2)]).
        return.
```

When `A<B` is true, a choice point is created to try both clauses. If a cut is inserted into the first clause:

```
:- mode((max(A,B,C) :- uninit(C))).  % C is unbound and unaliased.
max(A, B, C) :- A<B, !, B=C.         % Cut is added here.
max(A, B, C) :- A=C.
```

then the code becomes deterministic:

```
procedure(max/3).
        move(b,r(3)).
        deref(r(0),r(0)).
        deref(r(1),r(1)).
        jump(lts,r(0),r(1),l(max/3,4)).  % Conditional branch A<B.
        move(r(0),[r(2)]).
        return.
label(l(max/3,4)).
        cut(r(3)).
        move(r(1),[r(2)]).
        return.
```

Measurements done by Touati [70] justify this optimization. He finds that it makes about half of all choice point operations avoidable.

Chapter 7

Evaluation of the Aquarius system

1. Introduction

This chapter attempts to quantify some of the ideas that were introduced in previous chapters. The evaluation process is as important as any other part of the implementation of a large software system. During the design phase it guides the design decisions. After the design is complete, it shows what features of the design contributed most to its effectiveness and it gives a foundation for starting the next design. Quantitative measurements are the most reliable guideposts one has during the design. For example, it is easy to imagine many possible compiler optimizations, but most of these have an insignificant effect on performance. It is more difficult to discover optimizations that are widely applicable.

Five evaluations are performed in this chapter:

(1) The absolute performance of the system.

(2) The effectiveness of the dataflow analysis.

(3) The effectiveness of the determinism transformation.

(4) A brief comparison with a high performance implementation of the C language.

(5) A bug analysis, summarizing the number and types of bugs encountered during development.

Table 7.1 describes the benchmarks used in this chapter and their size in lines of code (not including comments). The benchmarks were chosen as examples of realistic programs doing computations representative of Prolog. This includes benchmarks that spend much of their time executing built-in predicates because this behavior is common in real-world programs. The benchmarks are divided into two classes, *small* and *large*, depending on whether the compiled code with analysis is smaller or larger than 1000 words. The benchmarks log10, ops8, times10, and divide10 are grouped together and referred to as *deriv* because they are closely related. The benchmarks are available by anonymous ftp to arpa.berkeley.edu.

All VLSI-BAM numbers in this chapter were obtained from the VLSI-BAM instruction-level simulator and include cache effects [17]. The simulated system has 128 KB instruction and data caches. The

Table 7.1 – The benchmarks

Benchmark	Lines	Description
nreverse	10	Naive reverse of a 30-element list.
tak	15	Recursive integer arithmetic.
qsort	19	Quicksort of a 50-element list.
log10	27	Symbolic differentiation.
ops8	27	Symbolic differentiation.
times10	27	Symbolic differentiation.
divide10	27	Symbolic differentiation.
serialise	29	Calculate serial numbers of a list.
queens_8	31	Solve the eight queens puzzle.
mu	33	Prove a theorem of Hofstadter's ''mu-math.''
zebra	36	A logical puzzle based on constraints.
sendmore	43	The SEND+MORE=MONEY puzzle.
fast_mu	54	An optimized version of the mu-math prover.
query	68	Query a static database (with integer arithmetic).
poly_10	86	Symbolically raise a polynomial to the tenth power.
crypt	64	Solve a simple cryptarithmetic puzzle.
meta_qsort	74	A meta-interpreter running qsort.
prover	81	A simple theorem prover.
browse	92	Build and query a database.
unify	125	A compiler code generator for unification.
flatten	158	Source transformation to remove disjunctions.
sdda	273	A dataflow analyzer that represents aliasing.
reducer_nowrite	298	A graph reducer based on combinators.
reducer	301	Same as above but writes its answer.
boyer	377	An extract from a Boyer-Moore theorem prover.
simple_analyzer	443	A dataflow analyzer analyzing qsort.
nand	493	A logic synthesis program based on heuristic search.
chat_parser	1138	Parse a set of English sentences.
chat	4801	Natural language query of a geographical database.

caches are direct mapped and use a write-back policy. They are run in warm start: each benchmark is run twice and the results of the first run are ignored. The cache overhead is greatest for tak compiled without analysis, and for poly_10, simple_analyzer, chat, and boyer. For these programs it ranges from 9% to 24%. For meta_qsort, reducer, and chat_parser the overhead ranges from 2% to 3%. For all other programs the overhead is less than 0.5%.

2. Absolute performance

This section compares the performance of Aquarius Prolog with Quintus Prolog. Tables 7.2 and 7.3 compare the performance of Quintus Prolog version 2.5 running on a Sun 4/65 (25 MHz SPARC) with that of Aquarius Prolog running on the VLSI-BAM (30 MHz). The ''Raw Speedup'' column gives the ratio of the speeds. The ''Normalized Speedup'' column divides this ratio by 1.8. Our group is in the process of

porting the Aquarius system to the MIPS, MC68020, and SPARC processors. It was not possible to get numbers for these systems in time for the final version of this dissertation.

The normalization factor of 1.8 takes into account the Prolog-specific extensions of the VLSI-BAM (a factor of 1.5) and the clock ratio (a factor of 30/25 = 1.2). The general-purpose base architecture of the VLSI-BAM is very similar to the SPARC. The effect of the architectural extensions of the VLSI-BAM [34] has been carefully measured to be about 1.5 for large programs. However, for the small programs the compiler is able to remove many Prolog-specific features, so that the normalized speedup numbers in Table 7.2 are an underestimate.

