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 = [ C1 , ... , Cn ] (list of clauses) and Ci = (Hi :- Bi ) */
                  Add the argument X to all Hi in P ;
                  Replace each occurrence of ''!'' in P by '$cut'(X);
                  P' := P ;
                  Add the predicate P' to the program;
                  H := (new head with same functor and arity as all Hi );
                  H' := (H with the additional argument X );
                  P := [ (H :- '$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 $\in$ {`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` $\in$ {`?`, `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(Tag^Imm, r(h), 1)
push(Tag^(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(Tag^Imm, r(h), 1)`
	`push(Tag^(r(h)-Disp1), r(h), 1)`
`equal(Arg, Var, F)`	
`equal(Arg, Tag^Imm, F)`	`move(Var1, Var2)`
	`move(Tag^Imm, Var)`
`unify(Arg, Var, F)`	`move(Tag^(r(h)-Disp), Var)`
	`move(Tag^r(h), Var)`
	`move(Var1, [Var2])`
	`move(Tag^Imm, [Var])`
	`move(Tag^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(tatm^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),tatm^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.