

INFORMATION TO USERS

This manuscript has been reproduced from the microfilm master. UMI films the text directly from the original or copy submitted. Thus, some thesis and dissertation copies are in typewriter face, while others may be from any type of computer printer.

The quality of this reproduction is dependent upon the quality of the copy submitted. Broken or indistinct print, colored or poor quality illustrations and photographs, print bleedthrough, substandard margins, and improper alignment can adversely affect reproduction.

In the unlikely event that the author did not send UMI a complete manuscript and there are missing pages, these will be noted. Also, if unauthorized copyright material had to be removed, a note will indicate the deletion.

Oversize materials (e.g., maps, drawings, charts) are reproduced by sectioning the original, beginning at the upper left-hand corner and continuing from left to right in equal sections with small overlaps.

Photographs included in the original manuscript have been reproduced xerographically in this copy. Higher quality 6" x 9" black and white photographic prints are available for any photographs or illustrations appearing in this copy for an additional charge. Contact UMI directly to order.

**Bell & Howell Information and Learning
300 North Zeeb Road, Ann Arbor, MI 48106-1346 USA
800-521-0600**

UMI[®]

Compiling Prolog to Logic-Inference Virtual Machine

by

Yifei (Fred), Wang ©

**A thesis submitted to
the Faculty of Research and Graduate Studies
in partial fulfillment of the requirements for the degree of**

**Master of Science
in Mathematical Sciences**

**Department of Computer Science
School of Mathematical Sciences
Lakehead University
Thunder Bay, Ontario**

September 10, 1998



**National Library
of Canada**

**Bibliothèque nationale
du Canada**

**Acquisitions and
Bibliographic Services**

**Acquisitions et
services bibliographiques**

**395 Wellington Street
Ottawa ON K1A 0N4
Canada**

**395, rue Wellington
Ottawa ON K1A 0N4
Canada**

Your file Votre référence

Our file Notre référence

The author has granted a non-exclusive licence allowing the National Library of Canada to reproduce, loan, distribute or sell copies of this thesis in microform, paper or electronic formats.

L'auteur a accordé une licence non exclusive permettant à la Bibliothèque nationale du Canada de reproduire, prêter, distribuer ou vendre des copies de cette thèse sous la forme de microfiche/film, de reproduction sur papier ou sur format électronique.

The author retains ownership of the copyright in this thesis. Neither the thesis nor substantial extracts from it may be printed or otherwise reproduced without the author's permission.

L'auteur conserve la propriété du droit d'auteur qui protège cette thèse. Ni la thèse ni des extraits substantiels de celle-ci ne doivent être imprimés ou autrement reproduits sans son autorisation.

0-612-52084-6

Canada

ABSTRACT

The Logic-inference Virtual Machine (LVM) is a new Prolog execution model consisting of a set of high-level instructions and memory architecture for handling control and unification. Different from the well-known Warren's Abstract Machine [1], which uses Structure Copying method, the LVM adopts a hybrid of Program Sharing [2] and Structure Copying to represent first-order terms. In addition, the LVM employs a single stack paradigm for dynamic memory allocation and embeds a very efficient garbage collection algorithm to reclaim the useless memory cells. In order to construct a complete Prolog system based on the LVM, a corresponding compiler must be written.

In this thesis, a design of such LVM compiler is presented and all important components of the compiler are described. The LVM compiler is developed to translate Prolog programs into LVM bytecode instructions, so that a Prolog program is compiled once and can run anywhere.

The first version of LVM compiler (about 8000 lines of C code) has been developed. The compilation time is approximately proportional to the size of source codes. About 80 percent of the time are spent on the global analysis. Some compiled programs have been tested under a LVM emulator. Benchmarks show that the LVM system is very promising in memory utilization and performance.

ACKNOWLEDGMENTS

I would like to express my thanks and appreciation to my supervisor, Professor Xining Li, for his guidance, advice and help.

I would like to thank to my wife (Huai-Rong, Hu) and my mother (Xue-Mei, Sun) for their understanding, encouragement and support.

Also I am grateful to the Natural Science and Engineering Council of Canada for the financial support.

Finally, I would like to thank my external examiner Dr. A. Nayak (Carleton University) and internal examiner Professor L.D. Black (Lakehead University) for their comprehensive comments.

List of Figures and Tables

Figure 1.1	Structure of LVM compiler	2
Figure 2.1	A tagged cell	5
Figure 2.2	Term representation in Structure Copying	5
Figure 2.3	Term representation in Structure Sharing	6
Figure 2.4	WAM data areas	7
Figure 2.5	Environment frame in WAM	7
Figure 2.6	Choice point frame in WAM	8
Figure 2.7	Term representation in Program Sharing	9
Figure 2.8	LVM memory architecture	10
Figure 2.9a	V-frame format in LVM	11
Figure 2.9b	C-frame format in LVM	12
Figure 2.9c	B-frame format in LVM	12
Figure 3.1	Interface to the lexical analyzer	19
Figure 3.2	Role of parser in the compiler	20
Figure 4.1	Order of stub and variable allocation on stack	22
Figure 5.1	Simulation of expression evaluation	48
Figure 8.1	Memory layout in case of BB below AF	76
Figure 8.2	Memory layout for BB above AF	76
Figure 9.1	Execution model of LVM bytecode	81
Table 3.1	Specifiers of operators	17
Table 3.2	Predefined operator table	17
Table 4.1	Code segment labeling rules	21
Table 4.2	Stack cell assignment and offsets of objects	29
Table 4.3	LVM unification instructions	32
Table 4.4	Memory layout of objects for p(+,-,+)/3 clause	33
Table 5.1	Integer arithmetic instructions	46
Table 5.2	Data load/store instructions	47
Table 5.3	Branching instructions	49
Table 5.4	Special comparison instructions	50
Table 5.5	Variable and stub initialization instructions	51
Table 5.6	Object allocation for p(+,-)/2 clause	51
Table 5.7	Opcode matrix of list initialization instructions	52
Table 5.8	Memory allocation for p(-,-)/2	54
Table 9.1	Execution time (sec)	82

Contents

Acceptance Sheet	i
ABSTRACT	ii
ACKNOWLEDGMENTS	iii
List of Figures and Tables	iv
Chapter 1. Introduction	1
1.1 Logic Programming and Prolog	1
1.2 Motivation	1
1.3 Prolog Compiler Design	2
1.4 Outline	4
Chapter 2. Overview of Prolog Implementation	5
2.1 Structure Term Representation	5
2.2 Warren Abstract Machine (WAM)	6
2.3 Logic-Inference Virtual Machine (LVM)	8
Chapter 3. Lexical and Syntax Analysis	13
3.1 Syntax of Prolog Text	13
3.2 Token Generator	19
3.3 Syntax Analysis	19
Chapter 4. Clause Analysis and Translation	21
4.1 Structure Argument Representation and Flattening	21
4.2 Variable Classification and Index Calculation	24
4.3 Register Allocations and Conflict in Parameter Passing	29
4.4 Unification Code of a Clause	31
4.5 Control Code of a Clause	34
4.5.1 Fact	35
4.5.2 Chain Call	37
4.5.3 Rule	38
4.5.4 Query	40

Chapter 5. Built-in Predicates, Arithmetic Expressions and Initializations	42
5.1 Built-in Predicates	42
5.2 Cut	43
5.3 Arithmetic Expressions	45
5.3.1 Arithmetic Operations	46
5.3.2 Register Allocation	47
5.3.3 Arithmetic Comparison	49
5.4 Initializations	50
Chapter 6. Predicate Determinacy Analysis and Indexing	55
6.1 Last Argument Dispatching	55
6.2 Guarded Dispatching	58
6.3 Nondeterministic Predicates	62
Chapter 6. Compilation Optimization	65
7.1 Variable, Stub and SCI Object Initialization Delay	65
7.2 Unnecessary Unification Instruction Elimination	66
7.3 Stub and Variable Initialization Instruction Compression	67
7.4 Combination of Frame Allocation and Unification Instructions	68
7.5 Special Optimization	69
Chapter 8. Assistance to Garbage Collection	73
8.1 Survey of Garbage Collection	73
8.2 Chronological Garbage Collector in the LVM	75
8.3 Garbage Level Estimation	77
8.4 GC Root Set Collection and GC Point Setting	78
Chapter 9. Conclusions	81
9.1 LVM Performance	81
9.2 Compiler Features	83
9.3 Future Work	84
Appendix A. LVM Instruction Set	85
Appendix B. LVM Built-in Instructions	88
Bibliography	91

Chapter 1. Introduction

1.1 Logic Programming and Prolog

The motivation of logic programming is to separate the problem solving process into two parts: (1) a logical specification and (2) execution description [5]. The programmers only focus on the logic specification of the problem. The logic systems automatically provide the execution control. Therefore, logic programming is a higher level of abstraction than the imperative programming in languages like Ada, C or Pascal.

A logic program is a set of logic sentences (clauses) written in Horn clause logic. A group of clauses with the same name and arity (number of arguments), called a predicate is used to define a relation, like a procedure in the imperative languages. Hence, a logic program can be defined as a set of predicates.

Logic programming has four main features as follows:

- (a) Logic variable can be instantiated only once.
- (b) Logic variable can hold value of any type, which is so called dynamic typing.
- (c) Unification is a pattern matching operation for binding variables, building and accessing compound term.
- (d) Backtracking is a searching operation for finding out all satisfying clauses in a predicate.

Prolog is an approximate logic programming language with two constraints on its implementation model for the balance between implementation efficiency and logical completeness. They are:

- (a) The clause listed lexically ahead in a predicate is tried first.
- (b) The goals in each clause are invoked from left to right.

Prolog has been applied in a variety of fields like expert systems, natural language understanding, theorem proving, deductive databases, CAD tool design, compiler writing, and applications of artificial intelligence with coded knowledge.

1.2 Motivation

Many Prolog systems are often an order of magnitude slower than imperative language systems like C. To optimize system performance, a heavy global static analysis based on abstract interpretation has been performed in some high performance Prolog compilers, such as Aquarius [3] and Parma [4]. But all these Prolog systems are built on the principle of Warren Abstract Machine (WAM) execution model [1].

To achieve the same goal, we approach the problem in a different way. A new Prolog system based on a new abstract machine, called Logic-Inference Virtual Machine

(LVM), has been developed. In the next chapter, the important features of WAM and LVM will be briefly described and compared. The LVM adopts a hybrid of Program Sharing and Structure Copying to represent first-order terms. Also, the LVM employs a single stack paradigm for dynamic memory allocation and embeds a very efficient garbage collection algorithm to reclaim the useless memory cells. In order to construct a complete Prolog system based on the LVM, a corresponding compiler has to be written.

1.3 Prolog Compiler Design

The main objective of this thesis is to write a compiler for the LVM system. The LVM compiler (LVMC) is to translate Prolog source code into LVM bytecode. The driving force in the LVMC design is to encode as many LVM features as possible. To generate the high quality of LVM bytecode, the compiler implements several optimizations and carefully considers the code execution performance. The structure of the LVM compiler is shown in Figure 1.1.

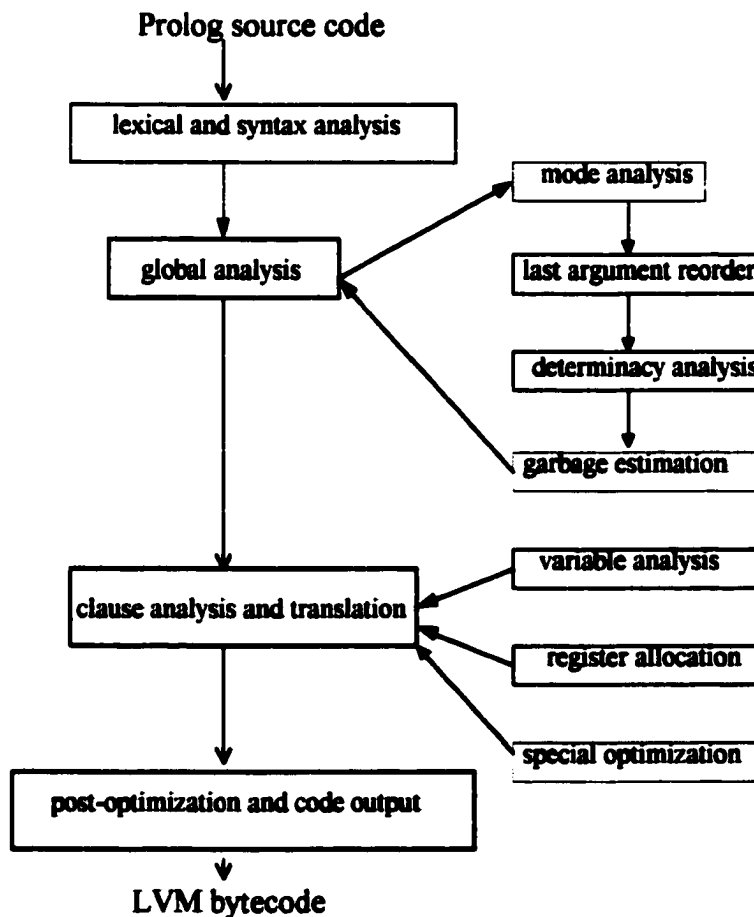


Figure 1.1 Structure of LVM compiler

The LVMC consists of four phases: (1) lexical and syntax analysis, (2) global analysis, (3) clause analysis and code generation, and (4) post-optimization and code output.

In the phase (1), source code is converted to parse trees. Meanwhile the syntactic errors in source code are detected.

In global analysis, the LVMC will collect or generate the argument input mode, then reorder the last arguments of clauses if necessary, and further extract determinacy information of each predicate as well as determine garbage estimation. If a predicate has the mode declaration, the mode analysis of this predicate will be ignored. Although LVM uses the last argument as the first-order discriminator for the purpose of partial unification and control dispatching, there is no constraint on the programmer's programming style. The compiler will dynamically reorder the clause arguments in the reference to LVM based on the mode information. During the determinacy analysis, redundant or unreachable clauses are eliminated.

The clause analysis is the key component of the LVMC. The clause analysis involves the variable analysis, frame allocation and register allocation. The variable analysis has three major tasks:

(a) Each variable must be classified into one of two types: stack variable and register variable. A stack variable is one, which resides on stack and possibly outlives the procedure call. A register variable is a temporary variable, which does not survive across procedure calls. Register variables are mainly used in fact clauses, arithmetic computation or deterministic clause chain-calls. However, they must not occur in any constructor.

(b) The status of variables inside all arguments must be analyzed. A variable may be used in two ways: uninitialized (V-type) or instantiated (L-type). In order to eliminate the unnecessary dereferencing or trailing operations, the V-type variables are bound by destructive assignment.

(c) Since a stack variable may be referenced by several structural terms, the different address offsets of this variable must be calculated according to the stub location in stack.

To speed up the arithmetic computation and term unification of deterministic clauses, soft registers must be scheduled carefully. Register allocation is done using a simple algorithm and register conflict is avoided.

The code generation for each clause is straightforward based on four basic formats. But the initialization of some constructor objects can be delayed until the unification. For a special set of predicates, a special translation algorithm is applied to produce optimized code.

In the final phase of LVMC, post-optimization on the intermediate code is done before the final LVM bytecode output. For example, several consecutive instructions are compacted into one instruction. The arithmetic computation strength is reduced as much as possible. Redundant instructions are removed.

1.4 Outline

The rest of the thesis is organized as follows: Chapter 2 gives a brief overview of Prolog term representation methods: Structure Sharing, Structure Copying and Program Sharing. Then, two virtual machines: WAM and LVM are compared. Chapter 3 describes the lexical and syntax analysis of the LVM compiler. Chapter 4 presents the clause analysis and translation methods. Chapter 5 discusses how to handle arithmetic operations and built-in predicates, and how to specify the initialization instructions. Chapter 6 shows how to use the indexing technique to generate bytecode for a predicate. Chapter 7 discusses some optimizations on the generated bytecode. It includes a special optimization on a set of predicates. Chapter 8 describes how to assist the LVM memory management system at compilation time. Chapter 9 gives conclusions.

Chapter 2. Overview of Prolog Implementation

2.1 Structure Term Representation

The term representation is one of the core parts of any Prolog system. The term representation determines the unification implementation technique in Prolog. Since Prolog is a dynamically typed language, the type and value of a variable is determined only at run time. Therefore, a variable cell is represented in general by a tagged value cell as shown in Figure 2.1.



Figure 2.1 A tagged cell

All Prolog terms are classified into three basic types: variable, constant and structure term. A variable is further categorized into two types: unbound or bound. An unbound variable is self-reference pointer with REF as its tag. In our current LVM system, there are three kinds of constant terms, which are represented as follows

INT	value1
CON	value2
NIL	

where INT denotes an integer, CON a string and NIL a null list.