Table 7.2 – Performance results for small programs (in ms)

Benchmark	Size (lines)	Quintus v2.5 (Sun 4/65)	Aquarius (VLSI-BAM)	Normalized Speedup	Raw Speedup
deriv		1.143	0.0913	7.0	12.5
log10	27	0.153	0.0168		
ops8	27	0.239	0.0189		
times10	27	0.345	0.0257		
divide10	27	0.406	0.0299		
nreverse	10	1.62	0.136	6.6	11.9
qsort	19	4.820	0.173	15.5	27.8
serialise	29	3.10	0.447	3.9	6.9
query	68	23.7	3.57	3.7	6.6
mu	33	7.04	0.808	4.8	8.7
fast_mu	54	9.08	0.932	5.4	9.7
queens_8	31	21.2	1.13	10.4	18.7
tak	15	1120.	25.4	24.5	44.1
poly_10	86	417.	35.5	6.5	11.7
sendmore	43	490.	38.4	7.1	12.8
zebra	36	423.	84.1	2.8	5.0
geometric mean				6.7	12.1
standard deviation of mean				1.9	3.3

For the small benchmarks, the normalized speedup is somewhere between 6.7 and 12.1 (Table 7.2). The normalized speedup of the large benchmarks without built-in predicates is about 5.2 (Table 7.3). Speedup is better for the small benchmarks because dataflow analysis is able to derive better types for many of them. For some of them (such as tak and nreverse) it derives essentially perfect types. The small programs show a large variation in speedups. The tak benchmark does well because it relies on integer arithmetic, which is compiled efficiently using uninitialized register types. The zebra benchmark does poorly for two reasons. First, it does a large amount of backtracking, which is inherently limited by memory bandwidth. Second, it works by successively instantiating arguments of a compound data

Table 7.3 – Performance results for large programs (in ms)					
Benchmark	Size (lines)	Quintus v2.5 (Sun 4/65)	Aquarius (VLSI-BAM)	Normalized Speedup	Raw Speedup
No built-ins					
prover	81	8.67	0.921	5.2	9.4
meta_qsort	74	49.6	4.71	5.8	10.5
nand	493	173.3	13.7	7.0	12.7
reducer_nowrite	298	312.	37.2	4.6	8.4
chat_parser	1138	1157.	129.5	5.0	8.9
browse	92	5450.	741.	4.1	7.4
geometric mean				5.2	9.4
standard deviation of mean				0.5	0.8
Including built-ins					
unify	125	18.3	1.40	7.2	13.0
flatten	158	13.6	1.42	5.3	9.6
sdda	273	29.5	2.94	5.6	10.0
crypt	64	21.7	4.00	3.0	5.4
simple_analyzer	443	180.	33.4	3.0	5.4
reducer	301	405.	44.9	5.0	9.0
chat	4801	3100.	699.	2.5	4.4
boyer	377	4870.	1360.	2.0	3.6
geometric mean				3.8	6.9
standard deviation of mean				0.7	1.3
geometric mean (all large programs)				4.4	7.9

Table 7.4 – Time spent in built-in predicates		
Benchmark	Time (%)	Most used built-ins
prover	0	–
meta_qsort	0	–
chat_parser	0	–
nand	<1	–
browse	1	length/2
reducer	40	write/1, compare/3, arg/3
unify	40	arg/3, functor/3, compare/3
crypt	50	div/2, mod/2, */2
boyer	60	arg/3, functor/3
simple_analyzer	70	compare/3, sort/2, arg/3
sdda	70	write/1, =../2, compare/3
flatten	80	write/1, sort/2, compare/3, name/2, functor/3, arg/3

structure. The analysis algorithm does not have a representation for this operation, so it cannot be optimized.

The built-in predicates in Aquarius Prolog are not greatly faster than those in Quintus Prolog, since many of the Quintus built-ins are not written in Prolog, but in hand-crafted assembly. The Aquarius system shows better speedup over Quintus built-ins written in Prolog (such as `read/1` and `write/1`) and the entry specialization transformation also speeds up the built-ins. Table 7.4 gives the percentage of time that

the benchmarks spend executing inside built-in predicates. This number does not take into account built-ins that are implemented as in-line code (arithmetic test, addition and subtraction, and type checking). The table also gives the most often used built-in predicates for each benchmark in decreasing order of usage.

Several benchmarks use built-in predicates significantly. The normalized speedup for these programs is 3.8, somewhat less than programs without built-ins (Table 7.3). The normalized speedup for all large programs is 4.4 (the reducer benchmark is counted only once in this average). The boyer benchmark does poorly because it relies heavily on the `arg/3` and `functor/3` built-in predicates. The chat benchmark uses these built-ins as well as others including `setof/3`, but it was not possible to measure the fraction of execution time spent in them. The sdda and flatten benchmarks do well partly because the `write/1` built-in is much faster in Aquarius than in Quintus.

3. The effectiveness of the dataflow analysis

This section evaluates the effectiveness of the dataflow analysis with three kinds of measurements. Tables 7.5, 7.6, and 7.7 give the effect of the dataflow analyzer on performance and code size, and the efficiency of the analyzer both in terms of its execution time and the fraction of arguments for which types can be deduced.

For a representative set of realistic Prolog programs of various sizes up to 1,100 lines, the analyzer is able to derive type information for 56% of all predicate arguments. It finds that on average 23% of all predicate arguments are uninitialized, 21% of arguments are ground, 10% of arguments are nonvariables, and 17% of arguments are recursively dereferenced. The sum of these three numbers is greater than 56% since it is possible for an argument to have multiple types, e.g. it can be ground and recursively dereferenced at the same time. Doing analysis reduces execution time on the VLSI-BAM by 18% for programs without built-ins and static code size by 43% for all programs.

Table 7.5 gives the execution time in microseconds of the benchmarks for the VLSI-BAM compiled without analysis (No Modes) and with analysis (Auto Modes). The last three columns give the ratios of the auto modes to the no modes times. To give an idea how built-ins affect the results of analysis, Table 7.5 gives two performance ratios for the large benchmarks: the first for all programs, and the second for

Table 7.5 – The effect of dataflow analysis on performance

Benchmark	No Modes (μs)			Auto Modes (μs)			Auto/No Modes		
	Time	Deref	Trail	Time	Deref	Trail	Time	Deref	Trail
deriv	146	18.2	5.5	91.3	0.3	0.1	0.63	0.02	0.02
log10	25.9	2.3	0.7	16.8	0	0			
ops8	28.5	3.3	1.0	18.9	0.3	0.1			
times10	39.7	5.1	1.3	25.7	0	0			
divide10	51.7	7.5	2.5	29.9	0	0			
nreverse	308	79.7	31.1	136	0	0	0.44	0.00	0.00
qsort	378	109	25.1	173	0	0	0.46	0.00	0.00
serialise	512	75.8	12.3	447	44.9	0.7	0.87	0.59	0.05
mu	992	154	48.0	783	139	34.7	0.79	0.90	0.72
fast_mu	1120	148	38.0	932	64.4	7.9	0.83	0.44	0.21
queens_8	1700	271	67.9	1090	33.4	0	0.64	0.12	0.00
query	5180	560	174	3570	0	0	0.69	0.00	0.00
tak	71700	13800	3180	25400	0	0	0.35	0.00	0.00
poly_10	60400	6280	1740	35600	1080	209	0.59	0.17	0.12
zebra	84600	11400	8.6	84100	11400	8.4	0.99	1.00	0.98
average							0.66	0.29	0.19
prover	1070	110	29.4	820	51.2	5.9	0.76	0.47	0.20
unify	1600	198	33.9	1400	138	19.3	0.88	0.69	0.57
flatten	1460	149	9.9	1420	133	6.5	0.97	0.90	0.66
sdda	3180	368	36.9	2940	296	21.3	0.92	0.81	0.58
crypt	4090	319	104	4000	262	104	0.98	0.82	1.00
meta_qsort	5330	674	182	4450	417	63.0	0.83	0.62	0.35
nand	18700	2290	542	13400	902	22.9	0.72	0.39	0.04
simple_analyzer	35400	3880	316	31900	3080	76.2	0.90	0.79	0.24
reducer	48800	6680	1210	44900	5580	731	0.92	0.84	0.61
chat_parser	151000	19400	6990	131000	11200	4360	0.87	0.58	0.62
browse	820000	117000	28600	741000	96700	20400	0.90	0.82	0.71
boyer	1410000	73900	6340	1360000	75000	6270	0.97	1.02	0.99
average							0.89	0.73	0.55
average (no built-ins)							0.82	0.58	0.39

programs that do not use built-ins significantly (the first five of Table 7.4). Data initialization times are subtracted from deriv, nreverse, qsort, serialise, and prover. The table also gives the time each benchmark spends performing dereferencing (Deref) and trailing (Trail).

