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 $[\, G_1, G_2, \ldots, G_r \,]$. 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 , ... , Aiai) */
        /* 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 , ... , Aiai , 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.