At the abstract level, two methods are used to represent a structure term, that is, structure copying (SC) and structure sharing (SS). The structure copying technique was introduced by Maurice Bruynooghe [6] and Christopher Mellish [7]. In SC Prolog systems, all terms of different types are fitted into the size of a machine word/register (a tagged value cell). When a variable comes to stand for a structure term during unification, the entire structure including all arguments is copied to a new allocated heap. After the copying, the information about the structure name (functor) and arity becomes indirectly accessible, the knowledge of this structure instance is concealed. For example, the term of $dsg(X, g(X,Y,a), Y)$ could be represented in Figure 2.2 by the SC method

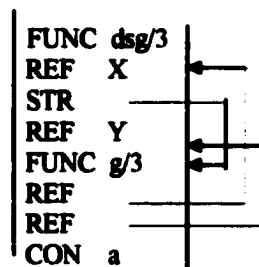


Figure 2.2 Term representation in Structure Copying

The SS method was first introduced by Boyer and Moore [8] and used in earlier

Prolog systems. The SS method takes advantage of the fact that all different instances of a same structure term share a single prototype and differ only in their variable bindings. Therefore, a structure term is divided into two components: skeleton and environment. The skeleton contains the constants and the offsets of environment. The skeleton is stored in the code area. The environment contains the variables and is stored in the heap. Two pointers are needed to represent a structure term, one to the skeleton and other to the environment. Two successive machine words have to be allocated to instance a structure term. For example, the term of $dsg(X, g(X, Y, a), Y)$ could be represented in Figure 2.3 by the SS method

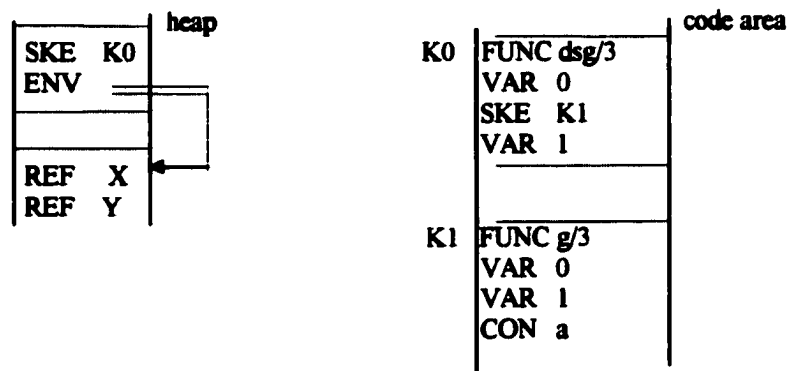


Figure 2.3 Term representation in Structure Sharing

Both structure term representation methods have been thoroughly investigated [6,7]. In general, the SS method is faster than the SC method in creating terms, but the SC method is faster in accessing and unifying terms because the SS method needs more time in decoding skeletons. Also the SC method consumes less memory than the SS method. Only the earlier Prolog system [9] used the SS method. Now the structure copying is the standard implementation method in various Prolog systems. In 1983, David Warren [1] presented a new abstract Prolog instruction set, now called Warren Abstract Machine (WAM), which adopts the SC method as the basic component for the efficient structure term unification. After 1983, most of high performance Prolog systems have been developed on WAM or WAM-like abstract machine for further efficiency improvement.

2.2 Warren Abstract Machine (WAM)

Normally, a Prolog abstract machine is divided into two components as follows:

An abstract machine = instruction set + memory model

2.2.1 Memory Model of WAM

The stack-based memory model of WAM is shown in Figure 2.4. The local stack is used to store the environment and choice point frames. The structure terms are allocated on the global stack, called heap. The trail is used to save the address of

bounded variables that have to be unbound during backtracking. The PDL (push-down list) is used to perform the unification of nested compound terms.

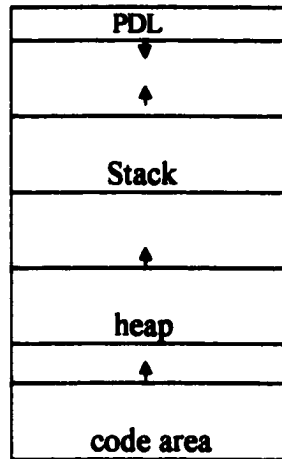


Figure 2.4 WAM data areas

2.2.2 Unification and Parameter Passing

The WAM embeds the unification instructions within the control sequences. Therefore, a single engine is employed for both control and unification. Two steps carry out the unification: (a) the copying of the arguments of the calling goal into arguments registers, and (b) the unification of the arguments registers with the arguments of the head of the called clause. Three types of instructions {PUT, GET, UNIFY} are available to handle unification. The instruction {UNIF} is executed in two modes. In the write mode, a new structure is created, while in the read mode the true unification is performed. As mentioned before, the SC method conceals the skeleton information of a structure term, a general unification algorithm is carried out when both objects are instances of the same structure term. For increasing the efficiency, a unification call is encoded into a sequence of special unification instructions if one of operands is known at compilation time.

2.2.3 Backtracking and Stack Management

The WAM uses two kinds of frames to store the information associated with predicate calls, that is, environment frame and choice-point frame. The environment frame has the following fields:

Parent	parent environment frame
CP	continuation program point
Y1,..., Ym	permanent variables

Figure 2.5 Environment frame in WAM

An environment frame is pushed on the (local) stack before any deterministic clause is executed.

The choice point frame is used to handle the backtracking of a nondeterministic predicate, which has multiple candidates of clauses to try. The choice point frame has the following fields:

BP	alternative program point
CP	continuation program point
E	current environment frame
B	most recent choice point
TR	top of trail
H	top of heap
X1,...Xn	argument registers

Figure 2.6 Choice point frame in WAM

After a new choice point frame is pushed on the stack, the choice point pointer is set to point to the new choice point so that a linked chain of choice points is formed. To increase the WAM efficiency, the clause indexing including conditional branching is necessary to avoid the creation of choice points if possible.

The WAM simulates the conventional procedure call to control Prolog program execution. The two steps of parameter passing result in a bottleneck of its performance. The argument registers have to be saved and restored for the backtracking. The information associated with a predicate call is stored in possibly two frames. A full tail recursion elimination is difficult to implement. Two alternatives to the WAM have been developed. The Vienna Abstract Machine (VAM) [11,12] eliminates the parameter passing bottleneck of WAM by performing the unification of each pair of a goal and a head argument in a single step without the register interface. Another abstract machine, called ATOM (yet Another Tree-Oriented Abstract Machine) [13,14] uses one frame for each predicate call and arguments are passed directly into stack frames. These new abstract machines successfully minimize the inefficiencies of WAM.

2.3 Logic-Inference Virtual Machine (LVM)

Apart from exploring the efficiency of parameter passing and frame allocation, the new abstract machine, called Logic-Inference Virtual Machine (LVM), blends a new structure term representation method - Program Sharing (PS) with SC to represent and handle structure terms. The unification instructions are defined and implemented in a totally different way. A brief introduction to LVM is given in the following.

2.3.1 Program Sharing

Program Sharing is a new method of Prolog structure term representation. This method was introduced by X. Li[2,15,16]. The PS method originating from SS method shares the same idea that the static information of a structure term is separated from its dynamic environments. But the PS method has three main features different from SS:

(1) A structure skeleton is stored in the code area as executable code or a segment of a program. There is no data in a Prolog program. All terms in a Prolog program are compiled into instruction code.

(2) Only one pointer (called a structure code stub) is used to solve the two-cell problem in SS method. The stub content is the entry of a structure code and the stub address is the environment base for executing the structure code. There are three types of structure code stubs: DCI (direct copied instance), SSI (static shared instance) and SCI (static copied instance).

(3) The environment variable indices in the structure code are calculated relative to each stub. In a nested structure term, a variable with multiple occurrences has multiple index values. In SS method, a variable has a unique index value, and variable indices are calculated against a common frame base.

For example, the term of $dsg(X, g(X, Y, a), Y)$ with (-) type of input mode is represented in Figure 2.7 by the PS method.

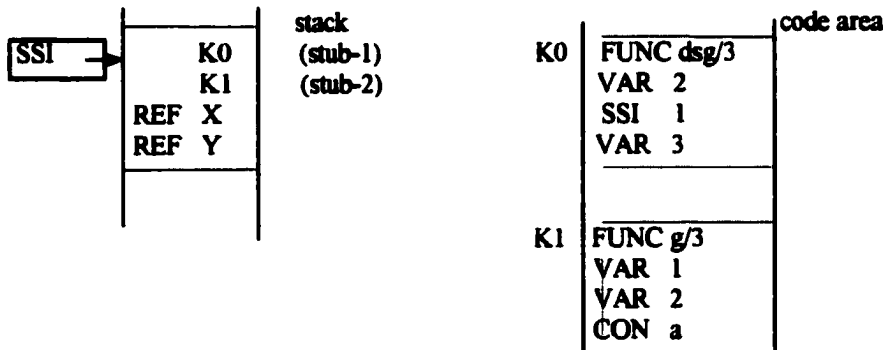


Figure 2.7 Term representation in Program Sharing

Where {VAR, FUNC, SSI and CON} are unification instructions. The variable (X) has the indices of 2 relative to the stub $\langle \text{ , K0} \rangle$ and 1 relative to another stub $\langle \text{ , K1} \rangle$. Similarly, the variable (Y) has two indices (3,2) relative to each stub, respectively. These relative indices are specified in the unification instruction code shown in Figure 2.7. For the structure unification, by accessing a stub we have the entry to the code segment, which defines the structure unification instruction code, and by the stub address we get the environment, which will be consulted to access stack variables and nested stubs.

The PS method has the advantage of both SC and SS methods. It represents terms of different types in the size of a machine word, and also translates the static information into executable instruction code and spends much less overhead for handling dynamic environment.

2.3.2 LVM Architecture and Instruction Set

LVM is a virtual machine based on the mixture of PS and SC methods. The choice between PS and SC to represent a structure term depends on the input mode of the term. For the efficiency, the LVM strongly supports and encourages mode declaration. The mode information of head arguments of clauses can be obtained from the user declaration or through a global analysis [18]. In our current version of LVM, four types of input modes of arguments are defined. They are (-) for out, (?) for in_or_out, (+) for in, (++) for in_and_ground. If a head argument is a compound term, all nested subterms inherit the same mode. For simplicity, all structure arguments in goals are assumed to have (?) mode.

The LVM classifies structure terms into selectors and constructors based on their modes. A structure term is a selector if it can't be bound to any variable outside the clause; otherwise, it is a constructor. Therefore, structure terms with (+) mode are always translated as selectors, and with (-) or (?) mode as constructors. Furthermore, all ground structure terms are treated as selectors, regardless of their input modes. A selector is represented by an instruction <DCI, code_entry>. Since we experience the high efficiency of list manipulation in WAM, LVM uses PS for all non-list constructors and SC for all list constructors. Correspondingly, two more unification instructions are provided. <SSI, stub_offset> represents a static shared constructor of a non-list term. <SCI, stub_offset> represents a static copied instance of a list term.

The LVM memory architecture is shown in Figure 2.8. The major difference from the WAM memory layout in Figure 2.4 is that LVM uses one stack to hold both dynamic objects and control frame information. Also the PDL stack is eliminated.

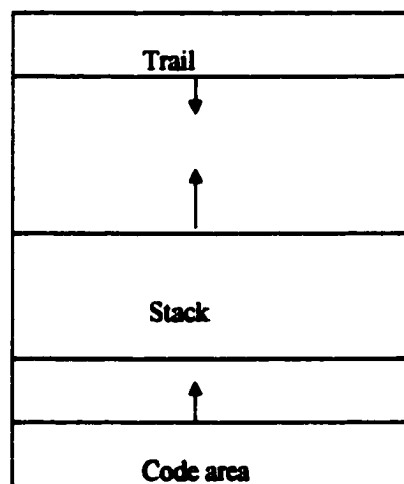


Figure 2.8 LVM Memory Architecture

Appendix A shows the complete instruction set of LVM. They cover unification, arithmetic computation, branching, frame allocation and object initialization. The details were given in X. Li's technical report [17].

2.3.3 Other Features of LVM

Except that LVM uses PS and SC methods to represent structure terms, LVM has other features in the following aspects:

(1) Unification and Parameter Passing

LVM consists of two coordinating processors: a 1P processor for control and a 2P processor for unification. The 2P engine sequentially executes pairwise unification instruction codes. It fetches two instructions simultaneously. Since the type of each operand is known at compilation time and LVM sees terms as executable instructions, the unification algorithm is much cheaper without loss of type information. The LVM compiler generates code segments for the control and unification, which can be loaded in separate locations. For example, with a caller of $f(Y,a)$ and a callee of $f(X,X)$, the unification codes for the caller are:

```
VAR Y
CON value,
```

and unification codes for the callee are:

```
VAR X
VAR X.
```

The 2P engine first unifies two unification instructions: {VAR Y} and {VAR X}, the variable Y becomes bound to the variable X. Then, it fetches next two unification instructions {CON value} and {VAR X}. The constant {value} is directly assigned to the variable X. Therefore after the unification, the variable Y becomes a constant cell.

LVM eliminates the parameter passing bottleneck by unifying caller and callee arguments in one step. Unlike the Vienna abstract machine (VAM_{2P}) [11,12], the LVM unification instructions are neutral. Also LVM delays the structure full unification until it is necessary in order to enhance the system efficiency.

(2) Backtracking and Stack Management

LVM implements procedure invocation and backtracking by allocating different chains of control frames on the stack. There are three kinds of control frames in the LVM: V-frame, C-frame and B-frame. Separation of V-frame from C-frame is for the minimization of memory usage and the speed up of frame reallocation.

(a) V-frame: It only consists of dynamic objects without control information. It is used for fact or chain-call clauses involving constructors.



Figure 2.9a V-frame format in LVM

(b) **C-frame:** It has three cells for control information and other cells for dynamic objects. It is used for any clause with more than one user defined goals in its body.

Variables
Stubs
ST: current stack top
CP: continuation program entry
AF: continuation frame

Figure 2.9b C-frame format in LVM

(c) **B-frame:** It contains all information needed for backtracking shown in Figure 2.9c. A B-frame is allocated at a new choice point.

NT: alternative program entry
TT: trail top
GP: caller's unification code
GF: caller's unification frame
R0: R0 register
CP: continuation program entry
AF: continuation frame
BB: most recent choice point

Figure 2.9c B-frame format in LVM

(3) Garbage Collection and Memory Reclaim

Since LVM uses one stack to hold both frames and dynamic objects, an efficient garbage collection (GC) algorithm is very important to the LVM implementation. In Chapter 9, a chronological garbage collection algorithm will be discussed. The LVM needs the compiler to generate GC instructions to tell the execution system when, where and what should be collected, so that garbage estimation must be carried out in the earlier phase of compilation. The experimental LVM system showed that the GC algorithm has very low runtime overhead.

In summary, several advantages of the current LVM are:

- (a) Memory usage is minimized for the structure term representation.
- (b) It is easy to allocate/reallocate stack frames.
- (c) The environment management is simple due to one stack policy.
- (d) There is no cost in parameter passing from a stack frame to another frame.
- (e) During backtracking, it is unnecessary to check binding directions, such as from local objects to global objects and from young generation to old generation, and the trailing/detrailing operation is cheap.
- (f) A natural generation line of procedure calls exists for the implementation of garbage collection.

Chapter 3. Lexical and Syntax Analysis

In this chapter, the first phase of the LVMC, which is the lexical analysis and syntax analysis of Prolog source code, is described. A simple error handler is used in the LVMC. When this phase is successful, a parse tree is generated for each clause. This chapter is organized as follows. First, the syntax of Prolog texts in standard Prolog is described. Then, the detailed method of lexical analysis and syntax analysis is shown.

3.1 Syntax of Prolog Text

A Prolog text is a sequence of directives and clauses in an order which is specified by directives. Directives and clauses are represented by terms. In standard Prolog [19], terms are written in function notation. Therefore, the structure of a term is specified without any ambiguity. The syntax of terms defines the syntactic rules for writing terms correctly. Here the syntax is presented using the context free grammar. The context free grammar has the form:

```
nonterminal  --> sequences of nonterminals and terminals
terminal     --> sequences of characters
```

3.1.1 Prolog Character Set

Terminal symbols, called tokens, consist of sequence of Prolog characters. There are five types of Prolog characters defined as:

```
prolog_char  --> alpha_num_char
              | graphic_char
              | solo_char
              | layout_char
              | meta_char
```

(1) Alpha_numeric characters

```
alpha_num_char --> _ | digit_char | letter_char

letter_char    --> capital_letter | small_letter
  capital_letter --> [A-Z]
  small_letter  --> [a-z]
digit_char     --> decimal_digit_char
              | binary_digit_char
              | octal_digit_char
              | hexa_digit_char
```

```

decimal_digit_char  --> [0-9]
binary_digit_char   --> 0 | 1
octal_digit_char    --> [0-7]
hexa_digit_char     --> [0-9] |[A-F]|[a-f]

```

(2) Graphic characters

```

graphic_char  --> # | $ | & | * | + | - | . | / | ;
               | < | > | = | ? | @ | ^ | ~

```

(3) Solo characters

```

solo_char  --> ! | ( | ) | [ | ] | { | } | | | , | ; | &

```

(4) Layout characters

```

layout_char  --> SP           // space character in ASCII
               | NL           // new-line character in ASCII
               | HT           // tab character in ASCII

```

(5) Meta characters

```

meta_char  --> \ | ' | ' | "

```

3.1.2 Syntax of Term

```

term  --> var | atom | integer | floating point | compound_term

```

The floating point number is not implemented in current version of the LVM system, but the compiler still parses the floating point numbers.

(1) Variables

Variables are strings of letters, digits and underscore starting with capital letter or underscore.

```

var  -->  named_var | anonymous_var

named_var  -->  capital_letter (alpha_num_char)*
               | _ (alpha_num_char)+

anonymous_var --> _

```

(2) Atoms

Atoms can be constructed in three ways:

- (a) Strings of letter, digits and underscore starting with a lower-case letter;
- (b) Strings of characters enclosed in single quotes;
- (c) Strings of special characters

```
atom --> letter_digit_token
      | quoted_token
      | graphic_token
      | semicolon_token
      | cut_token

letter_digit_token--> small_letter (alpha_num_char)*
quoted_token --> ' char_string '
graphic_token --> \
                | graphic_char_string
semicolon_token --> ;
cut_token      --> !
```

(3) Numbers

There are two types of numbers: integer and floating point.

```
integer --> integer_constant
          | binary_constant
          | octal_constant
          | hexa_constant

integer_constant--> decimal_digit_char
                  | decimal_digit_char integer_constant
binary_constant--> Ob binary_digit_char (binary_digit_char)*
octal_constant --> Oo octal_digit_char (octal_digit_char)*
hexa_constant  --> Ox hexa_digit_char (hexa_digit_char)*

floating point --> integer_constant . integer_constant [exponent | ε]

exponent      --> [e| E] sign integer_constant
sign          --> + | - | ε
```

Here ϵ is an empty string.