The time spent in dereferencing and trailing, two of the most common Prolog-specific operations, is significantly reduced by analysis. For the small benchmarks analysis reduces dereferencing from 17% to 5% of execution time, and trailing from 4% to 0.6% of execution time. This is because they are simple enough that analysis is able to deduce most relevant modes. For the large benchmarks dereferencing is reduced from 11% to 9% and trailing is reduced from 2.3% to 1.3%. These results are less extreme for two reasons: the large benchmarks use built-ins, which are unaffected by analysis, and the analyzer loses infor-

Table 7.6 – The effect of dataflow analysis on static code size

Benchmark	No Modes (instructions)	Auto Modes (instructions)	Auto/No Modes
tak	80	34	0.42
nreverse	287	139	0.48
queens_8	472	146	0.31
qsort	485	215	0.44
deriv	5891	1123	0.19
log10	1464	272	
ops8	1469	277	
times10	1479	287	
divide10	1479	287	
query	1425	403	0.28
serialise	860	520	0.60
mu	1169	731	0.63
fast_mu	1165	718	0.62
zebra	1271	814	0.64
poly_10	3023	893	0.30
average			0.45
crypt	1239	1027	0.83
browse	1863	1150	0.62
prover	4395	1318	0.30
meta_qsort	2484	1424	0.57
flatten	4267	2335	0.55
unify	6326	4210	0.67
sdda	6526	5031	0.77
simple_analyzer	9057	5836	0.64
nand	23406	6654	0.28
reducer	11726	7682	0.66
boyer	24862	9136	0.37
chat_parser	33557	20516	0.61
average			0.57

mation due to its inability to handle aliasing and its limited type domain.

Table 7.6 gives the static code size (in VLSI-BAM instructions) for the benchmarks compiled without analysis (No Modes) and with analysis (Auto Modes). The effect of analysis on code size is greater than the effect on performance. This follows from the compiler's implementation of argument selection: when no modes are given, the compiler generates more code to handle arguments of different types. If analysis derives the type then the code becomes much smaller. The code size compares favorably with other symbolic processors, and is low enough that there is no disadvantage to having a simple instruction set. With the analyzer, code size on the VLSI-BAM is similar to the KCM [6], about three times the PLM, a micro-coded WAM [28], and about one fourth the SPUR using macro-expanded WAM [8].

Table 7.7 – The efficiency of dataflow analysis

Benchmark	Args	Preds	Time (sec)	Modes (fraction of arguments) uninit	ground	nonvar	rderef	any
deriv	12	8	11.9	0.33	0.67	0.00	0.67	1.00
log10	3	2	2.9					
ops8	3	2	3.0					
times10	3	2	3.0					
divide10	3	2	2.9					
tak	4	2	2.3	0.25	0.75	0.00	0.75	1.00
nreverse	5	3	2.2	0.40	0.60	0.00	0.60	1.00
qsort	7	3	3.4	0.43	0.57	0.00	0.57	1.00
query	7	5	4.2	0.86	0.14	0.00	0.14	1.00
zebra	10	6	3.5	0.10	0.00	0.50	0.00	0.60
serialise	16	7	4.2	0.38	0.19	0.06	0.19	0.63
queens_8	16	7	6.0	0.31	0.69	0.00	0.69	1.00
mu	17	8	9.6	0.12	0.47	0.00	0.12	0.65
poly_10	27	11	16	0.33	0.67	0.00	0.67	1.00
fast_mu	35	7	21	0.29	0.55	0.05	0.55	0.89
average				0.35	0.48	0.06	0.45	0.89
meta_qsort	10	7	11	0.30	0.00	0.10	0.00	0.40
crypt	18	9	12	0.00	0.61	0.11	0.56	0.72
prover	22	9	13	0.27	0.09	0.27	0.14	0.68
browse	42	14	20	0.24	0.45	0.05	0.40	0.74
boyer	62	25	31	0.27	0.00	0.06	0.00	0.34
flatten	83	28	34	0.27	0.08	0.16	0.11	0.52
sdda	87	32	45	0.18	0.07	0.17	0.08	0.44
reducer	134	41	50	0.13	0.10	0.05	0.12	0.29
unify	141	29	84	0.18	0.19	0.14	0.21	0.56
nand	180	43	5900	0.26	0.67	0.00	0.28	0.93
simple_analyzer	270	71	77	0.23	0.10	0.08	0.10	0.41
chat_parser	744	156	263	0.44	0.19	0.02	0.09	0.67
average				0.23	0.21	0.10	0.17	0.56

Table 7.7 presents data about the efficiency of the dataflow analyzer. For each benchmark it gives the number of predicate arguments (Args) where a predicate of arity N is counted as N, the number of predicates (Preds), the analysis time (Time), the fraction of arguments that are uninitialized (uninit), ground (ground), nonvariable (nonvar), or recursively dereferenced (rderef), and the fraction of arguments that have any of these types (any). Analysis time is measured under Quintus release 2.0 on a Sun 3/60. It is roughly proportional to the number of arguments in the program, except for the nand benchmark. The sum of the individual modes columns is usually greater than the any modes column. This is because arguments can have multiple modes—they can be both recursively dereferenced and ground or nonvariable. Uninitialized arguments are present in great quantities, even in large programs such as chat_parser and simple_analyzer. Comparing the small and large benchmarks, the fraction of derived modes decreases for

the large programs for each type except nonvariable. For both the small and large benchmarks the analyzer transforms one third of the uninitialized modes into uninitialized register modes.

4. The effectiveness of the determinism transformation

To show what parts of the determinism transformation of Chapter 4 are the most effective, it is useful to define a spectrum of determinism extraction algorithms ranging from pure WAM to the full mechanism of the Aquarius compiler. To do this, the Aquarius mechanism for extracting determinism is divided into three orthogonal axes:

(1) The kind of tests used to extract determinism. These tests are separated into three classes: explicit unifications (e.g. `X=a`, `X=s(Y)`), arithmetic tests (e.g. `X<Y`, `X>1`), and type checks (e.g. `var(X)`, `atomic(X)`). Pure WAM uses only explicit unifications with nonvariables. Aquarius uses all three kinds.

(2) Which argument(s) are used to extract determinism. Pure WAM uses only the first argument of a predicate. Aquarius uses any argument that it can determine is effective. It uses enrichment heuristic 2 (Chapter 4 section 6.2).

(3) Whether the factoring transformation is performed (Chapter 4). Factoring significantly increases determinism for predicates that contain many identical compound terms in the head. Pure WAM does not assume factoring. Aquarius does factoring by default.

These three parameters define a three-dimensional space of determinism extraction algorithms. Each algorithm is characterized by a 3-tuple depending on its position on each of the axes (Table 7.8). This results in $3 \times 2 \times 2 = 12$ data points. Pure WAM selection corresponds to the first element in each column, denoted by the 3-tuple (U, ONE, NF). The Aquarius compiler's selection corresponds to the last element in each column, denoted by the 3-tuple (UAT, ANY, F).

For each of these 12 points three parameters were measured: execution time, static code size, and compile time. All programs are compiled with dataflow analysis and executed on the VLSI-BAM. All averages are geometric means. It was only possible to do measurements for nine benchmarks: nreverse, qsort, query, mu, fast_mu, queens_8, flatten, meta_qsort, and nand. Therefore the variance of the results is large and

Table 7.8 – Three dimensions of determinism extraction		
Kind of test	Which argument	Factoring
Explicit unifications only (*U*).	First argument only (*ONE*).	No factoring (*NF*).
Explicit unifications and arithmetic tests (*UA*).	Any argument (*ANY*).	Do factoring (*F*).
Explicit unifications, arithmetic tests, and type checks (*UAT*).		

they can be relied upon only to indicate trends. The benchmarks were written for the WAM. The measurements compare only the relative powers of different kinds of determinism extraction in the BAM. They do not compare the WAM and BAM directly.

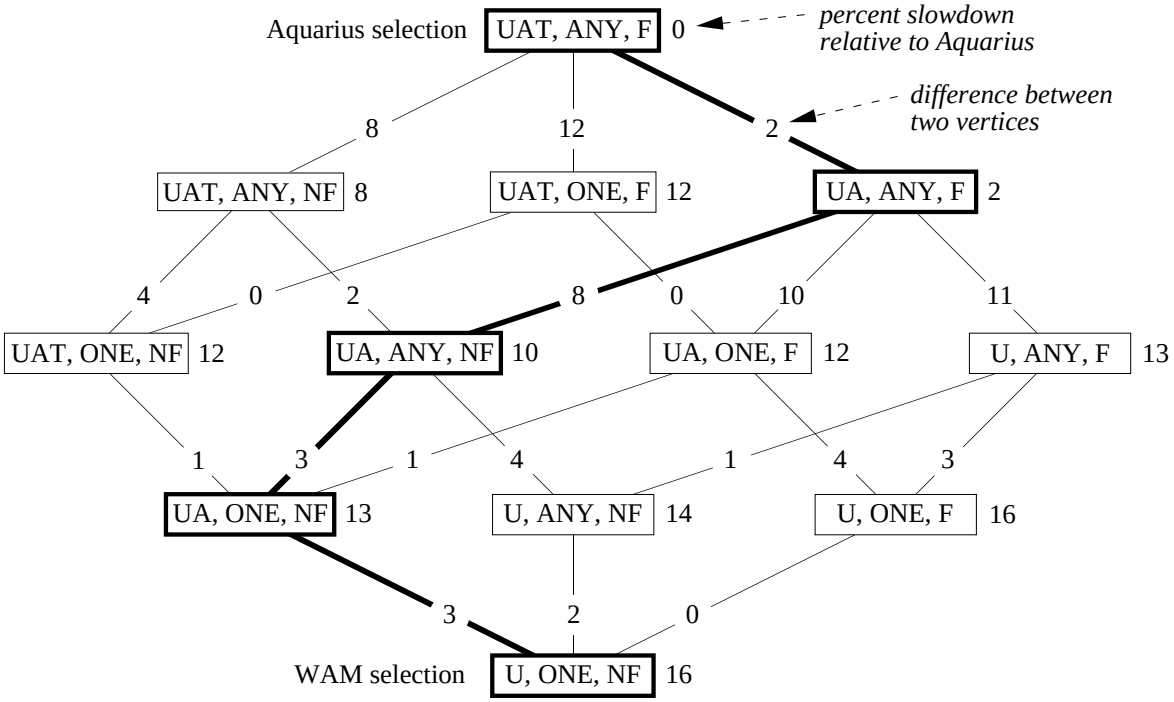

Figure 7.1 – The effectiveness of determinism extraction

Figure 7.1 depicts the 12 points as a lattice. Each vertex denotes one particular combination of determinism extraction. The top element corresponds to Aquarius selection and the bottom element corresponds to WAM selection. Each edge connects two points that differ by one step in one coordinate. The vertices

are marked with the percent slowdown compared to Aquarius selection. The edges are marked with the percent difference in execution time between their two endpoints.

The mean speedup for the nine benchmarks when going from WAM selection (U, ONE, NF) to Aquarius selection (UAT, ANY, F) is 16%. There is no significant change in mean code size for any of the twelve data points. The variance of the compile time is too large to make any conclusions about it.

The mean speedup of factoring is 8%. However, factoring is the only transformation that sometimes slows down execution. The factoring heuristic should be refined to look inside compound arguments to see whether there is any potential determinism there. If there is none, it should not factor that argument.

One way of finding a set of effective extensions for determinism extraction is by traversing the lattice from bottom to top, and picking the edge with the greatest performance increase at each vertex. Starting at WAM selection (U, ONE, NF), the first extension is the ability to use arithmetic tests in selection. This speeds up execution by 3%. The second extension is the ability to select on any argument. This speeds up execution by another 3%. The third extension is the factoring transformation. This speeds up execution by 8%. At this point, the resulting performance is within 2% of Aquarius selection.

The resulting vertex (UA, ANY, F) seems to be a particularly good one, i.e. the ability to select on arithmetic tests in any argument works well together with factoring. Leaving out any one of these three extensions reduces performance by at least 8%. A plausible reason for this result is that the benchmarks do many arithmetic tests on the arguments of compound terms and it is only the combination of the three extensions that is able to compile this deterministically.

5. Prolog and C

The performance of Aquarius Prolog is significantly better than previous Prolog systems. A question one can pose is how the system compares with an implementation of an imperative language. This section presents a comparison of Prolog and the C language on several small programs. The comparison is not exhaustive—there are so many factors involved that I do not attempt to address this issue in its entirety. I intend only to dispel the notion that implementations of Prolog are inherently slow because of its expressive power. A serious comparison of two languages requires answering the following questions:

(1) How can implementations of different languages be compared fairly? This comparison concentrates exclusively on the language and ignores features external to the language itself, such as user interface, development time, and debugging abilities. One method is to pick problems to be solved, and then to write the ‘‘best’’ programs in each language to solve the problems, choosing the algorithms appropriate for each language. The disadvantages of this approach are (a) different languages are appropriate for different problems, (b) how does one decide when one has written the ‘‘best’’ program? To avoid these problems I have chosen to compare algorithms, not programs.