(4) Compound_term

Compound terms can be written in one of four notations. They are:

(4.1) Function notation (bracketed expression)

compound_term --> atom (arg_list)

arg_list --> term (, term)*

where the name of function is called functor, and the number of arguments called arity. The outer-most functor is called its principal functor, e.g., $.(a1.(a2.(a3,[])))$.

(4.2) List notation

The principal functor is $./2$, and the first argument is a term and the second a list.

compound_term --> [term items]

items --> (, term)*
| | term
| NIL_list

NIL_list --> []

where a special case of NIL_list is treated as a special atom in LVMC. For example, $[a1,a2,a3]$, $[a1,a2|a3]$ and $[a1,a2|[b1,b2]]$ are list terms.

(4.3) Curly notation

compound_term --> { term }
| (term)

(4.4) Operator notation

Some terms are written as unbracketed expressions using functors in operator notation. Each operator is characterized by three parameters: name (an atom), specifier (one of xf,yf,xfx,xfy,yfx,fx,fy) and priority (an integer between 1 and 1200).

The specifier of an operator defines its arity (1 or 2), class (prefix,infix,postfix) and associativity (left-,right-,non-associative). Table 3.1 lists all types of specifier.

The priority of a term is 0 if it is written in functional, list notation, or it is a bracketed expression or atomic term. If a term is written in operator notation, its priority is the priority of the principal functor. The predefined operators are listed in Table 3.2.

Specifier	arity	class	associativity
fx	1	prefix	non-
fy	1	prefix	right-
xfx	2	infix	non-
xfy	2	infix	right-
yfx	2	infix	left-
xf	1	postfix	non-
yf	1	postfix	left-

Table 3.1 Specifiers of operators

Operator	specifier	priority
:- -->	xfx	1200
:- ?-	fx	1200
;	xfy	1100
->	xfy	1050
,	xfy	1000
\+	fy	900
= \=	xfx	700
== \== =:	xfx	700
@< @=< @> @>=	xfx	700
is := =\= < <= > >=	xfx	700
+ - ^ \	yfx	500
* // rem mod << >>	yfx	400
**	xfx	200
^	xfy	200
- \	fy	200

Table 3.2 Predefined operator table

```

compound_term    --> atom_preop term
                  | term atom_postop
                  | term atom_inop term

```

3.1.3 Clauses

```

clause --> head :- body
        | predication

```

where the first grammar defines a rule with ':-'/2 as the principle functor, the second grammar defines a fact.

```

head --> predication

```

```

body --> (body , body) // conjunction: principal functor ','/2
      |(body ; body) // disjunction: principal functor ';' /2
      |(body -> body) // implication: principal functor '->' /2
      |variable
      |predication

```

where the functor name of predication must be different from ','/2, ';' /2 and '->' /2.

```

predication --> atom | compound_term

```

where the compound_term must be callable term with arity >= 0.

3.1.4 Directives, Query and Declarations

The principal functor of directives is (-:)/1.

```

directive --> :- directive_term
directive_term --> directive_atom ( predication )
directive_atom --> discontinuous
                  |dynamic |ensure_loaded
                  |include |initialization
                  |multifile

query --> ?- predication

declaration --> mode_declaration
              |determinacy_declaration
              |gc_declaration

mode_declaration --> :- mode_atom ( mode_type (,mode_type)* ) .
mode_type --> ++ | + | - | ?
determinacy_declaration --> :- determinacy_atom/arity (integer_constant)
arity --> [0-9]
gc_declaration --> :- garbage_atom/arity ( integer_constant )

```

3.1.5 Comments

There are two types of comments: single line and multiple line comments.

```

comment --> single_line_comment | bracketed_comment

single_line_comment --> % comment_text new_line_character
bracketed_comment --> /* comment_text */

comment_text --> prolog_char comment_text | ε

```

In summary, the Prolog text is defined by the following grammar:

prolog_text --> (comment|query|declaration |clause |directive)*

3.2 Token Generator

The role of the lexical analyzer is to scan characters of program text and convert the input into a stream of tokens. The token with its attribute is passed to the parser. Therefore, the lexical analyzer and parser form a token producer-consumer pair shown in Figure 3.1.

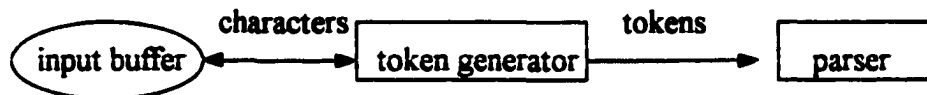


Figure 3.1 Interface to the lexical analyzer

The lexical analyzer is implemented by a token generator. An input buffer is set up for reading and pushing back characters. A block of 256 characters is read into the buffer from the file at a time. A pointer keeps track of the input position. A lookahead character is checked. It should be noted that all atoms including the operators belong to the `name_token`. The number_tokens are further classified into `integer_token` and `float_token`.

3.3 Syntax Analysis

In the compiler design, the syntax analysis of Prolog programs is implemented by a parser. The parser accepts a stream of tokens from the token generator and verifies that the stream of tokens can be generated from the Prolog grammar specified in Section 3.1. The output of the parser is a representation of the parse tree, which will be used in the next phase of the compiler. Meanwhile, a collection of information about a predicate is put into a symbol table. The interface of the parser is shown in Figure 3.2. In the current version of LVMC, the syntax error is handled in three steps:

- (a) reporting the presence of errors,
- (b) printing out the reason of errors, and then
- (c) terminating the compiler.

A better syntax error handler would be implemented in the future version of the compiler.

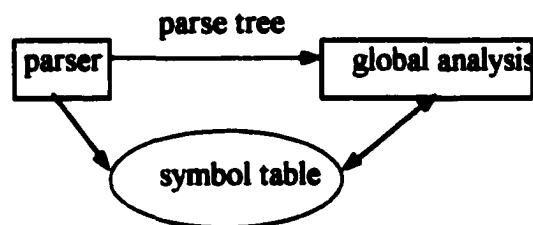


Figure 3.2 Role of parser in the compiler

Since the grammar of Prolog is relatively simple, an unambiguous grammar for Prolog is reconstructed so that no production right side has two adjacent nonterminals. Therefore, an easy-to-implement form of bottom-up syntax analysis method, called operator-precedence parsing [20], is employed in the parser. In the implementation, a stack with elements, of which data structure is defined as

```
typedef struct stacktype {
    int tokentype;
    int priority;
    int specifier;
    argument_information *term;
    structure stacktype *next;
} stacktype;
```

is used to hold the input tokens. The priority and specifier of some predefined operators is listed in Table 3.2. The basic parsing strategy of LVMC is

- (a) Initially, the parser sets up an empty stack, and accepts tokens one by one until an end_token is reached. When the end_token arrives, the parser outputs the parse tree of the clause if there is only one term remaining on the stack. Otherwise, it invokes syntax error handler.
- (b) For each token, if the token is one of the set of operators, then the parser invokes a corresponding reducing function to reduce the top elements on the stack. Otherwise, it simply shifts or pushes the token onto the stack.

Chapter 4. Clause Analysis and Translation

The clause analysis is the crucial part of the LVMC. It involves the structure argument classification, variable analysis, register scheduling and V/C-frame allocation. They will be discussed in detail in Section 4.1-4.3.

The translation of a clause generates two streams of instruction code: control instruction code and unification instruction code, in separate segments. In general, the control instructions for a clause include instructions of stack allocation, object initialization, unification code invocation, procedure invocation and garbage collection. The unification instructions represent all arguments of the head and the bodies in a clause. During the compilation, various code segments are labeled uniquely and systematically. The name conventions are listed in Table 4.1. The clause translation will be discussed in Sections 4.4 and 4.5.

labeling	Meaning	Code type
name/arity	predicate code entry	control code
name/arity.i	clause code entry	control code
name/arity.i.i	branching code entry	control code
name/arity.i.u.0	head unification code entry	unification code
name/arity.i.u.i	goal unification code entry	unification code
name/arity.i.s.i	selector or constructor code	unification code

Table 4.1 Code segment-labeling rules

4.1 Structure Argument Representation and Flattening

The LVMC classifies the structure arguments of a clause into three types: selector, constructor and dual. A structure argument is a selector if it is ground or its input mode is (+, ++). A structure argument is a constructor if its input mode is (- or out) and it contains at least one variable. If a structure argument has input mode of (? or in./out), it may be used as either a selector or a constructor, and also contains at least one variable, then it is a dual. Here, we can see that the types of structure arguments of a clause really depend on the input modes of the arguments. The input mode of a structure argument is obtained either from mode declaration or from mode global analysis.

If a compound structure has a certain mode, then all nested substructures inherit the same mode. The mode declaration applies to head arguments only. For the goals of a clause, we assume that all structure arguments are duals. It is enforced that a ground structure argument is always a selector regardless of its input mode.

The LVM uses instruction: `DCI c_entry` to represent a selector. Here `c_entry` specifies the code entry of the whole structure unification in code area. A constructor is represented either by instruction: `SSI offset if it is` a non-list structure or by instruction: `SCI offset if it is a list`. Here `offset` denotes a relative position of this stub with respect to a LVM stack frame of the clause. Since a pure constructor will be bound to an arbitrary variable during execution, the stub must be allocated inside its associated stack frame to represent a dynamic instance creation. For a clause, a V-frame or C-frame may need to be allocated (See Section 4.5). The order of stub and variable allocation is shown in Figure 4.1.

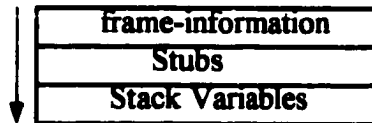


Figure 4.1 Order of stub and variable allocation on stack

During compiling a clause, all nested structure arguments must be flattened in order to determine the number of stubs and their offsets. A stack with element defined as

```
typedef struct fstack{
    char ftype,
    boolean delayable,
    char *functor,
    unsigned index,
    unsigned arity,
    arginf *arg_list,
    struct fstack *next
} fstack;
```

is used to implement the flattening algorithm. The flattening algorithm is:

(a) scan arguments of a clause from left to right. For each structure argument, push an element with data structure of "fstack" on the stack. The field of "ftype" of the element depends on its input mode, it may be one of types {SSI,SCI,DCI}. The field of "delayable" is reserved for {SCI,SSI} elements to implement initialization delay optimization. Only if a structure argument appearing in head of a clause has (?) -type of input mode, the "delayable" is set to "No". In all other cases, the field "delayable" of {SSI,SCI} elements is set to "YES".

(b) scan all elements of the stack from top to bottom. If its "arg_list" has substructure, an element of "fstack" type for the substructure is inserted upon its parent in the stack.

(c) repeat (b) until there is no nested structure in the field of "arg_list".

(d) index all SSI elements from top to bottom and then index SCI elements from top to bottom by second pass. A SSI element requires one cell in LVM stack frame, but a SCI element needs two cells for copying its structure term.

Example 4.1 Let's assume we have the following clause with mode declaration.

```
-----
:-pss mode(+,-,+).
pss(X, h(Y,g(a,b)), f(X,g(Z,b)):-qs(j(Z,k(a,b),W)),rs(X,Y,Z).
-----
```

After the flattening procedure (a), we have the following working stack:

type	index	delayable	arg_list
SSI	0	Y	h(Y,g(a,b))
DCI	1	-	f(X,g(Z,b))
SSI	1	Y	j(Z,k(a,b),W)

After all nested structures are completely flattened, the contents of the working stack are:

type	index	Delayable	arg_list
SSI	0	Y	h(Y, DCI:1)
DCI	1	-	g(a,b)
DCI	2	-	f(X, DCI:3)
DCI	3	-	g(Z,b)
SSI	1	Y	j(Z, DCI:4, W)
DCI	4	-	k(a,b)

For clause pss/3, the number of stubs is 2. The indexing of {SSI} elements starts from 0.

Example 4.2 This example has list arguments. The LVM uses Structure Copying method to represent them. The cells of {SCI} type need to be allocated on LVM stack frame.

```
-----
:-psc mode (+,-,-).
psc([X|Y], [X,a,b],[[Y],d]).
-----
```


After the first procedure (a) of the algorithm, we have the following working stack:

type	index	delayable	arg list
DCI	1	-	[X Y]
SCI	0	Y	[X [a,b]]
SCI	2	Y	[[Y],d]

After completely flattening all nested lists and indexing all SCI elements again, the working stack becomes:

type	index	delayable	arg list
DCI	1	-	[X Y]
SCI	0	Y	[X DCI:2]
DCI	2	-	[a DCI:3]
DCI	3	-	[b []]
SCI	2	Y	[SCI:4 DCI:4]
SCI	4	Y	[Y []]
DCI	4	-	[d []]

For this clause psc/3, three SCI elements need to be copied to LVM stack frame through special initialization instructions (See Section 4.4).

4.2 Variable Classification and Index Calculation

4.2.1 Classifications of Variables

The LVMC classifies the variables of a clause into two types:

(1) **stack variable (S-type):** A stack variable is one which possibly outlives a procedure call and must reside in the LVM stack frame. A stack variable is represented by an offset to its dynamically allocated stack frame and is accessed by "base plus index" method.

(2) **register variable (R-type):** A register variable is a temporary variable which does not survive across procedure calls. A register variable is used in a fact clause, arithmetic computations or a deterministic chain call. A register variable never occurs in any constructor.

Except above two types of variables, a special variable, called "void" variable, may exist in a clause. A void variable is a variable which occurs once and only in the argument with (+) input mode of head of a clause. A void variable must be declared by "_" in a clause. The LVMC treats a void variable as constant term by an instruction:

VOID

A stack or register variable may be used in two ways: uninitialized (V-type) or instantiated (L-type). The LVMC uses {RGV and RGL} to represent the two states of a register variable respectively, and {VAR and VAL} to represent the two states of a stack variable respectively. A V-type variable is a variable without initial value. A L-type variable is a variable with initial value, but may be unbound. Here, unbound means that the variable is initialized as a self-referential pointer. Binding a L-type variable is expensive because the variable must be dereferenced. One of the major sources of inefficiency of early WAM implementations arises from that V-type variables are created as self-referential pointers and then unified soon afterward. Beer [21] first noticed this problem, and proposed a solution that the V-type variables are bounded via destructive assignments without dereferencing. In other word, the initialization of V-type variables can be canceled. The LVM bounds the V-type variable by destructive assignment. The analysis of variable status in a clause is important to improve the efficiency of any implementation.

Naturally, the next question is how to identify status of variables in a clause. Van Roy [22] used a global analysis algorithm to extract the uninitialized variable from a clause. Lindgren [23] detected the uninitialized variables by a syntactic transformation method. In the LVMC, we supposed that the input modes of arguments of predicates are known before the clause analysis, the algorithm of variable analysis becomes simple. Our algorithm also takes the LVM unification method into account.

The unification method of LVM system is quite different from all others. Most Prolog systems implement unification in the depth-first order, but the LVM invokes the unification code segments in the reverse depth-first order. When two unification instructions represent both structure terms (either selector or constructor), the unification of deeper-level structure terms is delayed unless those two instructions are the last pair in the current code segments. In other word, partial unification is performed.

In the variable analysis, a linked list "xlist" with node data structure defined as

```
typedef struct varinfo {
    char *var-name,
    char RS-type,
    char used,
    char first-occurred,
    char delayable,
    struct varinfo *next
} varinfo;
```

is used to hold the variable information in a clause in order to decide variable R/S-type and necessity of variable initialization as well as possible initialization delay. The LVMC uses four rules to set first-occurred and delayable flags:

- (1) If a stack variable occurs in plain argument list of head, its first-occurred flag is set to true;
- (2) If a stack variable occurs in any head structure argument of selector before any constructor argument during scanning and flattening the head argument lists, its first-occurred flag is set to true;
- (3) If a stack variable occurs in arguments of goals before any constructor argument during scanning and flattening the goal argument lists, its first-occurred flag is set to true.
- (4) If a stack variable doesn't appear in structure arguments of head with (?) mode, then its delayable flag is true.

After the analysis, the R/S-type of all variables is set correctly. For a stack variable in the "xlist", if its first-occurred flag is false, this variable must be initialized. Furthermore, if its delayable flag is true, this variable initialization can be delayed until after head unification control instruction.

The V/L-type information is associated with each variable. A variable may appear in the argument lists of head and goals or inside structure argument terms. Assuming that the clause is

$$H(..) :- G1(...), G2(...), \dots, G_n(...),$$

the LVMC uses the following algorithm to determine the V/L-type of a variable:

- (1) Scan the argument lists of head and flatten its structure terms. Then do the same for each goal argument in order of $G_1 \rightarrow G_2 \rightarrow \dots \rightarrow G_n$.
- (2) following (1) process, if a variable appears first time and the current argument is plain argument or selector, then it is V-type; otherwise it is L-type.

Actually, the variable analysis is naturally embedded into the structure term flattening process. The void variables always are of V-type.

Example 4.3 To see how the variable V/L type identification works, let's consider the following clause. All variables in this clause are stack variables. The V/L type of variables is listed in the right side.

:-mode p(+,-,+).

p(X,	X: V-type
h(Y, g(Z)),	Y: L-type, Z: L-type(*)
f(X,g(Y))	X: L-type, Y: V-type
):-	
q(j(Z,W)),	Z: L-type, W:L-type(*)
r(X,Y,Z).	X: L-type, Y: L-type, Z: L-type

The LVMC scanning order of arguments is:

$X \rightarrow f(X,g(Y)) \rightarrow h(Y,g(Z)) \rightarrow j(Z,W) \rightarrow X \rightarrow Y \rightarrow Z$.