(2) Which algorithms will be implemented in both languages? Ideally one should select a range of algorithms, from those most suited to imperative computations (e.g. array computations) to those most suited to symbolic computation (e.g. large dynamic data objects, pattern matching). Prolog is at an advantage at the symbolic end of the spectrum because to implement symbolic computations in an imperative language we effectively have to implement more and more of a Prolog-like system in that language. The programmer does the work of a compiler. At the imperative end of the spectrum, the efficiency of Prolog depends strongly on the ability of the compiler to simplify its general features.

(3) What programming style will be used in coding the algorithms? I have made an attempt to program in a style which is acceptable for both languages. This includes choosing data types in both languages that are natural for each language. For example, in Prolog dynamic data accessed by pointers is easiest to express, whereas in C static arrays are easiest to express. It is possible to use dynamic data in C, but it requires more effort and is used only for those tasks that need it specifically.

(4) How are architectural features taken into account? For fairness both implementations should run on the same machine. The measurements use the same processor, the MIPS, for both implementations. However, a general-purpose architecture favors the execution of imperative languages, since it has been designed to execute such languages well. This shows up for algorithms whose Prolog implementation makes heavy use of Prolog-specific features. To allow the reader to make an informed judgment, the table does not correct for this fact. It is important to bear in mind that by adding additional architectural features comprising 5% of the chip area to the VLSI-BAM (a pipelined processor similar in many ways to the MIPS), the performance increases by 50% for programs that use

Prolog-specific features (compiled with the current version of the Aquarius compiler). Architectural studies done by our research group suggest that these features could be added to a future MIPS processor.

Table 7.9 compares the execution time of small algorithms coded in both C and Prolog on a 25 MHz MIPS processor. Measurements are given for tak, fib, and hanoi, which are recursion-intensive integer functions; and for quicksort, which sorts a 50 element list 10000 times. Prolog and C source code is available by anonymous ftp to arpa.berkeley.edu. In all cases the user time is measured with the Unix ‘‘time’’ utility. The C versions are compiled with the standard MIPS C compiler using both no optimization and the optimization level that produces the fastest code (usually level 4). The Prolog versions are compiled with dataflow analysis and translated into MIPS assembly by a partial translator. The same algorithms were encoded for both Prolog and C, in a natural style for each. The natural style in C is to use static data, whereas in Prolog all data is allocated dynamically.

Table 7.9 – Comparing Prolog and C (in sec)

Benchmark	Aquarius Prolog	MIPS C	
		Unoptimized	Optimized
tak(24,16,8)	1.2	2.1	1.6
fib(30)	1.5	2.0	1.6
hanoi(20,1,2,3)	1.3	1.6	1.5
quicksort	2.8	3.3	1.4

Recursive functions are fast in Prolog for three reasons: last call optimization converts recursion into iteration, environments (stack frames) are allocated per clause and not per procedure as in C, and outputs are returned in registers (they are of uninitialized register type). Last call optimization allows functions with a single recursive call to execute with constant stack space. This is essential for Prolog because recursion is its only looping construct. The MIPS C compiler does not do last call optimization. C has constructs to denote iteration explicitly (e.g. ‘‘for’’ and ‘‘while’’ loops) so it does not need this optimization as strongly. The time for `fib(30)`, the only recursive integer function that is not able to use last call optimization in Prolog, is closest to C.

The two quicksort implementations are careful to use the same pivot elements. The C implementation uses an array of integers and does in-place sorting. The Prolog implementation uses lists and creates a new sorted list. The list representation needs two words to store each data element. Coincidentally, the

Prolog version is twice as slow as the C version, the same as the ratio of the data sizes.

Table 7.10 – Classification of bug types

Kind	Description	%
Mistake	A part of the compiler that is incorrect due to an oversight. When many mistakes occur related to one particular area, then they become hotspot bugs.	39
• Local	A problem that can be fixed by changing just a few predicates. For example, it may be due to a typographical error or a simple oversight in a predicate definition.	(37)
• Global	A problem that can be fixed only with many changes throughout the compiler. This kind of mistake is more fundamental. For example, avoiding the generation of BAM instructions with double indirections requires many small changes.	(3)
Incomplete	A part of the compiler whose first implementation is incomplete because of incomplete understanding of its purpose. Later use stretches it beyond what it was intended to do, so that it needs to be extended and/or cleaned up. For example, the updating of type formulas when new information is given.	19
Hotspot	A critical area of the compiler that requires much thinking to get correct. Its importance is much greater than its size would indicate. Such an area gets more than its share of mistakes.	16
• Conceptual	A concept in the compiler design whose implementation is prone to many mistakes. For example, the concept of uninitialized variables.	(13)
• Physical	A part of the compiler's text. For example, symbolic unification in the dataflow analyzer and parameter passing in the clause compiler both resulted in many bugs.	(14)
Mixture	An undesired interaction between separate parts of the compiler. Despite careful design, often the separate transformations and optimizations are not completely orthogonal, but interact in some (usually limited) way. For example, maintaining consistency between the dataflow analyzer and the clause compiler. This leads to the dereference chain transformation, which in its turn leads to the problem of interaction between it and the preferred register allocation.	16
Improvement	A possible improvement in the compiler. This is not strictly a bug, but it may point to an important optimization that could be added to the compiler. For example, a possible code optimization or reduction in compilation time.	9
Understanding	A problem due to the programmer misunderstanding the required input to the compiler. This is not strictly a bug, but it may point to difficulties in the compiler's user interface or in the language. For example, the difference between the terms `_is_` and `_<_` in Prolog. The first is a variable and the second is a structure.	4

6. Bug analysis

This section gives an overview of the number and types of bugs encountered during compiler development. A *bug* in a program is a problem that leads to incorrect or undesired behavior of the program. In the compiler, this means incorrect or slow compilation, or slow execution of compiled code.

Table 7.10 classifies the bugs found during development [76]. (The percentages do not add up to 100% because bugs can be of more than one type.)

The development of the compiler started early 1988 and proceeded until late 1990. An extensive suite of test programs was maintained to validate versions of the compiler. The test suite was continually extended with programs that resulted in bugs and with programs from external sources. Records were kept of all bugs reported by users of the compiler other than the developer. A total of 79 bug reports were sent from January 1989 to August 1990 by five users. The frequency of bug reports stayed constant near four per month. Statistical analysis is consistent with the distribution being random with no time dependence, i.e. the number of bug reports fluctuates, but there is no increasing or decreasing trend. Therefore the development introduced bugs at about the same rate as they were reported and fixed. This coincidence can be explained by postulating that the time spent developing was limited by the necessity of having to spend time debugging to maintain a minimum level of robustness in the compiler. This is consistent with my personal experience during the development process.

Chapter 8

Concluding Remarks and Future Work

‘‘So many things are possible just as long
as you don’t know they’re impossible.’’
– Norton Juster, *The Phantom Tollbooth*

1. Introduction

In this chapter I recapitulate the main result of this dissertation, distill some practical lessons learned in the design process, talk about the caveats of language design, and give directions for future research.