Here, variables {Z,W} need to be initialized.

4.2.2 Indexing of Variables

At each clause invocation point, a stack frame (V/C-frame) and registers are used to hold dynamic objects generated from the called clause. The stack frame is allocated in the LVM stack area. Although the size of a V/C-frame varies from one clause to another, the stack frame is an integral memory in the LVM working stack. Later on, any object on the stack can be accessed using the base plus offset addressing method. Normally, the LVM uses the bottom of going allocated frame as the consulting base. The offset is the distance of location of the object inside the stack frame from the frame bottom in units of machine words.

A stack frame may contain environment parameters, stubs, SCI objects and stack variables. In order to prevent negative offsets in nested unification instructions, the LVMC assigns the stack cells by the following rule:

- (1) Cell assignment starts from the relative location 0;
- (2) Cells are assigned by the order of environment parameter->stubs-->SCI objects-->stack variables;
- (3) Stub arrangements are consistent with their flattening order to guarantee the correct accessing scope of the nested stubs and stack variables;
- (4) Stack variables, which need initialization, are assigned with lower indices than those uninitialized.

In the LVMC, there are two types of offsets: S-offset and C-offset. The S-offset is an offset against the base of the allocated frame. The S-offset is used to generate the unification instructions for the head and goal arguments, and also to access the variables in the structure term of a selector. Therefore, the following objects have S-offsets:

- (1) stack variables, which directly are plain arguments of the head and goals;
- (2) stack variables, which occur in structure terms of selector or DCI objects;
- (3) stubs/SCI objects, which are arguments of the head and goals.

For a constructor stub or SSI object, since it will be bound to a variable and then be carried to any place during execution, so its environment is created to associate with the stub. Therefore, the absolute address of the stub serves as the base of the unification code environment and its content as the unification code entry pointer. A C-offset is an offset against the location of a constructor stub. The following objects have C-offset:

- (1) stack variables, which occur inside structure terms of constructor;
- (2) stubs/SCI objects, which are substructures of a structure term of constructor.

Since all stubs, SCI objects and stack variables have been assigned locations in the stack frame during compilation, the offset calculation becomes easy. Let X be any object in C and C a constructor stub, then

$$\text{C-offset (X in C)} = \text{location(X)} - \text{location (C)}.$$

The S-offset of an object is its relative location with respect to the stack frame.

Example 4.4 Let's consider the following clause:

```
-----
:-mode p(-,?,-).
p(f(Y,g(Z)),X,h(Y,g(W))):-q(X,Y), r(X,Z).
-----
```

After nested structure flattening and variable analysis, we find that there are four stubs and 4 stack variables {X,Y,Z,W}. Stack variables {X,Y,Z} have first occurrence, so they don't need initialization. The variable W must be initialized. A C-frame is used to hold all objects. Thus a total of eleven cells will be allocated as the execution environment of the clause on the LVM stack area. The cell assignments are listed in the Table 4.2.

In the head unification codes of clause p/3, we use S-offset: offset 3 to refer to C1, offset 8 to X and offset 5 to C3. The variable Y has three offsets: one S-offset and two C-offsets. Since variable Y occurs in constructors C1 and C3, the two C-offsets are:

$$\text{C-offset (Y in C1)} = \text{location (Y)} - \text{location (C1)} = 9-3 = 6$$

$$\text{C-offset (Y in C3)} = \text{location (Y)} - \text{location (C3)} = 9-5 = 4$$

Therefore, in the unification code segment of C1, we use C-offset 6 to refer to Y and C-offset 1 to refer to C2.

Location	Object	S-offset	C-offset
0	c-frame env		
1	c-frame env		
4 2	c-frame env		
3	C1=f(Y,C2)	C1:3	Y: 6 C2: 1
4	C2=g(Z)	C2:4	Z: 6
5	C3=h(Y,C4)	C3:5	Y: 4 C4: 1
6	C4=g(W)	C4:6	W: 1
7	W	7	
8	X	8	
9	Y	9	
10	Z	10	

Table 4.2 Stack cell assignment and offsets of objects

Since the LVM uses the Structure Copying method to represent the list constructors, it is possible that several copied list instances share a single variable. Therefore, a special set of list initialization instructions are designed to handle the shared occurrence problem. The detail will be discussed in Section 4.4.

4.3 Register Allocations and Conflict in Parameter Passing

In the LVM, there are 64 general purpose registers emulated by memory, which can be directly accessed without indirection. Eight of them labeling from R0 to R7 are dedicated to pass parameters between the caller and the callee. Others from R8 to R63 are used to store the intermediate results in arithmetic computation. Here, we only discuss the parameter passing registers.

4.3.1 Magic Register: R0

The role of R0 register is quite different from that of other registers (R1..R7). R0 register is used to speed up last argument dispatching. Although Prolog programmers have a natural tendency to write the codes using discriminating patterns as first arguments, the LVM does not depend on this tendency. Since the LVM implements unification delay algorithm, the LVMC chooses an appropriate argument from the head arguments of a clause and swaps it with the last argument, if one or more head arguments have (+) input mode. In such cases, the clauses of a predicate may be dispatched by their last arguments. To speed up the dispatching, the LVM uses R0 register to store the last argument before issuing a procedure call. Two control instructions {SWT and SHS} are designed to implement the dispatching based on the R0 value.

If a predicate is last argument dispatchable, the LVMC uses the following conditions to select a variable from the last argument of a head for each clause to be the R0 register variable:

- (1) In the head, the variable occurs in the last argument once.
- (2) Since the R0 represents a dereferenced value of the corresponding variable in the goal side of the clause, the variable may be used in any arithmetic expression before the first user defined goal. The variable can only appear in the first user defined goal but not other goals, and can also be used as its last argument.

Example 4.5 To illustrate the selection, let us consider the following two clauses, which are last argument dispatchable.

```
:-mode p1(-,+,+).
p1([X|Xs],Y,Z):-Y>Z, W is Z+1,p1(Xs,W,Z).
:-mode p2(-,+).
p2([X|Xs], f(Y,X)):-p2(Xs,Y).
```

In the first clause, variable Z is selected as R0 register variable. The variable Z is the last-argument of the head, also the last argument of user defined goal p1/3. Before the invocation of the goal call p1/3, the variable Z is used in two arithmetic expressions. In the second clause, variable Y is selected as R0 register variable.

Except speeding up the dispatching, R0 register can help to pass parameters faster between the caller and callee. When the callee is last argument dispatchable, the caller can directly store its last argument in R0 regardless of the properties of the caller predicate, then the later-on unification process directly involves R0 register. If the caller predicate is nondeterministic, R0 is protected in choice point or B-frame. To implement this optimization, the LVM designs two sets of procedure invocation instructions {CAL, CCL, LAC} as follows:

CAL	u_code_entry	predicate_code_entry
CAL Ri/Vi	u_code_entry	predicate_code_entry
CCL	u_code_entry	predicate_code_entry
CCL Ri/Vi	u_code_entry	predicate_code_entry
LAC	u_code_entry	predicate_code_entry
LAC Ri/Vi	u_code_entry	predicate_code_entry

If the first operand of the procedure invocation instructions is of R_i/V_i , it means that the specified register (R_i) or stack variable (V_i) is dereferenced and its final value is saved in R_0 . Then the last instruction in the goal unification code segment is replaced by instruction: "RGL 0". The LVMC arranges the R_0 register to pass the last argument of a goal call when the goal predicate is last argument dispatchable.

4.3.2 $R_1...R_7$ Registers

$R_1...R_7$ register variables can only be used in fact, deterministic chain clause to pass parameters. Unlike R_0 , those register variables can't be used in nondeterministic chain clause and normal rule clause. If these register variables are not enough to represent the parameter passing variables in a clause, then the remainder of parameter variables will be treated as stack variables. In this case, extra stack cells need to be allocated on the LVM stack frame.

During register allocation, a conflict may happen. To demonstrate this problem, let us check the following deterministic clause p/2:

$$p(X, Y, Z) :- q(Y, W, X).$$

After variable analysis, the clause is first transformed into:

$$p(R_1, R_2, R_3) :- q(R_3, R_4, R_1).$$

Here, p/3 passes its parameters to q/3 goal via registers in the order of $R_3 \rightarrow R_4 \rightarrow R_1$. If the predicate q/3 uses " $q(R_1, R_2, R_3) :- \dots$ " to accept these parameters, after the caller's R_3 unifies with the callee's R_1 , the caller's R_1 information has been lost, therefore the register conflict occurs. To avoid this conflict, the LVMC uses the following strategies to schedule $R_1..R_7$ registers in register allocation:

- (1) Registers ($R_1..R_7$) are allocated incrementally by the order of index to the head and goal arguments from left to right.
- (2) If the current register variable index is less than the left-hand register variable index, a new register variable with a greater index will replace this register variable, and a register to register assignment will be inserted before this goal call.

Thus, in the above example, the LVMC will change the clause to be in the form of

$$p(R_1, R_2, R_3) :- R_5=R_1, q(R_3, R_4, R_5).$$

4.4 Unification Code of a Clause

For a clause, there may be several independent unification code segments for the head and its goal arguments respectively. Each unification code segment consists of several unification instructions. These unification instructions have a single format of

Opcode	Operand
--------	---------

Totally twelve unification instructions are defined in LVM and listed in Table 4.3.

Opcode	Operand	Meaning
INT	value	an integer
CON	value	a constant
NIL		a null list
VOID		a void variable
VAR	offset	an uninitialized stack variable
VAL	offset	an instantiated stack variable
RGV	index	an uninitialized register variable
RGL	index	an instantiated register variable
DCI	entry	a direct/dynamic-copied instance
SSI	offset	a static shared instance
SCI	offset	a static copied instance
FUN	name/arity	a functor with name/arity

Table 4.3 LVM unification instructions

Unification code segments of a clause can be generated in any order, provided that they are properly labeled. The LVMC uses the name conventions listed in Table 4.1. Inside each code segment the order of unification instructions must obey the lexical order of the objects in the argument list.

After the clause analysis, the unification code generation method is simple. It simply describes the argument lists of head and goals from left to right. For a flattened structure stored in "fstack" (See Section 4.1), it starts with a FUN instruction and follows with a sequence of instructions to match its arguments. But a selector list is treated as a structure without functor so that the FUN instruction is omitted.

Example 4.6 Suppose that we have the following clause p/3:

```

-----
:- mode p(+,-,+).
p(Y,f(X,Y,Z),[X|Xs]):-p(Y,Z,Xs).
-----

```

After the clause analysis, we find that there are three stack variables {X,Y,Z}, a R0 register variable {Xs}, and one structure stub as shown in Table 4.4.

Object	Object type	S-offset	C-offset
f(X,Y,Z)	SSI	0	
[X Xs]	DCI	-	
X	Stack var	1	1
Y	stack var	2	2
Z	Stack var	3	3
Xs	Register var	0	

Table 4.4 Memory layout of objects for p(+,-,+)/3 clause

For this clause, four unification code segments should be generated. One is for the clause head arguments, one is for the goal arguments and other two segments are for the nested structure terms in head arguments. The unification code segments are labeled as follows:

U-code Labeling	Meaning
p/3.u.0	head unification
p/3.s.1	constructor f(X,Y,Z)
p/3.s.2	selector [X Xs]
p/3.u.1	goal unification

These unification code segments are:

```

-----
p/3.u.0:  VAR 2          // Y
          SSI  0          // f/3 stub
          DCI p/3.s.2    // [X|Xs]
p/3.u.1:  VAL 2          // Y
          VAL 3          // Z
          RGL 0          // Xs
  
```

```

p/3.s.1:  FUN f/3
          VAL 1      // X
          VAL 2      // Y
          VAL 3      // Z
p/3.s.2:  VAR 1      // X
          RGV 0      // Xs

```

In this example, a FUN instruction for the list [X|Xs] is omitted from the unification code segment of p/3.s.2 , since the list is a selector.

4.5 Control Code of a Clause

In the LVM, each clause has one corresponding control code segment. The control code specifies possible stack frame allocation, necessary initialization, and unification invocation and consecutive goal calls. It may also contain arithmetic computation instructions and garbage collector trigger. The initialization includes stub initialization, SCI object initialization and V-type variable initialization.

Two types of LVM stack frames may be allocated to each clause: V-frame and C-frame. They have been discussed in Section 2.3. A V-frame is allocated by instruction:

ALV	n
-----	---

and a C-frame is allocated by instruction:

ALC	n
-----	---

where n is the total number of cells of this frame in machine word. For a V-frame, if n=0, then the ALV instruction is omitted. For a C-frame, the value of "n" must be equal to or greater than 3, since it contains three environment parameters. The LVMC uses two simple conditions to identify the stack frame type for a clause:

- (1) A V-frame is used for a fact or chain-call.
- (2) A C-frame is used for a rule clause, which has more than one goal.

The control code generation for a clause is straightforward. It follows four basic translation formats:

(1) Fact:

[ALV] -> {initialization} *->[UNI] ->{initialization} *->PCD.

(2) Chain call:

[ALV]->{initialization} *->[UNI]->{initialization} *->CCL.

(3) Rule:

ALC-> {initialization} *->[UNI] ->{initialization} *->{CAL}+ ->LAC

or

ALC ->{initialization} * ->[UNI] ->{initialization} * ->{CAL}+
->builtin's ->RET

(4) Query:

STA->{initialization} *->{CAL}+ ->FIN.

where * denotes zero or more occurrence, + means at least one occurrence and [] indicates optional. The symbol “->” represents “followed by”. The meaning of LVM instructions {PCD, UNI, CCL, CAL, LAC, RET, FIN, STA} are described in Appendix A. The SCI object initialization will be discussed in Chapter 4. The variable and stub initialization are specified by LVM instructions:

IVn starting_position

 for variables

and

ITn starting_position code_entry_address
--

 for stubs. In the following,

each translation format is discussed.

4.5.1 Fact

For a fact clause, if there is a constructor or dual in its head arguments, then a V-frame must be allocated, some of the stack variables and copied list instances as well as stubs should be initialized. The delay of the initialization depends on the input modes of the arguments. If the fact only has ground arguments, the translation format degenerates to a format: UNI -> PCD. Furthermore, if the fact is a proposition, it has no unification code, the control instruction code degenerates to a single instruction PCD. In the following, some fact definitions and their corresponding LVM code are shown.

(1) A proposition:

p.

The LVM bytecode is

p/0: PCD

(2) A fact with constant arguments:

q(a,b).

The LVM code is:

q/2: UNI 2 q/2.u.0
PCD
q/2.u.0: CON a
CON b

Here label q/2 is the control code entry and q/2.u.0 is the unification code entry of head.

(3) A fact without stack variables:

r(X,X,Y).

After variable analysis, it has the form of

r(R1,R1,R0).

The LVM code is:

r/3: UNI 3 r/3.u.0
PCD
r/3.u.0: RGV 1
RGL 1
RGV 0

Since there are no stubs, SCI objects and stack variables, a V-frame is not needed.

(4) A fact with constructor:

:- mode u(-,?).
u(f(X),Y).

After clause analysis, we know that variable X is a stack variable with index of 1, the fact has the form of

u(SSI:0, R0).

The stub must be allocated in the stack frame via initialization instruction: IT1, also the stack variable X must be initialized via initialization instruction: IV1. The LVM code of the fact is shown as follows:

```
-----  
u/2:  ALV  2  
      UNI  2    u/2.u.0  
      IT1  0    u/2.s.1    // initialize a stub at position 0  
      IV1  1                    // initialize variable X  
      PCD
```

```

u/2.u.0: SSI 0
          RGV 0
u/2.s.1: FUN f/1           // static shared structure
          VAL 1
-----

```

4.5.2 Chain Call

For a chain call, if the callee is a deterministic predicate, registers (R0..R7) can be allocated to pass some parameters, otherwise stack variables and R0 register must be used to pass all parameters. The LVM uses instruction CCL to invoke the goal call procedure.

Example 4.7 To illustrate these cases, let's check two examples by assuming that goal q/1 is a deterministic predicate, and app/3 is a nondeterministic predicate.

(1) A chain call with a deterministic goal call:

```

:-mode p(-,-).
p(X,Y):-q(f(X),Y).

```

After clause analysis, the clause becomes:

```

p(V1,R1):-q(SSI:0,R1).

```

The R1 register is used to pass parameter {Y}. A V-frame with one stub and one variable {X} has to be allocated. The stub at position 0 needs to be initialized. The LVMC will generate the following code:

```

-----
p/1:  ALV  2
      IT1  0    p/1.s.1
      UNI  1    p/1.u.0
      CCL  p/1.u.1 q/1
      p/1.u.0:  VAR 1
                RGV 1
      p/1.u.1:  SSI 0
                RGL 1
      p/1.s.1:  FUN f/1
                VAL 1
-----

```

(2) A recursive chain call:

```

:-mode app(-,-,+).
app(L1,[X|L2],[X|L3]):-app(L1,L2,L3).

```

The clause is first transformed to the form of

```
app(V2,[V0|V1],[V0|R0]):-app(V2,V1,R0).
```

Here R0 register is used to pass a variable in the last argument. All variables in this clause are classified as stack variables plus R0 register variable. A V-frame with 3 cells needed to be allocated in the LVM stack area. The LVM code is:

```
-----
app/3: ALV 3
        UNI 3      app/3.u.0
        CCL R0     app/3.u.1 app/3

app/3.u.0: VAR 2
           SCI 0           // a SCI object
           DCI app/3.s.1
app/3.u.1: VAL 2
           VAL 1
           RGL 0
app/3.s.1: VAR 0
           RGV 0
-----
```

4.5.3 Rule

For a rule clause, a control frame containing three local environment parameters needs to be kept on the LVM stack, so a C-frame must be allocated. A rule clause can't use the register variables, except R0, hence all parameters are passed via stack variables.

If the last goal is a user defined goal, then an instruction: LAC is used to issue the last call. The LAC instruction does not implement the so called last call optimization [10].

Example 4.7 Suppose that we have the following clause and t/2 is an user defined predicate,

```
p( X,Y,Z):-q(U,V,W),r(Y,Z,U),s(U,W),t(X,V).
```

From the clause analysis, we know that no parameter can pass by registers (R0..R7), so all six parameters are allocated inside the C-frame as stack variables. Therefore, totally nine cells are needed for the C-frame. The LVM code is shown as follows:

```
-----
p/3:   ALC 9
        UNI 3      p/3.u.0
        CAL p/3.u.1 q/3
        CAL p/3.u.2 r/3
        CAL p/3.u.3 s/2
        LAC p/3.u.4 t/2
-----
```

```

p/3.u.0: VAR 3
          VAR 4
          VAR 5
p/3.u.1: VAR 6
          VAR 7
          VAR 8
p/3.u.2: VAL 4
          VAL 5
          VAL 6
p/3.u.3: VAL 6
          VAL 8
p/3.u.4: VAL 3
          VAL 7

```

If the last goal is a built-in predicate or an arithmetic computation, then an alternative translation format is applied. The instruction:RET is used to return the control to its parent continuation point. Let's demonstrate this case with the second clause of predicate length/2:

```

:-mode length(-,+)
length(0,[]).
length(N,[_|L]):-length(N1,L),N is N1+1.

```

In this clause, a built-in arithmetic predicate "is" is used as the last goal. Also this predicate is deterministic by last argument dispatching. After the clause analysis, the second clause becomes:

```

length(V3,[VOID|R0]) :-length(V4,R0),V3 is V4+1.

```

The generated code is:

```

length/2.2: ALC      5
            UNI      2      length/2.2.u.0
            CAL      R0      length/2.2.u.1      length/2
            LOD     4  8
                                     // move V4 to R8
            INC      8
            STI     8  3
                                     // load R8 to V3
            RET
length/2.2.u.0:      VAR  3
                   DCI  length/2.2.s.1
length/2.2.u.1:      VAR  4
                   RGL  0
length/2.2.s.1:      VOID
                   RGV  0

```

4.5.4 Query

For a query, the LVMC assumes that its predicate name is "main" without head arguments by default. All variables must be stack variables. A special frame is allocated to store these variables by the instruction "STA n", where n is the number of variables. A special instruction: FIN signals the LVM system to exit.

The LVMC handles queries with a single goal, and also queries with multiple goals. Two examples are given as follows:

Example 4.8 A query with one goal call:

?- p(Y,Z,123).

The LVMC transforms the query into the form:

main:- p(V0,V1,INT:123).

Therefore, the LVM code is:

```
-----  
main: STA 2  
      CAL main.u.0    p/3  
      FIN  
main.u.0: VAR 0  
          VAR 1  
          INT 123  
-----
```

Example 4.9 A query with multiple goals:

?- p(X),q(f(Y)),r(X,Y).

After clause analysis, the query becomes:

main:-p(V2),q(SSI:0),r(V2,V1).

where the stub must be initialized. The LVM code is:

```
-----  
main: STA 3  
      IT1 0    main.s.1    //stub initialization  
      CAL main.u.0    p/1  
      CAL main.u.1    q/1  
      CAL main.u.2    r/2  
      FIN  
-----
```

```
main.u.0: VAR 2
main.u.1: SSI 0
main.u.2: VAL 2
          VAR 1
main.s.1: FUN 0/1           // static shared structure
          VAL 2
```

Chapter 5. Built-in Predicates, Arithmetic Expressions and Initializations

In this chapter, the code generation of a clause involving built-in predicates and arithmetic expressions is discussed. Also, the initialization of variables, stubs and especially SCI objects is described.

5.1 Built-in Predicates

The built-in predicates are defined in Appendix B (also in [19]). There are nine classes of built-in predicates in Prolog. They are:

- (1) unification, e.g., =, \=
- (2) arithmetic comparison, e.g., =:=, =\=, <, =<, > and >=,
- (3) term test, e.g., atom(X), integer(X) and compound(X),
- (4) term comparison, e.g., @>, @>=, @<, @<=, == and \==,
- (5) term manipulation, e.g., functor/3, arg/3 and copy_term/2,
- (6) all solutions, e.g., findall/3, bagof/3 and setof/3,
- (7) character-string operations, e.g., atom_codes/2 and char_code/2,
- (8) I/O operations, e.g., get_char/1, read/1 and open/3,
- (9) miscellaneous, e.g., halt/1, throw/1 and repeat/0.

Although the implementation of these built-in predicates in current LVM has not been completed, the corresponding LVM instructions calling these built-in predicates are specified in Appendix B. During compilation, the LVMC loads these predicates into a special predicate table. The convention for calling a built-in predicate is that arguments passing to a built-in must be through the registers in order of (R1..R7).

Example 5.1 For example, if a clause calling integer/1 is defined as follows:

`p(X):-integer(X), q(X).`

the LVM code for this clause is:

```
-----  
p/1: UNT 1 p/1.u.0  
      B24  
      CCL R0 p/1.u.1 q/1  
  
p/1.u.0: RGV 1
```

p/1.u.1: RGL 1

where the built-in predicate integer/1 accepts the argument from R1 register.

5.2 Cut

The cut (!) predicate is a built-in predicate (labeled as B84) which is used to trim the search tree. The operational semantics of cut (!) is that when it is passed, the execution control can not go back to any other alternative clause in the current predicate and any preceding body goal in the current clause. Therefore in the view of implementation, all choice frames that were created after the choice frame corresponding to the current predicate should be discarded.

There are two types of cuts in Prolog: neck and deep cuts. A neck cut is the cut as the first body goal in the clause, i.e.,

```
head :- !, body_1, body_2,...body_n.
```

while any other clause with cut inside its body goal list is a deep cut case, i.e.,

```
head :- body_1, ...,!,...body_n.
```

The LVM uses the instruction NCT for the neck cut and two instructions LCT and DCT for the deep cut. As shown in Figure 1.9c, BB register is used as the choice frame pointer. In the neck cut case, the NCT instruction will remove the current choice frame created by the current predicate via the TRY instruction, BB is reset to point to its preceding choice frame. The LVMC only generates NCT code for neck cut when the current predicate is nondeterministic.

Example 5.2 Suppose that the first clause of a nondeterministic predicate p/1 is:

```
p(X) :- !, b(X).
```

A choice frame (B-frame) is allocated for first calling this predicate via instruction TRY,

```
p/1.1: TRY p/1.2           // p/1.2 is code entry of next clause
```

the other code for this clause is:

```
UNI 1    p/1.1.u.0
NCT
CCL R0   p/1.1.u.1   b/1
p/1.1.u.0: RGV 0
p/1.1.u.1: RGL 0
```

The deep cut is little complex. All choice frames between a choice frame upon calling this clause's predicate and this deep cut may be discarded. The LVM defines two instructions to handle this case. The first instruction

LCT n

is to save the current BB register to stack variable n, so that an extra cell must be allocated for this special variable (BB register) in the current C-frame, and also the LCT instruction must be located before the invocation of any body goal. The second instruction

DCT n

restores the BB register from the saved stack variable n to trim the choice frame linked list.

Example 5.3 Assume that we have the following clause in a nondeterministic predicate p/1:

p(X) :- b(X),!,c(X).

the LVM code generated by LVMC is:

```

-----
p/1:  ALC 5
      UNI 1    p/1.u.0
      LCT 4                // save BB register into stack cell (V4)
      CAL V3  p/1.u.1  b/1
      DCT 4                // restore BB value from V4
      LAC V3  p/1.u.1  c/1
p/1.u.0: VAR 3
p/1.u.1: RGL 0
-----

```

where two stack cells (V3 and V4) are allocated. The first cell (V3) is used by the stack variable X, the second cell (V4) is used to save the BB value of current predicate p/1.

Except the roles as neck cut and deep cut in nondeterministic predicates, cut (!) may be used as a guarded condition in deterministic predicates. The LVMC explores the new functionality of cut by replacing the cut with a guarded condition in order to make the determinism explicit.

Example 5.4 For example, we have the following predicates:

```

-----
:-mode p(+,+).

```

$p(X,X):-!$.
 $p(X,Y):-q(X,Y)$.
 $:-\text{mode fat}(+,+,-)$.
 $\text{fat}(X,Y,Z):-X<Y,! ,q(X,Z)$.
 $\text{fat}(X,Y,Z):-q(Y,Z)$.

The LVMC rewrites them in the following form:

$p(X,Y):-X=Y$.
 $p(X,Y):-X \neq Y,q(X,Y)$.

 $\text{fat}(X,Y,Z):-X<Y,q(X,Z)$.
 $\text{fat}(X,Y,Z):-X \geq Y,q(Y,Z)$.

The neck cut in predicate $p/2$ and deep cut in predicate $\text{fat}/3$ is removed, but the semantics of these predicates remains unchanged during the compilation. The guarded conditions $\{X=Y, X \neq Y$ and $X \geq Y\}$ are inserted into the clauses.

5.3 Arithmetic Expressions

An arithmetic expression is a term built from numbers, variables, and the following functors of two types:

(1) arithmetic computation,

$X+Y$	addition
$X-Y$	subtraction
$X*Y$	multiplication
X/Y	division
$X//Y$	integer division
$X \bmod Y$	modulo
$X \gg Y$	bit shift right
$X \ll Y$	bit shift left
$X \wedge Y$	bitwise and
$X \vee Y$	bitwise or
$X \hat{\ } Y$	bitwise exclusive or
$-X$	sign reversal
$X \text{ is } 1+2$	is/2 predicate

(2) arithmetic comparison

X:=Y	equal to
X =Y	not equal to
X > Y	greater than
X>=Y	greater than or equal to
X < Y	less than
X<=Y	less than or equal to

The current LVMC explicitly generates LVM code for any integer arithmetic expression defined as above without referencing the LVM built-in instructions (B03-B09).

5.3.1 Arithmetic Operations

The current LVM defines a set of arithmetic instructions for the integer arithmetic expressions, which are listed in Table 5.1. There are three features of these definitions:

- (1) All arithmetic operations are register actions;
- (2) A special hard-register, namely R8, is used to implicitly store the result after an arithmetic operation except "DEC and INC" instructions. It should be noted that R0 is not allowed in arithmetic calculation, since the main purpose of R0 is for last argument dispatching.
- (c) Any register of R9...R63 can be used to store the intermediate result of any arithmetic expression.

Operator	Operand1	Operand2	Meaning
ADD	i	j	(Ri)+(Rj)->R8
SUB	i	j	(Ri)-(Rj)->R8
MUL	i	j	(Ri)*(Rj)->R8
DIV	i	j	(Ri)/(Rj)->R8
MOD	i	j	(Ri) mod (Rj)->R8
SFL	i	j	(Ri)<<(Rj)->R8
SFR	i	j	(Ri)>>(Rj)->R8
AND	i	j	(Ri) & (Rj)->R8
ORR	i	j	(Ri) (Rj)->R8
XOR	i	j	(Ri) ^ (Rj)->R8
INC	i		(Ri)+1->Ri
DEC	i		(Ri)-1->Ri

Table 5.1 Integer arithmetic instructions

For arithmetic computation, all static or dynamic objects are loaded into registers before the calculation begins, and the result is stored after the calculation. For this, the LVM defines a set of load/store instructions for the data movement as listed in Table 5.2.

Operator	Operand1	Operand2	Meaning
CLD	n/c	i	load integer(n) or constant (c) to Ri
LOD	v	i	load VAL variable v to Ri
LOI	v	i	load VAR variable v to Ri
MOV	i	j	move RGL Ri value to Rj
MOI	i	j	move RGV Ri value to Rj
STO	i	v	store Ri value to VAR variable v
STI	i	v	store Ri value to VAL variable v
MDC	v	i	create <dc,addr(v)> to Ri
MDS	v	i	create <ds, addr(v)> to Ri

Table 5.2 Data load/store instructions

5.3.2 Register Allocation

In the LVM, a total of 64 registers are simulated by physical memory addresses, and registers (R8..R63) are reserved for the storage of intermediate results of arithmetic computation. Although theoretically we need infinite registers to store the intermediate result for an arithmetic expression in Prolog, experiments showed that expression calculations are not so complex to require five registers for intermediate results. Therefore, many compilers allocate only four registers for expression evaluation. Since 56 registers could be used in the LVM, the LVMC does not implement the register spill.

To optimize the register allocation and generate the register code of an arithmetic expression, the LVMC uses a virtual stack to simulate the execution of arithmetic computation. The main strategies are as follows:

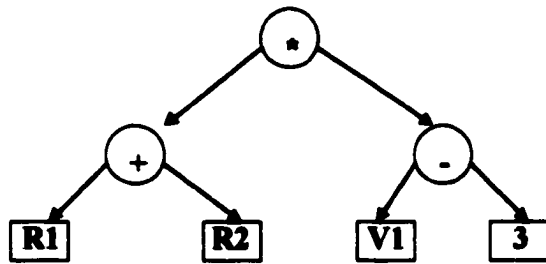
- (1) set register[8..63]=1 as a register pool. When a register (i) is used in the stack, then set register[i]=0.
- (2) push two items from the expression tree into the stack in postorder. If these two items are not register variables, use LVM load instruction to load them into unused registers. If register R8 has been used in the stack, move the content of R8 into an unused register.
- (3) pop two items from stack for an arithmetic operator and then place R8 on the top of stack.
- (4) only one item, R8 should be left on the stack after the completion of calculation.

Example 5.5 To illustrate the register allocation, assume that we have the expression tree for $(R1+R2)*(V1-3)$. The code generation procedure from the simulation execution of this expression is shown in Figure 5.1. The LVMC generates the following code for this expression:

```

-----
ADD 1 2
LOD 1 9
CLD 3 10
MOV 8 11
SUB 9 10
MOV 8 9
MUL 11 9
-----

```



expression tree

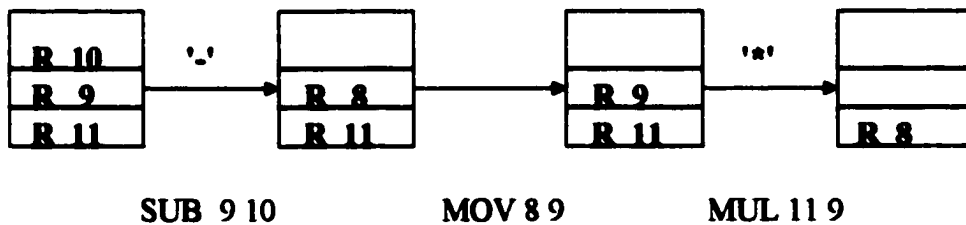
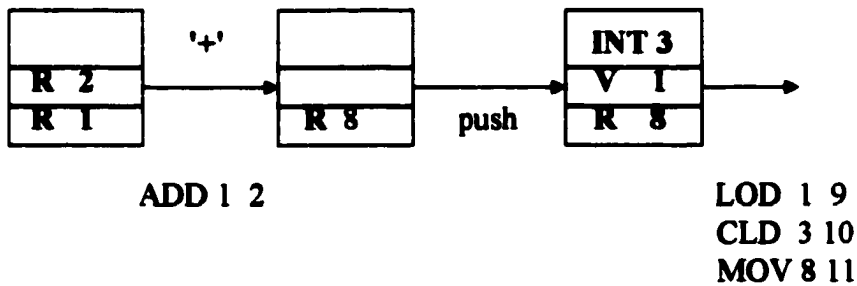


Figure 5.1 Simulation of expression evaluation

5.3.3 Arithmetic Comparison

The LVM defines five branching instructions to handle arithmetic comparison operators, which are listed in Table 5.3. In these instructions, all registers including R0 and R8 can be used as the first operand.

Operator	Operand1	Operand2	Semantics
JNE	i	e	if Ri value is negative, jump to e
JPO	i	e	if Ri value is positive, jump to e
JZE	i	e	if Ri value is zero, jump to e
JLE	i	e	if Ri value is negative or zero, jump to e
JGE	i	e	if Ri value is positive or zero, jump to e

Table 5.3 Branching instructions

The LVMC uses the following formats to generate the LVM bytecode for an arithmetic comparison:

Expression	Codes
...,R1<R2,...	SUB 1 2 JGE 8 fail
...,R1=<R2,...	SUB 1 2 JPO 8 fail
...,R1>R2,...	SUB 1 2 JLE 8 fail
...,R1>=R2,...	SUB 1 2 JNE 8 fail
...,R1:=R2,...	SUB 1 2 JPO 8 fail JNE 8 fail
...,R1=\=R2,...	SUB 1 2 JZE 8 fail

For an arithmetic expression, if one operand is an integer or constant, the integer or constant needs to be loaded into an unused register first, and then follow the above formats to generate the code. To increase the LVM speed, an extra instruction CMP is available to handle comparison of this type. Also a generic comparison instruction CMG is provided. The definition of these two instructions is listed in Table 5.4. For instance, a body goal, R1>123, in a clause is compiled into the code:

```
CMP 123 1
JGE 8 fail
```

And the goal, $R1:=R2$, can also be compiled into

```
-----
CMG 1 2
JPO 8 fail
JNE 8 fail
-----
```

Operator	Operand1	Operand2	Meaning
CMP	n/c	i	if $n/c > (Ri)$, $R8=1$; or $n/c := (Ri)$, $R8=0$; or $n/c < (Ri)$, $R8=-1$.
CMG	i	j	if $(Ri) > (Rj)$, $R8=1$; or $(Ri) := (Rj)$, $R8=0$; or $(Ri) < (Rj)$, $R8=-1$.