2. Main result

My thesis is that logic programming can execute as fast as imperative programming. For this purpose I have implemented a new optimizing Prolog compiler, the Aquarius compiler. The driving force in the compiler is to specialize the general mechanisms of Prolog (i.e. the logical variable, unification, dynamic typing, and backtracking) as much as possible. The main ideas in the compiler are: the development of a new abstract machine that allows more optimization, a mechanism to generate efficient code for deterministic predicates (converting backtracking to conditional branching), specialization of unification (encoding each occurrence of unification in the simplest possible way), and the use of global dataflow analysis to derive types.

The resulting system is significantly faster than previous implementations and is competitive with C on programs for which dataflow analysis is able to do sufficiently well. It is about five times faster than Quintus Prolog, a popular commercial implementation.

3. Practical lessons

During the design of this compiler I have found four principles useful.

(1) **Simplicity is common**. Most of the time, only simple cases of the general mechanisms of the language are used. For example, most uses of unification are memory loads and stores. Many of these simple cases are easily detected at compile-time.

(2) **Use the design time wisely**. There are many possible optimizations that one can implement in a compiler of this sort. To get the best results, rank them according to their estimated performance gain relative to their implementation effort, and only implement the best ones. Do not be distracted by clever ideas unless you can prove that they are effective.

(3) **Keep the design simple**. For each optimization or transformation, implement the simplest version that will do the job. Do not attempt to implement a more general version unless it can be done without any extra effort. It is easy to become entangled in the mechanics of implementing a complex optimization. Often a simple version of this optimization achieves most of the benefits in a fraction of the time.

(4) **Document everything, including bugs**. Documentation is an extension to one's memory and it pays for itself quickly. The mental effort spent in writing down what one has done results in a better recollection of what happened. In this design, I have maintained two logs. The first is a file in chronological order that documents each change and the reason for it. The second is a directory containing bug reports contributed by the users of the compiler and brief discussions of the fixes.

The first three of these principles are corollaries of what is sometimes called the ''80-20 rule'': 80% of the results are obtained with 20% of the effort. Using this principle consistently was very important for my work and for the BAM project as a whole.

4. Language design

The Prolog language is only an approximation to the ideal of logic programming. During this research, our group has grappled with some of the deficiencies of Prolog. There are deficiencies in the area of logic: Prolog's approximation to negation (i.e. negation-as-failure) is unsound (i.e. it gives incorrect results) when used in the wrong way. Prolog's implementation of unification can go into infinite loops when creating circular terms. The default control flow is too rigid for data-driven programming.

There are deficiencies in the area of programming: The correspondence between a program and its execution efficiency is not always obvious. Unification is only able to access the surface of a complex data structure. Because the clauses of a predicate are written separately, many conditions have to be repeated or

extra predicates have to be defined. There is a sense in which Prolog is a kind of assembly language.

All of the above problems have solutions, some of which have been implemented in existing systems and in the Aquarius system. However, for three reasons I have resisted the impulse to change the language more than just a little. First, of all logic languages, the Prolog language has the largest and most vigorous user community, and this is a resource I wanted to tap. There are many programs written in Prolog, in various styles, and I wanted to see if this existing pool of ingenuity could be made to run faster. Second, it is unwise to change more than one component of a system at the same time, especially if they can interact in unpredictable ways. That is, one should not design a new language and a compiler for it at the same time. Third, I do not deem myself competent yet to design a language. I believe in the rule of bootstrapped competence: Before writing a compiler, write programs. Before designing a language, write compilers. Competence in each task is limited by competence in its prerequisite.

The best languages are those which distill great power in a small set of features. This makes such languages useful as tools for thought as well as for implementation. Practical aspects such as how efficient it can be implemented are as important in a good language design as theoretical aspects. A good language is theoretically clean (i.e. easily understood) as well as being efficiently implementable. Examples of such languages are Pascal (many algorithms are specified in an Pascal-like pseudo-code), Scheme, and Prolog. To create such a language, a person must have completely digested a set of ideas as well as have a large amount of practical experience. This is a difficult combination—it is easy to gloss over the areas one does not know well.

5. Future work

The goal of achieving parity with imperative languages has been achieved for the class of programs for which dataflow analysis is able to provide sufficient information, and for which the determinism is accessible through built-in predicates. To further improve performance these limits must be addressed.

To guide the removal of these limits it is important to build large applications and study the interaction between programming style and the implementation. This is a problem of successive refinement. A more sophisticated implementation catalyzes a new style of programming, which in its turn catalyzes a new

implementation, and so forth. The first step in this process was the development of the first Prolog compiler and the WAM. The Aquarius system is only the second step. It is able to generate efficient code from programs written in a more logical style than standard Prolog. However, the limits of this style are not yet understood as they are in the WAM. Further work in this area will lead to a successor to Prolog that is closer to logic and also efficiently implementable.

5.1. Dataflow analysis

When writing a program, a programmer commonly has a definite intention about the data's type (intending predicates to be called only in certain ways) and about the data's lifetime (intending data to be used only for a limited period). Because of this consistency, I postulate that a dataflow analyzer should be able to derive this information and a compiler should be able to take advantage of it.

There has been much good theoretical work on global analysis for Prolog, but few implementations, and fewer still that are part of a compiler that takes advantage of the information. Measurements of the Aquarius system show that a simple dataflow analysis scheme integrated into a compiler is already quite useful. However, the implementation has been restricted in several ways to make it practical. As programs become larger, these restrictions limit the quality of the results. I hope the success of this experiment encourages others to relax these restrictions. For example, it would not be too difficult to:

- Extend the domain to represent common types such as integers, proper lists, and nested compound terms. This is especially important for general-purpose processors.
- Extend the domain to represent variable aliasing explicitly. This avoids the loss of information that affects the analyzer.
- Extend the domain to represent data lifetimes. This is useful to replace copying of compound terms by in-place destructive assignment. In this way dynamically allocated data becomes static. The term ‘‘compile-time garbage collection’’ that has been used to describe this process is a misnomer; what is desired is not just memory recovery, but to preserve as much as possible of the old value of the compound term. Often a new compound term similar to the old one is created at the same time the old one becomes inaccessible. Destructive assignment is used to modify only those parts that are

changed.

- Extend the domain to represent types for each invocation of a predicate. For example, the analyzer could keep track not only of argument types for predicate definitions, but of argument types for goals inside the definitions. This is useful to implement multiple specialization, i.e. to make separate copies of a predicate called in several places with different types. For the chat_parser benchmark, making a separate copy of the most-used predicate for each invocation results in a performance improvement of 14%.

5.2. Determinism

The second area in which significant improvement is possible is determinism extraction. The Aquarius compiler only recognizes determinism in built-in predicates of three kinds (unification, arithmetic tests, and type checking). Often this is not enough. In many programs, user-defined predicates are used to choose a clause.

References

1. H. Ait-Kaci, The WAM: A (Real) Tutorial, *DEC PRL Report Number 5*, January 1990.

2. *ALS Prolog, Version 1.0*, Applied Logic Systems, Inc, 1988.