Table 5.4 Special comparison instructions

Except these instructions for arithmetic expressions, the LVM defines one instruction to handle term unification operation ($=$) as follows:

```
CUF i j
```

where R_i and R_j registers must be not R_8 and R_0 . If contents of R_i and R_j are unifiable, the result $(R_8) > 0$, otherwise $(R_8) \leq 0$. Therefore, LVMC uses the following two instructions to cover another unification operator: $\backslash=$.

```
-----
CUF i j
JPO 8 fail
-----
```

5.4 Initializations

The non-ground structure constructor is a new object born from invoking a clause. It may survive for a long time. The LVM allocates all potential objects before issuing a clause unification through certain initialization instructions. Since the LVM uses the SC method to represent list constructors and the PS method to all other constructors, the LVMC needs to generate the following types of initialization instruction code:

- (1) unbound stack variable initialization
- (2) stub initialization
- (3) SCI object initialization

5.4.1 Stub and Unbound Stack Variable Initialization

For non-ground and non-list structure constructors, the LVMC generates a static code segment for the skeleton of the structure, but the dynamic environment (variables) is allocated on the LVM stack frame. To associate the shared static code to an instance of structure, the corresponding stub on the frame must be specified through an assignment instruction, e.g., IT1, IT2, IT3 and ITN. Also the variables inside the structure should be set to unbound via initialization if they do not have first occurrence, and they are indexed by their C-offsets. The set of stub and variable initialization instructions are listed in Table 5.5. Normally, these initializations are carried out before the head unification inside control code of a clause.

opcode	operand	meaning
IV1	i	set the i-th cell unbound
IV2	i	set the i-th and (i+1)-th cells unbound
IV3	i	set three successive cells unbound
IVN	i n	set n successive cells unbound
IT1	i e	assign e to the i-th cell
IT2	i e1 e2	assign e1, e2 to the i-, (i+1)-th cells
IT3	i e1 e2 e3	assign e1, e2, e3 to the i-, (i+1)-, (i+2)-th cells
ITN	i e	assign a table of stubs (e) to the cells starting from the i-th cell

Table 5.5 Variable and stub initialization instructions

Example 5.6 As an example, let us consider a fact clause:

```

-----
:- mode p(-,+).
p(f(Y,g(a,X),X).
-----

```

Among the head arguments, there is a non-ground structure constructor $f/2$, which has to be flattened. After clause analysis, all object allocation on the stack V-frame is shown in Table 5.6.

Location	Object	S-offset	C-offset
0	$C1=f(Y,C2)$	0	Y: 2
1	$C2=g(a,X)$	1	X: 2
2	Y	2	
3	X	3	

Table 5.6 Object allocation for $p(+,-)/2$ clause

For this clause, two stubs {C1, C2} and one variable Y must be initialized. The variable X has its first occurrence as the first argument of $p(..)/2$. The control and unification code segments of the clause are listed as follows:

```

-----
p/2.s.1: FUN f/2
        VAL 2           //Y
        SSI 1           // C2 stub
p/2.s.2: FUN g/2
        CON a
        VAL 2           //X

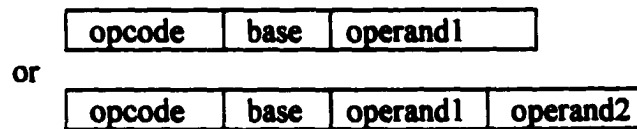
p/2:   ALV 4           // control codes
        IT2 0 p/2.s.1 p/2.s.2 // initialize two stubs
        IV1 2           // initialize Y
        UNI 2 p/2.u. 0
        PCD

p/2.u.0: SSI 0         //unification codes
        VAR 3
-----

```

5.4.2 SCI Object Initialization

In the LVM, there is no static code for non-ground list constructors. The non-ground list object is dynamically created by initialization instructions. Since there are 41 possible types of list [H|T], each type of list has a corresponding initialization instruction, all of these instructions are listed in Table 5.7. The definitions of list initialization instructions are shown in Appendix A2. They have the formats of



Here base is the starting location of the list constructor in the stack frame.

[H T]	ssi	sci	dci	cn	nil	u var	e var	f var
ssi	-	ISL	ISG	-	ISN	ISU	ISE	ISF
sci	-	ILL	ILG	-	ILN	ILU	ILE	ILF
dci	-	IGL	x	x	x	IGU	IGE	IGF
cn	-	ICL	x	x	x	ICU	ICE	ICF
nil	-	INL	x	x	x	INU	INE	INF
u var	-	IUL	IUG	-	IUN	IUU	IUE	IUF
e var	-	IEL	IEG	-	IEN	IEU	IEE	IEF
f var	-	IFL	IFG	-	IFN	IFU	IFE	-

Table 5.7 Opcode matrix of list initialization instructions

In Table 5.7, the head (H) and tail (T) of a list [H|T] may be an element of the following types :

```

-----
ssi:   a static shared instance
sci:   a static copied list
dci:   a ground copied list
cn:    a constant or an integer
nil:   a null list
u_var: an unbound variable
e_var: an equated variable
f_var: a first-occurred variable
-----

```

The symbol "x" means that the LVM has treated them as a selector, not as a constructor, and the symbol "-" denotes a case of Prolog syntax prohibition.

It should mention that in the ILL initialization instruction:

```

ILL I1 I2

```

I1 indicates the starting position of this list initialization in LVM stack frame and I2 is head initialization location. This list initialization instruction must obey certain order by setting the {I1, I2} values properly. The flattening algorithm of LVMC guarantees that this order is set appropriately.

Example 5.7 Suppose that we have the following fact clause p/2:

```

-----
:- mode p(-,-)
p([X|Y],[f(X,Y),a]).
-----

```

After flattening the arguments, we have the following objects:

```

C1=[X|Y]
C2=f(X,Y)
C3=[C2| S1]
S1=[a|[]]

```

The C1 is a list constructor of IEE type, and C3 is a list constructor of ISG type. But the C2 is a non-ground structure constructor. The S1 is a ground list selector. The stack cell allocation is shown in Table 5.8. The stub {C3} must be allocated before the list constructors {C1,C2}. Each list constructor occupies two successive cells. Also variables {X,Y} must be set to unbound by initialization. Therefore, a total of seven cells need to be reserved by the compiler.

Location	Object	S-offset	C-offset
0	C3=f(X,Y)	0	X:5 Y:6
1	C1=[X Y]	1	X: 5 Y:6
2		2	
3	C2=[C3 S1]	3	
4		4	
5	X	5	
6	Y	6	

Table 5.8 Memory allocation for p(-,-)/2

The unification code segments and initialization instructions are listed as follows:

```

-----
p/2.u.0: SCI 1 //C1
          SCI 3 //C2

p/2.s.1: FUN f/2
          VAL 5 //X
          VAL 6 //Y

p/2.s.2: CON a
          NIL

```

initializations:

```

IV1 0 p/2.s.1 // stub C3 initialization
IEE 1 5 6 // SCI C1 initialization
ISG 3 0 p/2.s.2 // SCI C2 initialization
IV2 5 // (X,Y) initialization

```

Here, variables {X,Y} are shared by two types of constructors {SSI:0 and SCI:1}. The instruction:

```
IEE 1 5 6
```

indicates that the list is initialized at cell of location = 1, its head is an equated variable (V5) and the tail is another equated variable (V6). The instruction:

```
ISG 3 0 p/2.s.2
```

denotes to initialize the SCI at third cell with a head, which is a shared object at cell of 0 and a tail, which is a ground list with the code entry: p/2.s.2.

Chapter 6. Predicate Determinacy Analysis and Indexing

Determinacy analysis of predicates is very important for the LVMC to generate optimized control codes. Currently, the LVMC defines four levels of determinacy:

- (1) Determinacy-0: A predicate consists of a single clause.
- (2) Determinacy-1: The clauses are type/value mutual-exclusive.
- (3) Determinacy-2: The clauses are conditional mutual-exclusive.
- (4) Determinacy-3: The predicate is nondeterministic.

A predicate is deterministic if it has Determinacy-0/1/2; otherwise it is nondeterministic. Only for a nondeterministic predicate, WAM-like choice frame (B-frame) is allocated, and the control sequence of *try*, *retry* and *trust* is used.

The determinacy of a predicate could be declared using the following declaration:

```
:-determinacy atom/arity (integer_constant).
```

In order to verify the declaration and determine the best determinacy level, the LVMC still does the determinacy analysis for the predicate. In this chapter, we will discuss the predicates with Determinacy-1/2/3, respectively.

6.1 Last Argument Dispatching

As mentioned in Chapter 1, since the LVM implements unification in the reverse depth-first order, the LVMC reorders the arguments of heads and goals of clause in all predicates according to their input modes regardless of their original order in its earlier phase. An argument of head in a clause, which is chosen as the last arguments of head, must have (+) or (++) input mode. If two or more head arguments at different positions in a predicate with multiple clauses have (+) or (++) input mode, then the most dispersed argument in term of its type or value is chosen as the last argument. Naturally, the clause head's last argument is the indexing key of the predicate in such a case.

The LVMC checks if the types or values of last arguments of clauses in a predicate are mutually exclusive. If it is true, the determinacy level of this predicate is Determinacy-1 and the predicate is called last argument dispatchable. Here, the type of an argument is one of {constant/integer, variable, null list, structure term}, the value of an argument means character constant and integer number. Suppose that a predicate is defined by two clauses, which accept an argument of [] and [X|Y] respectively, and the input mode to this argument is (+), this argument is mutually type exclusive. This case happens in many practical predicates.

During the compilation, two indexing methods are applied to last argument dispatchable predicates. The first dispatching method is performed according to the last argument types. The second dispatching method is used to discriminate different constant or integer values.

6.2.1 Switch Table

If the last arguments of a callee (predicate) are type dispatchable, the callee will check the type of last argument of the caller and branch the caller to a correct clause entry. The LVMC implements this dispatching technique by using the LVM instruction:

```
SWT e1 e2 e3 e4
```

to switch on a switch table. Here “e1 e2 e3 e 4” are code entries for the last argument type of callee to be variable, structure, null list and constant/integer, respectively.

Since the LVM defines a special register R0 for the fast dispatching, the instruction SWT uses the content of R0 to carry out the switching. Therefore, if the callee (predicate) is last argument type dispatchable, the LVMC will enforce that the caller (goal call) passes its last argument into R0, then the call invocation instructions such as CAL, CCL and LAC use the first operand to mark the R0, and the last unification instruction in this call unification code segment is replaced by instruction:

```
RGL 0
```

.

Example 6.1 To illustrate the use of switch table, let's check the following predicate nrev/2, where predicate append/3 is last argument dispatchable.

```
-----
:-mode nrev(-,+).

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

After clause analysis, the predicate is transformed to

```
nrev(NIL,NIL).
nrev(V5,[V3|R0]):-nrev(V6,R0),append(V5,[V3|NIL],V6).
```

The control code of predicate nrev/2 is

```
nrev/2: SWT fail nrev/2.2 nrev/2.1 fail
```

where nrev/2.1 and nrev/2.2 are control code entries of two clauses, respectively.

```
nrev/2.1: UNI 1 nrev/2.1.u.0
PCD
```

```

nrev/2.1.u.0:  NIL

nrev/2.2:     ALC 7
              UNI 2 nrev/2.2.u.0
              IFN 3                      // SCI initialization
              CAL R0 nrev/2.2.u.1 nerv/2
              LAC V6 nrev/2.2.u.2 append/3

nrev/2.2.u.0: VAR 5
              DCI nrev/2.2.u.3

nrev/2.2.u.1: VAR 6
              RGL 0

nrev/2.2.u.2: VAL 5
              SCI 3
              RGL 0                      // replace VAL 6

nrev/2.2.u.3: VAR 3
              RGV 0

```

In the control code segment of clause `nrev/2.2`, `V6` is marked in the goal `append/3` invocation instruction: “`LAC V6 ..`”, and the unification instruction: “`VAL 6`” is replaced by instruction: “`RGL 0`” in the unification code segment of `nrev/2/2.u.2`.

6.2.2 Hash Table

If the last arguments of a callee (predicate) are value dispatchable, the callee will hash the value of last argument of caller and branch the caller to a correct clause entry. The LVMC implements this dispatching technique by using the LVM instruction:

SHS hash_table_entry

to hash on a hash table. The hash table is defined by a macro of `THS`:

hash_table_entry: THS n con/int-1 e1 con/int-2 e2 .. con/int-n en

where “`con/int-i`” is the value and “`ei`” is the corresponding the code entry of clause.

In the same way as the instruction `SWT`, `SHS` uses the content of `R0` to carry out the switching. Therefore, if the callee (predicate) is last argument value dispatchable, the LVMC will arrange that the caller (goal call) passes its last argument into `R0`, then the call invocation instructions such as `CAL`, `CCL` and `LAC` use the first operand to mark the `R0`, and the last unification instruction in this call unification code segment is replaced by instruction:

RGL 0

.

Example 6.2 Let’s examine the following predicate `ftable/2`:

```
-----  
:- mode ftable(?,++).
```

```
ftable(a,c).  
ftable(a,b).  
ftable(e,f)  
-----
```

The control code of predicate `ftable/2` is:

```
ftable/2: SHS ftable_hash
```

where `ftable_hash` specifies the hash table as follows:

```
ftable_hash: THS 3 c ftable/2.1 b ftable/2.2 f ftable/2.3
```

Again, `ftable/2.1`, `ftable/2.2` and `ftable/2.3` are control code entries of three fact clauses, respectively, as shown below:

```
-----  
ftable/2.1: UNI 1 ftable/2.1.u.0  
          PCD  
ftable/2.1.u.0: CON a
```

```
ftable/2.2: UNI 1 ftable/2.2.u.0  
          PCD  
ftable/2.2.u.0: CON a
```

```
ftable/2.3: UNI 1 ftable/2.3.u.0  
          PCD  
ftable/2.3.u.0: CON e  
-----
```

6.2 Guarded Dispatching

A predicate may have explicit conditions as the first body goals of its clauses or include implicit conditions in its head arguments. If these conditions are mutually exclusive, then the predicate has a determinacy level of Determinacy-2. The LVMC implements the guarded dispatching method to differentiate between the clauses of such kind of predicates.

6.2.1 Guarded Conditions

The current LVMC exploits five types of guarded conditions in a predicate:

(1) The last argument of clause head is a variable with input mode (+), and the first goal of the clause defines the condition on data type of the variable.

Example 6.3:

```

-----
:-mode p(+,+).

p(...,X):- constant(x), ...
p(...,X):- integer(X),...
p(...,X):- compound(X),...
-----

```

(2) The last argument of clause head is a variable with input mode (+), and the first goal of the clause is an arithmetic comparison of this single variable with a constant or an integer.

Example 6.4:

```

-----
:- mode p(+,+).

p(X,N) :- N>0,...
p(1,0).
-----

```

The first clause contains an explicit condition ($N > 0$) and the second clause has an implicit condition ($N = 0$). The LVMC rewrites the second clause in the following form:

```

p(1,N):- N=0.

```

(3) The guarded condition involves a single variable argument implicitly with cut (!) as its first goal of a clause.

Example 6.5:

```

-----
:-mode q(+,+).

q(1,0):- !.
q(X,N):- p(..),...
-----

```

The cut (!) in the first clause implicates the $N = 0$ condition for the second clause. The LVMC rewrites the predicate into the form:

```

q(1,N):-N=0.
q(X,N):-N=0,p(..),...

```

(4) The guarded condition involves an explicit comparison of two head variables arguments as the first goal of a clause. The two arguments have (+) or (++) input mode.

Example 6.6:

```
-----
:-mode p(+,+).
```

```
p(X,Y):-X<Y, ...
p(X,Y):-X >Y,...
```

```
-----
```

Example 6.7:

```
-----
:-mode q(+,+).
```

```
q(X,Y):-X>Y,...
q(X,X):- p(..),...
```

```
-----
```

The predicate p/2 has two explicitly guarded conditions involving two variables (X and Y). The second clause in predicate q/2 implicitly holds the relationship: X:=Y. Therefore, the LVMC transforms this clause into:

```
q(X,Y):- X:=Y,p(..),...
```

(5) The cut (!) goal in the previous clause implicates a compliant condition for all of the following clauses and the condition involves two variables.

Example 6.8:

```
-----
:-mode r(+,+).
```

```
r(X,X):-!,p(..),...
r(X,Y):-q(..),...
```

```
-----
```

The predicate r/2 is transformed into the following form by the LVMC:

```
r(X,Y):- X = Y,p(..),...
r(X,Y):- X \= Y,q(..),....
```

6.2.2 Code Optimization

For a predicate containing guarded condition, the LVMC uses a mutually exclusive table of all comparison operators to determine if all the conditions are mutually exclusive. For the predicate with Determinacy-2, the LVMC performs guarded dispatching technique to generate the optimized control code. As shown above, the LVMC inserts in-line testing to make all implicit conditions explicit. The callee (predicate) uses the in-line test to branch the caller so that any shallow backtracking is avoided.

Actually, the implementation has three steps:

- (1) All clauses are rewritten into an unified format with the same head and also to make any implicit condition explicit so that only a single entry exists for the whole predicate,
- (2) A block of testing/branching code is created after head unification of the unified clause,
- (3) Each clause has a distinct entry for its subsequent control code of its body calls.

Example 6.9 The predicate `delete/3` is to delete an element `X` from a given list, and returns the resulting list.