3. M. Auslander and M. Hopkins, An Overview of the PL.8 Compiler, *SIGPLAN Notices '82 Symposium on Compiler Construction Vol. 17*, 6 (1982).

4. *BIM_Prolog Version 2.5* , BIM, Everberg Belgium, Feb. 1990.

5. J. Beer, The Occur-Check Problem Revisited, *Journal of Logic Programming Vol. 5*, 3 (Sept. 1988), pp. 243-261, North-Holland.

6. H. Benker, J. M. Beacco, S. Bescos, M. Dorochevsky, T. Jeffre, A. Pohimann, J. Noye, B. Poterie, A. Sexton, J. C. Syre, O. Thibault and G. Watzlawik, KCM: A Knowledge Crunching Machine, *16th International Symposium on Computer Architecture*, May 1989, pp. 186-194.

7. W. Bledsoe and R. Hodges, A Survey of Automated Deduction, *Exploring Artificial Intelligence: Survey Talks from the National Conferences on Artificial Intelligence*, 1988, pp. 483-543.

8. G. Borriello, A. Cherenson, P. Danzig and M. Nelson, RISCs vs. CISCs for Prolog: A Case Study, *2nd International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS II)*, October 1987, pp. 136-145.

9. K. A. Bowen, K. A. Buettner, I. Cicekli and A. K. Turk, The Design and Implementation of a High-Speed Incremental Portable Prolog Compiler, *3rd International Conference on Logic Programming*, July 1986, pp. 650-656.

10. M. Bruynooghe and G. Janssens, An Instance of Abstract Interpretation Integrating Type and Mode Inferencing (Extended Abstract), *5th International Conference and Symposium*, 1988, pp. 669-683.

11. W. R. Bush, G. Cheng, P. C. McGeer and A. M. Despain, An Advanced Silicon Compiler in Prolog, *1987 IEEE International Conference on Computer Design: VLSI in Computers and Processors*, October 1987, pp. 27-31.

12. R. Carlson, The Bottom-Up Design of a Prolog Architecture, Report No. UCB/CSD 89/536, Master's Report, UC Berkeley, June 1989.

13. M. Carlsson, Freeze, Indexing, and Other Implementation Issues in the WAM, *4th International Conference on Logic Programming Vol. 1* (May 1987), pp. 40-58, MIT Press.

14. M. Carlsson, *Private Communication*, Logic Programming '88, August 1988.

15. M. Carlsson, On the Efficiency of Optimising Shallow Backtracking in Compiled Prolog, *6th International Conference on Logic Programming*, June 1989, pp. 3-16.

16. M. Carlton, B. Holmer and P. Van Roy, The Implementation of a Graph Reducer in VAX 8600 Microcode, CS255 Final Project, Prof. Y. Patt, UC Berkeley, December 1986.

17. M. Carlton, J. Pendleton, B. Sano, B. K. Holmer and A. M. Despain, Cache & Multiprocessor Support in the BAM Microprocessor, *4th Annual Parallel Processing Conference*, April 1990.

18. J. Chang and A. M. Despain, Semi-Intelligent Backtracking of Prolog Based on A Static Data Dependency Analysis, *2nd Symposium on Logic Programming*, July 1985.

19. J. Chang, High Performance Execution of Prolog Programs Based on A Static Data Dependency Analysis, Report UCB/CSD No. 86/263, Ph. D. Thesis, UC Berkeley, October 1985.

20. D. Chen and H. Nguyen, Prolog on SPUR: Upper Bound To Performance Of Macro-Expansion Method, CS252 Final Project, UC Berkeley, May 1987.

21. W. V. Citrin, Parallel Unification Scheduling in Prolog, Report UCB/CSD No. 88/415, Ph. D. Thesis, UC Berkeley, April 1988.

22. J. Cohen and T. J. Hickey, Parsing and Compiling Using Prolog, *Transactions on Programming Languages and Systems Vol. 9* (April 1987), pp. 125-163.

23. P. Cousot and R. Cousot, Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints, *4th ACM Symposium on Principles of Programming Languages*, January 1977, pp. 238-252.

24. S. K. Debray and D. S. Warren, Automatic Mode Inference for Prolog Programs, *3rd Symposium on Logic Programming*, September 1986, pp. 78-87.

25. S. K. Debray, Static Analysis of Parallel Logic Programs, *5th International Conference on Logic Programming*, August 1988, pp. 711-732.

26. T. P. Dobry, Y. N. Patt and A. M. Despain, Design Decisions Influencing the Microarchitecture for a Prolog Machine, *Micro 17*, October 1984.

27. T. P. Dobry, A. M. Despain and Y. N. Patt, Performance Studies of a Prolog Machine Architecture, *12th International Symposium on Computer Architecture*, June 1985.

28. T. P. Dobry, *A High Performance Architecture for Prolog*, Kluwer Academic Publishers, 1990.

29. *SEPIA (Standard ECRC Prolog Integrating Advanced Features) Version 3.0*, ECRC (European Computer-Industry Research Centre), Munich Germany, 1990.

30. R. Haygood, A Prolog Benchmark Suite for Aquarius, Report No. UCB/CSD 89/509, UC Berkeley, April 1989.

31. R. Haygood, *Aquarius Prolog User Manual and Aquarius Prolog Implementation Manual*, UC Berkeley, Spring 1991 (to appear).

32. B. K. Holmer, The Design of Instruction Set Architectures for High Performance Prolog Execution, *Thesis Proposal*, October 4, 1988.

33. B. K. Holmer, *Measurements of General Unification*, Computer Science Division, UC Berkeley, March 1989.

34. B. K. Holmer, B. Sano, M. Carlton, P. Van Roy, R. Haygood, J. M. Pendleton, T. Dobry, W. R. Bush and A. M. Despain, Fast Prolog with an Extended General Purpose Architecture, *17th International Symposium on Computer Architecture*, May 1990, pp. 282-291.

35. B. K. Holmer, Automatic Design of Prolog Instruction Sets, *Ph.D. Thesis (in preparation)*, Expected May 1991.

36. D. Jacobs and A. Langen, Accurate and Efficient Approximation of Variable Aliasing in Logic Programs, *North American Conference on Logic Programming '89*, October 1989, pp. 154-165.

37. G. Kildall, A Unified Approach to Global Program Optimization, *ACM Symposium on Principles of Programming Languages*, January 1973, pp. 194-206.

38. F. Kluzniak, The ‘‘Marseille Interpreter’’—a personal perspective, *Implementations of Prolog*, 1984, pp. 65-70.

39. H. Komatsu, N. Tamura, Y. Asakawa and T. Kurokawa, An Optimizing Prolog Compiler, *Logic Programming '86*, June 1986, pp. 104-115.

40. R. Kowalski, *Logic for Problem Solving*, Elsevier North-Holland, 1979.

41. P. Kursawe, How to Invent a Prolog Machine, *3rd International Conference on Logic Programming*, July 1986, pp. 134-148.

42. J. W. Lloyd, *Foundations of Logic Programming*, Springer-Verlag, 1987.

43. D. Maier and D. S. Warren, *Computing with Logic – Logic Programming with Prolog*, Benjamin/Cummings, 1988.

44. A. Marien, *An Optimal Intermediate Code for Structure Creation in a WAM-based Prolog Implementation*, Katholieke Universiteit Leuven, May 1988.

45. A. Marien and B. Demoen, On the Management of Choicepoint and Environment Frames in the WAM, *North American Conference on Logic Programming*, October 1989, pp. 1030-1047.

46. A. Marien, G. Janssens, A. Mulkers and M. Bruynooghe, The Impact of Abstract Interpretation: an Experiment in Code Generation, *6th International Conference on Logic Programming*, June 1989, pp. 33-47.

47. K. Marriott and H. Sondergaard, Bottom-Up Abstract Interpretation of Logic Programs, *5th International Conference on Logic Programming*, August 1988, pp. 733-748.

48. M. Meier, Compilation of Compound Terms in Prolog, *North American Conference on Logic Programming*, October 1990, pp. 63-79.

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

50. C. S. Mellish, Some Global Optimizations for a Prolog Compiler, *Journal of Logic Programming Vol. 1* (1985), pp. 43-66, North-Holland.

51. H. Mulder and E. Tick, A Performance Comparison Between the PLM and an MC68020 Prolog Processor, *4th International Conference on Logic Programming Vol. 1* (May 1987), pp. 59-73, MIT Press.

52. H. Nakashima and K. Nakajima, Hardware Architecture of the Sequential Inference Machine: PSI-II, *Symposium on Logic Programming*, August 1987, pp. 104-113.

53. R. A. O´Keefe, Finite Fixed-Point Problems, *4th International Conference on Logic Programming Vol. 2* (May 1987), pp. 729-743, MIT Press.

54. Y. N. Patt and C. Chen, A Comparison Between the PLM and the MC68020 as Prolog Processors, Report UCB/CSD No. 87/397, UC Berkeley, January 1988.

55. F. C. N. Pereira and D. H. D. Warren, Definite Clause Grammars for Language Analysis—A Survey of the Formalism and a Comparison with Augmented Transition Networks, *International Journal of Artificial Intelligence Vol. 13*, 3 (May 1980), pp. 231-278, North-Holland.

56. F. C. N. Pereira and S. M. Shieber, *Prolog and Natural–Language Analysis*, Center for the Study of Language and Information (CSLI), Lecture Notes Number 10, 1987.

57. D. A. Plaisted, A Simplified Problem Reduction Format, *Artificial Intelligence Vol. 18* (1982), pp. 227-261.

58. *Quintus Prolog Version 2.5*, Quintus Computer Systems, Inc, January 1990.

59. P. B. Reintjes, A VLSI Design Environment in Prolog, *5th International Conference on Logic Programming*, August 1988, pp. 70-81.

60. V. P. Srini *et al*, VLSI Implementation of a Prolog Processor, *Stanford VLSI Conference*, March 1987.

61. V. P. Srini *et al*, Design and Implementation of a CMOS Chip for Prolog, Report UCB/CSD No. 88/412, UC Berkeley, March 1988.

62. L. Sterling and E. Shapiro, *The Art of Prolog*, MIT Press, 1986.

63. *Sicstus Prolog version 0.5*, Swedish Institute of Computer Science (SICS), August 1987.

64. K. Taki, *Parallel Logic Programming and Execution on the Multi-PSI—A Progress Report of Parallel Inference Systems and Parallel Processing (presentation)*, ICOT, March 1990.

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

66. A. Taylor, Removal of Dereferencing and Trailing in Prolog Compilation, *6th International Conference on Logic Programming*, June 1989, pp. 48-60.

67. A. Taylor, LIPS on a MIPS: Results from a Prolog Compiler for a RISC, *7th International Conference on Logic Programming*, June 1990.

68. E. Tick and D. H. D. Warren, Towards a Pipelined Prolog Processor, *Symposium on Logic Programming*, February 1984, pp. 29-40.

69. E. Tick, Studies in Prolog Architectures, Technical Report No. CSL-TR-87-329, Computer Systems Laboratory, Stanford University, June 1987.

70. H. Touati and A. Despain, An Empirical Study of the Warren Abstract Machine, *Symposium on Logic Programming*, August 1987, pp. 114-124.

71. H. Touati, *A Report on FGCS'88*, UC Berkeley, January 1989.

72. A. K. Turk, Compiler Optimizations for the WAM, *3rd International Conference on Logic Programming*, July 1986, pp. 657-662.

73. P. Van Roy, A Prolog Compiler for the PLM, Report UCB/CSD No. 84/203, Master's Report, UC Berkeley, November 1984.

74. P. Van Roy, B. Demoen and Y. D. Willems, Improving the Execution Speed of Compiled Prolog with Modes, Clause Selection, and Determinism, *TAPSOFT '87 – Springer-Verlag Lecture Notes on Computer Science Vol. 250*, March 1987, pp. 111-125.

75. P. Van Roy, An Intermediate Language to Support Prolog's Unification, *North American Conference on Logic Programming '89*, October 1989, pp. 1148-1164.

76. P. Van Roy, Can Logic Programming Execute as Fast as Imperative Programming?, *Ph.D. Thesis*, Expected December 1990.

77. P. Van Roy, *Detailed Chronological Log of Bugs, Fixes, and Improvements in the Aquarius Compiler*, UC Berkeley, 1990.

78. P. Van Roy and A. M. Despain, The Benefits of Global Dataflow Analysis for an Optimizing Prolog Compiler, *North American Conference on Logic Programming '90*, October 1990.

79. P. Voda, *Trilogy version 1.0*, Complete Logic Systems, Inc, September 1987.

80. D. H. D. Warren, *Applied Logic – Its Use and Implementation as a Programming Tool*, Ph.D. Thesis, University of Edinburgh, also SRI Technical Report 290, 1977.

81. D. H. D. Warren and F. C. N. Pereira, An Efficient Easily Adaptable System for Interpreting Natural Language Queries, *American Journal of Computational Linguistics Vol. 8*, 3-4 (July-December 1982), pp. 110-122.

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

83. D. S. Warren, S. Dietrich and F. Pereira, *The SB-Prolog System, Version 2.2*, SUNY at Stony Brook (presently contact S. K. Debray, University of Arizona, Dept. of Computer Science), March 1987.

84. R. Warren, M. Hermenegildo and S. K. Debray, On the Practicality of Global Flow Analysis of Logic Programs, *5th International Conference and Symposium on Logic Programming*, August 1988, pp. 684-699.

85. *Xenologic Reasoning System Manual*, Xenologic, Inc., 1988.