```
-----
:-mode delete/3(+,-,+).

delete(X,L,[X|L]):-!
delete(X,[Y|L1],[Y|L2]):-delete(X,L1,L2).
-----
```

First, the predicate is transformed into the following form:

```
delete(R1,R2,[R3|R0]):-R1=R3,R2<=R0.
delete(R1,R2,[R3|R0]):-R1\=R3,R2<=[V0|V1],V0<=R3,
    delete(R1,V1,R0).
```

The generated code is:

```
-----
delete/3:    SWT fail delete/3.0 fail fail

delete/3.0:  UNI 3 delete/3.1.u.0
             CUF 1 3                //unifiable(r1,r2)
             JZE 8 delete/3.2

delete/3.1:  ALV 2                  // codes for second clause
             MDC 0 2                // <dc,addr(V0)>=>R2
             STO 3 0                // R3=>V0
             CCL R0 delete/3.1.u.1 delete/3

delete/3.1.u.0  RGV 1
                RGV 2
                DCI delete/3.1.u.2

delete/3.1.u.2: RGV 3
                RGV 0

delete/3.1.u.1: RGL 1
                VAL 1
                RGL 0
-----
```

```

delete/3.2: MOV 0 2           // codes for first clause
           PCD

```

The single entry of the predicate: "delete/3.0" is shared by two clauses with one unification control instruction "UNI 3 delete/3.u.0". After the unification, one test instruction: CUF 1 3 is used to branch the control to an entry either *delete/3.1* or *delete/3.2*. The two entries don't have unification instructions.

6.3 Nondeterministic Predicates

The control instructions for nondeterministic predicates are similar to those in the WAM. A choice frame (B-frame) is allocated by the TRY instruction in the first clause entry of the predicate. However, LVM does not save all registers except R0 in the choice frame, so the choice frame has a fixed size. No register variables are allowed to be used in the clauses of a nondeterministic predicate except facts. If there are more than two clauses in the predicate, the first instruction for the second clause will be "RTY e" instruction. The RTY instruction restores control environment from the B-frame and retry the next alternative entry "e". For the last clause, "TST" is the first instruction, which restores the control environment from the B-frame and discards the B-frame. Therefore, the LVMC uses the following format for the code generation of a nondeterministic predicate .

```

-----
first_clause:  TRY second_clause
               control code segment of first clause

second_clause: RTY third_clause
               control code segment of second clause

third_clause:  RTY fourth_clause
               control code segment of third clause

fourth_clause: ...

last_clause:   TST
               control code segment of last clause
-----

```

Example 6.10 The nondeterministic version of predicate *append/3* is:

```

-----
:-mode append(?,?,?).

append(L,L,[]).
append(L,[X|Y],[X|Z]):-append(L,Y,Z).
-----

```

Since the list arguments are declared to be dual, the code of this predicate must cope with different cases. During clause analysis, the predicate becomes:

```
-----  
append(R1,R1,NIL).  
append(V4,[V0|V1],[V0|V3]):-append(V4,V1,V3).  
-----
```

In the second clause, five stack cells must be allocated. The first SCI object: [X|Y] is initialized by instruction IUU. The second SCI object:[X|Z] is initialized by instruction IEU.

The LVM code for append/3 is:

```
-----  
append/3:   TRY  append/3.2  
            UNI 3  append/3.1.u.0  
            PCD  
append/3.1.u.0: RGV 1  
                RGL 1  
                NIL  
  
append/3.2:  TST  
            ALV 5  
            IUU 0  
            IEU 2 0  
            UNI 3  append/3.2.u.0  
            CCL  append/3.2.u.1 append/3  
append/3.2.u.0: VAR 4  
                SCI 0  
                SCI 2  
append/3.2.u.1: VAL 4  
                VAL 1  
                VAL 3  
-----
```

Since the predicate append/3 has only two clauses, no RTY instruction is needed.

Although some predicates are nondeterministic, the clauses of those predicates can partially be discriminated by last argument dispatching and guarded dispatching method. The LVMC actually handles it by partitioning the clause sequences of a predicate into several subsequences and analyzing the determinacy of each subsequence [10,24,25]. Therefore, a mixture of control instructions of {SWT, SHS, TRY..RTY..TST} can be seen.

Example 6.11:

```
-----  
:- mode fac(?,++).  
-----
```



```
fac(a,c).
fac(a,b).
fac(d,c).
```

The predicate `fac/2` is nondeterministic, but the values of the last argument are partially dispatchable, so the LVMC generates the following code:

```
-----
fac/2: SHS fac_hash
fac_hash: THS 2 b fac/2.1 c fac/2.2

fac/2.1: ... // for fac(a,b).

fac/2.2: TRY fac/2.2.1 // for fac(a,c).
.....
fac/2.2.1: TST // for fac(d,c).
.....
-----
```

Chapter 7. Compilation Optimization

The optimization is an important step to improve the quality of generated code. It includes initialization delay, common subexpression elimination, dead code elimination, code compression, unnecessary unification code elimination and special optimizations. Some optimizations are carried out during the preliminary code generation, other are performed on the intermediate code. In this chapter, they will be discussed in detail.

7.1 Variable, Stub and SCI Object Initialization Delay

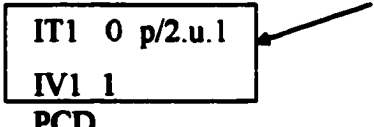
Generally speaking, the LVM allocates all possible useful objects, i.e., stack variables, stubs and SCI objects, before starting the head unification of a clause. In some cases, the initializations of these objects can be delayed until after the head unification in the control code.

Example 7.1 Suppose that we have the following fact clause,

```
-----  
:-mode p(-,?).  
  
p(f(X),a).  
-----
```

The LVMC generated code for this clause is:

```
-----  
p/2:  ALV 2  
      UNI 2 p/2.u.0  
      IT1 0 p/2.u.1  
      IV1 1  
      PCD  
        
p/2.u.0: SSI 0  
        CON a  
p/2.u.1: FUN f/1  
        VAL 1  
-----
```



In this example, the stub of “p/2.u.1” and variable X initialization is delayed until the unification control instruction “UNI 2 p/2.u.0” succeeds. If the unification fails, the execution of these two initialization instructions is skipped. However, when the input mode of the predicate is “:-mode p(?,?)”, the initializations must be done before the unification control instruction.

The LVMC uses a flag in the clause analysis to mark whether an object initialization can be delayed. The conditions for the initialization delay are:

- (1) if unbound variables only appear inside constructors with (-) mode, they can be delayed;
- (2) only non-ground constructors with (-) mode can be delayed.

7.2 Unnecessary Unification Instruction Elimination

Besides removing the dead clauses and eliminating the duplication of unification code segments of a predicate, the LVMC further eliminates unnecessary unification instructions from the head unification code segment for each clause.

Normally, the number of head unification instructions of a clause is equal to the number of head arguments. No matter whether a predicate is deterministic or nondeterministic, the last argument unification instruction of a clause could be removed if the value of the last argument is used in dispatching. In this case, the number of head unification instructions of a clause is less than the number of head arguments.

Example 7.2 The source code of predicate filter/2 is:

```
-----
:-mode filter(-,+).

filter([],[]).
filter(Z,[a|Y]):-filter(Z,Y).
-----
```

The LVMC generates the following control code for the predicate and complete code for the first clause: filter/2.1.

```
-----
filter/2:  SWT fail filter/2.2 filter/2.1 fail

filter/2.1: UNI 1 filter/2.1.u.0
           PCD
filter/2.1.u.0: NIL

filter/2.2: ...           // second clause
-----
```

The predicate filter/2 is deterministic. When the callee (predicate) uses the last argument type to dispatch the call to the null list [] entry: filter/2.1, only one unification instruction corresponding to the first head argument of this clause is needed, so the second unification instruction NIL is removed. Therefore, the unification control instruction for this clause is "UNI 1 filter/2.1.u.0".

When the instruction SHS is used to branch a call to an entry according to the value of the last argument, the number of head unification instructions of each clause may also be reduced by 1.

Example 7.3 We have an example using hashing table as follows:

```
-----
:-mode const(-,+).

const(a,1).
const(b,2).
-----
```

The LVMC generates the following control code for the predicate and complete code for the first clause: const/2.1.

```
-----
const/2: SHS const_h
const_h: THS 2 1 const/2.1 2 const/2.2

const/2.1: UNI 1 const/2.1.u.0
          PCD
const/2.1.u.0: CON a

const/2.2: ..... //second clause
-----
```

The predicate const/2 uses the R0 content to select the code entry with referencing the hash table "const_h". Once the entry point (e.g., const/2.1) is chosen, the unification instruction of unifying the last argument of the clause head, i.e., INT 1, is removed. The unification control instruction of this clause is "UNI 1 const/2.1.u.0".

7.3 Stub and Variable Initialization Instruction Compression

In order to increase the efficiency of LVM bytecode loader, the LVM defines extra six instructions for stub and unbound stack variable group initialization, except the IV1 and IT1 instructions. The instructions for stack variable group initialization are:

```
-----
IV2   i
IV3   i
IVN   i n
-----
```

where IV2 is used to initialize 2 successive unbound cells at location i and i+1, IV3 initializes 3 successive unbound cells starting from the location I, and IVN initializes n successive unbound cells starting from i-th location.

The instructions for the stubs group initialization are:

```

-----
IT2  i e1 e2
IT3  i e1 e2 e3
ITN  i e
-----

```

where IT2 is used to assign two code entries "e1" and "e2" to the i-th and (i+1)-th cells respectively, IT3 assigns three code entries to 3 successive cells starting from the location i and ITN initializes stubs starting from i-th location using "e" as the stub table.

The LVMC performs this optimization on the intermediate code. When several unbound stack variables or stubs occupying the continuous cells have to be initialized, the group initialization instructions are used instead of single variable or stub initialization instructions.

Example 7.4 The LVMC uses the right side instructions to replace the left side instructions as follows:

IV1 3 IV1 4 IV1 5	<=>	IV3 3
-------------------------	-----	-------

and

IT1 2 e1 IT1 3 e2 IT1 4 e3 IT1 5 e4	<=>	ITN 2 stub_table ... stub table: TSB 4 e1 e2 e3 e4
--	-----	--

where the macro TSB defines the stub initialization table.

7.4 Combination of Frame Allocation and Unification Instructions

In the control instruction set of LVM, two instructions {ACU and ACV} are designed to combine the frame allocation instructions {ALV and ALC} and unification instruction UNI. Their equivalent relations are displayed as follows:

ACV n1 n2 e	<=>	ALV n1 UNI n2 e
-------------	-----	--------------------

ACU n1 n2 e	<=>	ALC n1 UNI n2 e
-------------	-----	--------------------

The LVMC performs this optimization when there isn't any instruction between the ALC/ALV instruction and UNI instruction in the control code segments of clauses. For the same purpose of the group initialization, the instruction folding can reduce the decoding time of LVM bytecode emulator.

7.5 Special Optimization

7.5.1 Features of a Set of Special Predicates

Except for the above optimizations, the LVMC does a special optimization on a set of special predicates. First, let us examine the predicate member/2,

```
-----
:-mode member(+,+).

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

The function of member(X,L) is to check if X is the member of list L. This is a nondeterministic predicate with R0 dispatching.

Without optimization, the LVMC generates the following code:

```
-----
member/2: SWT fail member/2.1 fail fail           // R0-dispatching

member/2.1: TRY member/2.2
      UNI 2 member/2.1.u.0
      PCD

member/2.1.u.0: RGV 1                             // member(X,
      DCI member/2.1.u.1                          //   [X|_])
member/2.1.u.1:RGL 1
      VOID

member/2.2: TST
      ACU 4 2 member/2.2.u.0
      CCL R0 member/2.2.u.2 memeber/2 // recursive call.

member/2.2.u.0:VAR 3                             // member(X,
      DCI member/2.2.u.1                          //   [_|L])
member/2.2.u.1:VOID
      RGV 0
member/2.2.u.2:VAL 3                             // member(X,
      RGL 0                                       //   L)
-----
```

In this case, each time member/2 is called, a B-frame will be allocated via TRY instruction. If the first clause (member/2.1) fails, the B-frame is discarded by the TST instruction of the second clause (member/2.2). The second clause first creates a C-frame with one local variable (X) before the head unification, and calls member/2 recursively, therefore the B-frame must be allocated again. The B-frame allocation-deallocation loop really slows down the execution. In the imperative language like C, a pointer to a list is advanced to do the element comparison, so the execution space and time is minimal.

A set of this type of nondeterministic predicates exists. We list a few of them here:

right_of(A,B,L): to check if A is on the right side of B in list L.

:-mode right_of(+,+,+).

:-garbage right_of/3(0).

right_of(A,B,[A,B|_]).

right_of(A,B,[_L]):- right_of(A,B,L).

neighbor(A,B,L): to check if A and B are neighbor in list L.

:-mode neighbor(+,+,+).

:-garbage neighbor/3(0).

neighbor(A,B,[A,B|_]).

neighbor(A,B,[B,A|_]).

neighbor(A,B,[_L]):- neighbor(A,B,L).

subset(A,B,C,L): to check if (A,B,C) is a subset in list L.

:-mode subset(+,+,+).

:-garbage subset/3(0).

subset(A,B,C,[A,B,C|_]).

subset(A,B,C,[_L]):- subset(A,B,C,L).

They are selection and checking predicates frequently used in the Prolog application as built-in library predicates.

In summary, these predicates have the four main features:

- (1) There is no (- or ?) input mode in their head arguments.
- (2) A predicate consists of one or more facts and a recursive clause as the last clause.
- (3) The predicate is nondeterministic.
- (4) Only one argument is a list or structure and others are flat variables.

7.5.2 V/B Frame Reuse

The LVMC optimizes the LVM code for the above predicates by sharing one V-frame and one B-frame. The principles of the optimization are:

- (1) A working V-frame is allocated to store all the shared variables before the choice B-frame is allocated by TRY instruction. The V-frame is shared by all clauses of this predicate.
- (2) The choice B-frame is reused with minimum initialization with OTR instruction when the last recursive call is issued. Two entry points are set up for the first fact clause: one for the outside call and another for the recursive call.
- (3) The unification instructions with the same variable index are eliminated.

The LVMC implements this optimization by a three-step algorithm:

STEP-1: Assert a new deterministic chain clause before the predicate, which allocates an appropriate V-frame for the predicate invocation, and rename the original predicate. For example,

```
member(V0, R0):- new_member(V0,R0).
new_member(X,[X|_]).
new_member(X,[_|L]):-new_member(X,L).
```

STEP-2: Set up two entry points for the first fact clause of original predicate in the following format:

```
-----
new_member/2.1.1: SWT fail label_1 fail fail // first entry for outside call
label_1:          TRY new_member/2.2        // allocate B-frame
                  JMP label_3

new_member/2.1.2: SWT fail label_2 fail fail // second entry
label_2:          OTR new_member/2.2        // share B-frame
label_3:          control code for first clause

new_member/2.2:  OST
                  control code for last clause
-----
```

where new_member/2.1.1 is the first code entry for outside call and new_member/2.1.2 is the second entry for itself recursive call. The first entry will pass the program control flow through TRY instruction, but the second entry will pass through OTR instruction.

STEP-3: Change RTY instruction to ORT, and change TST instruction to OTS for other facts and the last clause of original predicate. Eliminate unnecessary unification instructions.

Example 7.5 The optimized code for the predicate member/2 generated by the LVMC is:

```

-----
member/2: AVU 1 2 member/2.u.0
          CCL R0 member/2.u.1 new_member/2      // first call new_member/2
member/2.u.0: VAR 0
              RGV 0
member/2.u.1: RGL 0                               // eliminate 1 unification instructions
-----

new_member/2: SWT fail label_1 fail fail
label_1:      TRY new_member/2.2
              JMP label_3

new_member/2.1: SWT fail label_2 fail fail
label_2:      OTR new_member/2.2
label_3:      UNI 1 new_member/2.1.u.0
              PCD

new_member/2.1.u.0: DCI new_member/2.1.u.1
new_member/2.1.u.1: VAL 0
                  VOID

new_member/2.2:  OTS
                  UNI 1 new_member/2.2.u.0
                  CCL R0 member/2.u.1 new_member/2.1      //second call

new_member/2.2.u.0: DCI new_member/2.2.u.2
new_member/2.2.u.2: VOID
                  RGV 0
-----

```

This special optimization has three advantages:

- (1) it saves working memory because only one V-frame is used.
- (2) it saves execution time, since one OTR instruction has 6 assignment statements less than one TRY instruction, and eliminates B-frame's allocation/deallocation.
- (3) it uses less unification instructions, because variable unifications with the same variable index are eliminated.

However, this optimization can't be generalized to predicates with constructors.

Chapter 8. Assistance to Garbage Collection

Dynamic memory management is an extremely important issue in the implementation of logic programming systems. Since Prolog is a symbolic language, it usually requires more memory than the imperative languages like C. The Prolog systems often manipulate large data structures with complex inter-dependence. Although the advances in the computer hardware technique have made memory chips more and more cheap, memory always is a precious and limited resource. During the execution of a Prolog program, a dynamic object may outlive the procedure that creates it. The dynamic objects which are not live but not freed yet, are called garbage. Therefore, an efficient garbage collector has to be designed for the success of a Prolog system.

A well-designed garbage collector should have the following features [25]:

- (1) Garbage collection (GC) must be safe. Live data must never be erroneously reclaimed.
- (2) GC should be comprehensive such that the locality of memory is taken into account.
- (3) GC should be cost-effective. The ratio of GC execution time to overall execution time should be as low as possible.
- (4) GC should minimize the pause time to facilitate the interactive system.

Since the LVM uses one stack policy, it is impossible to run practical Prolog programs without an efficient garbage collector. In this chapter, we will first survey the existing GC algorithms, and then introduce the idea of chronological GC used in the LVM. After that, the LVMC assistance to the chronological garbage collector will be described.

8.1 Survey of Garbage Collection

In the WAM, most dynamic objects are stored on a global stack (called the heap), while the choice points and environments are stored on a local stack (called the stack). A trail stack records all bindings to be undone on backtracking. The WAM saves the state of the machine on the choice point frame when a choice point is created. Using this information, stacks can be reset and storage reclaimed. The choice point frame delimits the heap into several segments. Creating a new choice point creates a new segment. Backtracking removes segments and performs the segment cutting and merging. The dynamic objects are allocated in the topmost segment of heap. Variable bindings are implemented by assigning a cell address of the heap to the variable. The bindings are recorded on the trail stack when the variable's cell is not in the topmost segment. When two variables are unified, a pointer from one cell of the heap to the other cell is created. Therefore, pointer chains may arise.

The garbage collection is done by starting at a set of root pointers, which are registers and the local stack variables, and discovering what objects on heap are reachable from these pointers. These reachable objects are live, otherwise are garbage. The memory of garbage is reclaimed.

Many different GC algorithms are available for searching dead objects and recovering the memory in Prolog. The common methods are:

(1) Reference Counting algorithm [26]: Each cell of heap has an additional field as the reference counter. The counter is updated when a pointer to this cell is created or deleted. When the counter drops to zero, the cell is garbage.

The advantages of this algorithm are: (a) GC overheads are distributed throughout the computation; (b) Performance does not degrade with heap residency. But the algorithm has two disadvantages: (a) The maintenance of the counter is expensive; (b) It can't reclaim cyclic structures.

(2) Mark-Sweep algorithm [27]: It is performed in two phases. The first phase is to mark all live cells. The second phase, called sweep, scans the heap linearly from bottom to top, and puts the dead cells onto the free list.

The advantages of this algorithm are: (a) Cycling is handled naturally. (b) No overhead is placed on pointer manipulation. But this algorithm has four drawbacks: (a) The algorithm is stop/start type, unsuitable for real-time and interactive systems. (b) The complexity of this algorithm is proportional to the size of heap. (c) It tends to fragment memory. (d) GC becomes more frequent as the heap residency of a program increases. To overcome the memory fragmentation, a compact phase is used in the following algorithm.

(3) Mark-Compact algorithm [28]: This algorithm works in three phases. The first phase is the same as the Mark-Sweep algorithm, that is, marking the live data structure. It adds two phases: compact the live cells and update the values of pointers for these moved cells.

(4) Copy algorithm [29]: The algorithm divides the heap equally into two semi-spaces, one of which contains current objects and the other obsolete objects. It traverses the live objects in the old space and copies each live cells into the new space. Then the roles of the two spaces are flipped.

The advantages of the copy algorithm are: (a) Live objects are compacted into the bottom of the new space; (b) Allocation costs are extremely low. The disadvantages are: (a) Two semi-spaces are used. (b) Performance will degrade as the residency of a program increases.

(5) Generation GC algorithms [30,31]: It relies on the observation that most newly created objects tend to be short-lived. The GC should concentrate on the young objects. Thus, the heap is split into two or more generations. When the youngest generation fills up, a collection spanning more generations is done, and the survivors move to the older generations.

Usually, the implementations use two generations delimited by some choice points. Memory space is reclaimed in the new generation only. All objects in the old generation are potentially useful. Therefore, the collection roots must include the pointers from the older to the younger generation. To avoid exhaustively searching all areas for pointers into the new generation, all cross generation bindings are recorded by carefully setting the trail condition. The trail stack is used as the write barrier. This results in some extra trailing. Also another drawback is that the write barrier may be in such a state that old generation is almost empty for some Prolog programs, then this type of generation GC fails. A solution is to create artificial choice point.

In summary, an ideal GC collector should have low CPU and space overhead, good virtual memory and cache performance, short pause time. Also, it should be ecological. The memory reclaim is a natural way of "life" of the execution system, not as a deliberate garbage collection. In the following section, it will show that an ecological GC collector is possible for Prolog under the LVM architecture.

8.2 Chronological Garbage Collector in the LVM

As well known, a Prolog procedure is defined as a set of selected clauses, where each clause can be rewritten to involve no further branching. With the input mode declaration or analysis, the number of objects created in each clause can be determined at compilation time. Under one stack policy of the LVM, all dynamic objects (short lived and long-lived) of a clause are stored in the C-frame or V-frame. The V/C frames are allocated on the LVM stack area by following the chronological order of procedure invocation. For a nondeterministic predicate, a B-frame is inserted before the V/C frame. Therefore, a natural and dynamic generation line, called C-line, is established. The C-line splits the entire LVM stack into two generations: the part above the C-line is the younger generation and the part below the C-line is the older generation. The location of C-line dynamically moves up and down as a new C/B frame is pushed on the stack.

Under the LVM architecture, there are two possible stack layouts of the C-line, which are illustrated in Figure 8.1-8.2, where AF is the current active V/C frame, BB is the most recent choice frame (B-frame), TT is the top of trail, and ST is the top of stack. In the case of BB below AF, the C-line is the top of the current active frame (AF) as shown in Figure 8.1. These variables in the current active frame are called C-variables. In the other case of BB above AF, the C-line is determined by $BB + \text{sizeof}(B\text{-frame})$.

The C-variables are most important to distinguish useful objects from the garbage inside the young generation above C-line. Since no globally scoped variables exist in Prolog and all variables are local to their clauses, the only way of passing an old variable to a young procedure is through the parameter passing mechanism. The C-variables are the unique bridges to connect the old objects to the young objects. If any dynamic object above the C-line can't be reached via these C-variables, then this object is treated as garbage. Furthermore, the LVMC can distinguish useful variables from these C-variables.

Hence, the implementation of the GC becomes simple. The GC collector collects all useful old-to-young cross generation references via these useful C-variables. The cost of this GC method is proportional to the number of useful C-variables, which is extremely low by comparison with other GC algorithms [31]. It also can be seen that this GC collector is incremental.

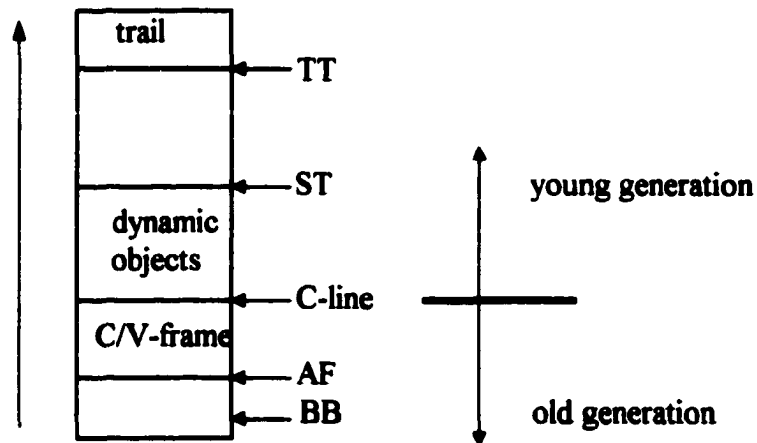


Figure 8.1 Memory layout in case of BB below AF

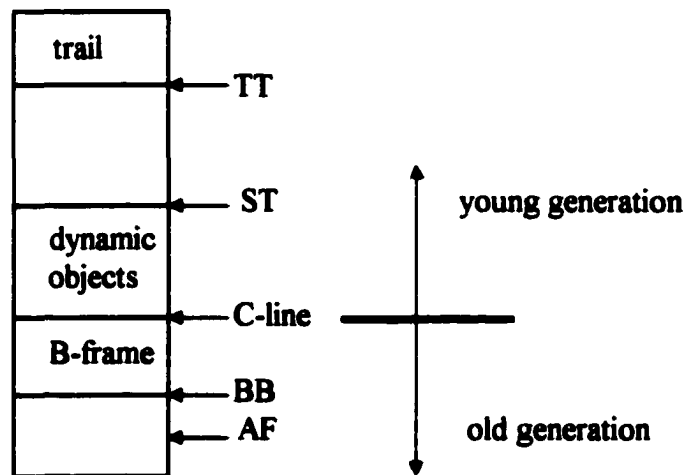


Figure 8.2 Memory layout for BB above AF

The chronological GC algorithm could be described as follows:

- (1) get the initial root from the instruction code inserted by the LVMC.
- (2) search its binding (r') of each root (r):

- (2.1) If r' is atomic, then assign $r' \rightarrow r$;
- (2.2) If r' is an unbound variable in younger generation, create a copy;
- (2.3) If r' is a SCI/SSI object, in older generation, scan the instance for new roots or in younger generation, copy the instance.
- (2.4) Skip other cases.

A stack is used in the GC algorithm implementation. More details are presented in reference [32].

8.3 GC Level Estimation

The above GC algorithm in the LVM system strongly depends on the LVMC guidance. The GC point setting is determined on the GC level estimation of predicates. Clearly, it is impossible to do quantitative analysis of garbage at compilation time. However, a qualitative estimation is possible. In the global analysis phase, the LVMC performs a qualitative analysis of garbage for each predicate. For simplicity, the current LVMC distinguishes the amount of garbage of a predicate in three levels:

- (1) Garbage-0: no garbage.
- (2) Garbage-1: linear dependence on the number of stack variables.
- (3) Garbage-2: quadratic dependence or high dependence.

For example, if a predicate only generates one list as output and uses two or more intermediate lists as its working space, then its garbage level is ranked as 2. After examining clause definitions of different types, we can find that only a deterministic and recursively defined predicate may generate a linear or high degree amount of garbage. Therefore, the garbage estimation process becomes very simple.

Example 8.1 To illustrate the GC level estimation, let's use the predicate of quicksort/3 as an example:

```

-----
:mode qsort(-,+,+).
:mode partition(+,-,+).

qsort(X,X,[]).
qsort(Z,X,[Y|L]):-partition(Y,L1,L2,L), qsort(Z1,X,L2), qsort(Z,[Y|Z1],L1).
-----

```

The `qsort/3` is deterministic predicate due to the last argument dispatching. Its first clause is a fact. A fact has no garbage. The second clause is a recursively defined rule. All variables in the rule represent lists. In the first goal, two variables {L1 and L2} simultaneously are used as its working variables, therefore, the GC level of `qsort/3` is 2.

8.4 GC Root Set Collection and GC Point Setting

In order to cope with the GC along the execution path, the LVMC must insert instruction to mark where the GC collector should be triggered and provide an initial root set to indicate which useful objects should be collected for keeping. Since the GC collector of LVM system is totally passive, the success of GC collector implementation depends on the LVMC assistance.

First, the LVM defines a macro to set up an initial root table, whose format is:

```
TGC n i1 i2 ... in
```

where n is the length of the root table, and (i_1, i_2, \dots, i_n) are the indices of some stack variables to be collected during GC.

The question is how the LVMC selects a root set from the stack variables of a clause. From the LVM's view, for a clause $H:-G_1, G_2, \dots, G_n$, only VAL-type variables in the proceeding goals $(G_{i+1} \dots G_n)$ are useful after the current goal (G_i) invocation. These VAR-type variables are uninitialized variables, which we do not collect unless they are in the head arguments. Therefore, the identification of useful objects becomes straightforward. The LVMC uses two rules to extract the root variables from a clause:

- (1) All VAL-type variables which appear proceeding the current goal must be collected.
- (2) All stack variables in head arguments must be collected.

Example 8.2 Let's use the second clause of predicate `qsort/3` as the example. After the clause analysis, the index and types of stack variables are:

```
-----  
qsort(V5, V6, [V3|R0]):-           // V3, V5, V6--VAR type  
    partition(V3, V7, V8, R0),      // V3--VAL type, V7, V8--VAR type  
    qsort(V4, V6, V8),             // V4--VAR type, V6, V8--VAL type  
    qsort(V5, [V3|V4], V7).        // V3, V4, V5, V7--VAL type  
-----
```

Since the garbage level of `qsort/3` is 2, let's insert a GC instruction after the second goal `qsort/3`. The variables $\{V3, V4, V5, V7\}$ in the last goal are of VAL-type. Therefore, after the execution of second goal call `qsort/3`, only objects reachable from these variables need to be kept for further reference. Also, since the stack variables $\{V3, V5, V6\}$ appear in the head arguments, the root set includes five variables $\{V3, V4, V5, V6, V7\}$. The variable `V8` is excluded. The root table is set up by the instruction:

```
TGC 5 3 4 5 6 7
```

where five stack variables from `V3` to `V7` are specified in the TGC table.

Secondly, the LVM defines two types of garbage collection instructions, that is,

and

```
MGC gc_entry
CGC gc_entry
```

where MGC means "must collect", CGC means "test and collect if necessary", and gc_entry gives the entry of initial root table defined by the macro TGC. For the CGC instruction, the LVM system can set a constant memory limit to determine if the collection is necessary.

The location and type of GC points depends on the GC levels of the body goals in a clause. The LVMC uses four rules to set the GC points in a clause:

- (1) The clause must be a rule with C-frame allocation.
- (2) If a goal with GC level greater than 0 has a different name from the current clause, a MGC point is set after the goal call.
- (3) If a body goal with GC level =2 is a recursive call, a CGC point is set after the call.
- (4) No GC point is set after the last call.

Example 8.3 To illustrate the GC point setting, let's continue the analysis of the predicate qsort/3. For the first clause, it is a fact and there is no stack variable. The second clause is a rule with six stack variables. After clause analysis, the qsort/3 is transformed into the form

```
-----
qsort(R1,R1,[]).
qsort(V5,V6,[V3|R0]):-partition(V3,V7,V8,R0),qsort(V4,V6,V8),qsort(V5,[V3|V4],V7).
-----
```

From the above GC setting rules, only one CGC instruction needs to be inserted after the second goal call qsort/3 of the second clause. From the root analysis of the second clause, we know that after invoking qsort(V4,V6,V8), these variables {V3,V4,V5,V6,V7} must be collected. The LVMC generates the following code for qsort/3:

```
-----
qsort/3:      SWT fail qsort/3.2 qsort/3.1 fail

qsort/3.1:    UNI 2 qsort/3.1.u.0                // first clause
              PCD

qsort/3.1.u.0: RGV 1
              RGL 1
-----
```



```

qsort/3.2:   ACU 9 3 qsort/3.2.u.0           //second clause
             CAL R0 qsort/3.2.u.1 partition/4 // call partition/4
             CAL V8 qsort/3.2.u.2 qsort/3     // call qsort/3
             CGC qsort/3.2.g.1                // set CGC point
             LAC V7 qsort/3.2.u.3 qsort/3     // last call

```

```

qsort/3.2.u.0: VAR 5
               VAR 6
               DCI qsort/3.2.s.1

```

```

qsort/3.2.u.1: VAL 3
               VAR 7
               VAR 8
               RGL 0

```

```

qsort/3.2.u.2: VAR 4
               VAL 6
               RGL 0

```

```
// V8=>R0
```

```

qsort/3.2.u.3: VAL 5
               SCI 3
               RGL 0

```

```
// V7=>R0
```

```

qsort/3.2.s.1: VAR 3
               RGV 0

```

```
qsort/3.2.g.1: TGC 5 3 4 5 6 7
```

```
// GC root set
```

In this example, an instruction: "CGC qsort/3.2.g.1" is arranged after the goal call: "CAL V8 qsort/3.2.u.2 qsort/3". The entry "qsort/3.2.g.1" specifies the initial root table.

Chapter 9. Conclusions

The first version of LVM compiler (LVMC) has been designed. About 8000 lines of C code have been written. The compilation time is approximately proportional to the program size. About 80 percent of the time is spent in the global analysis. If the source code includes mode declarations, the total compilation time could be greatly reduced. Some compiled programs have been tested under a LVM emulator. In the following sections, the performance of the LVM will be discussed, and then the main features of the current LVMC will be summarized. Finally, some future improvements on LVMC will be proposed.

9.1 LVM Performance

A Prolog program is translated into LVM bytecode by the LVM compiler. LVM bytecode programs are portable to any computer platform. There are three ways to execute these LVM bytecode programs, which are shown in Figure 9.1. The first implementation method includes an interpreter for the LVM instruction set. The interpreter sees the compiled LVM bytecode as an input stream of bytes and interprets them as LVM instructions. The inner loop of this interpreter is

```
do {  
    fetch a LVM instruction;  
    execute an action according to its opcode value;  
} while (there is more to do);
```

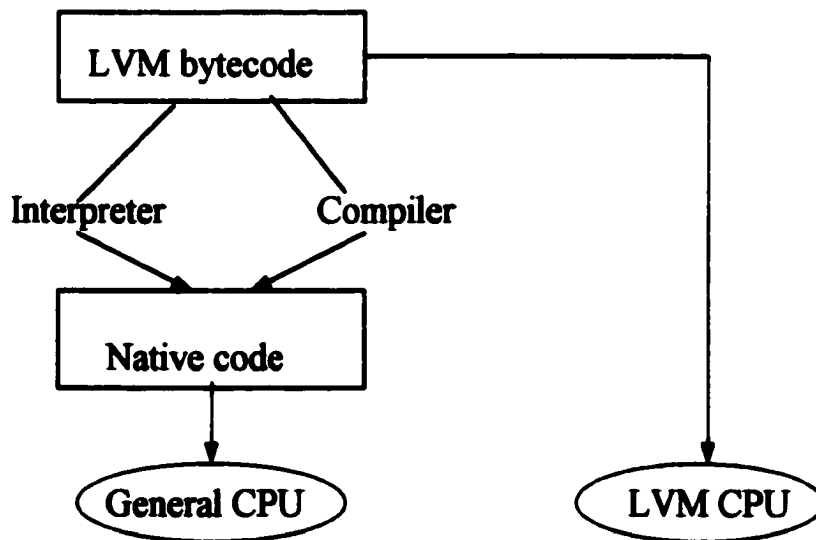


Figure 9.1 Execution model of LVM bytecode

The second method is to compile the LVM bytecode further into native machine code. This compilation is straightforward. Each LVM instruction is expanded by a set of native code. Therefore, the total size of code will be increased by several times before the loader loads the code into RAM.

Another alternative method is to rewrite a set of macro machine instructions so that the machine (LVM CPU) can identify the LVM bytecode and directly execute them on CPU. This method is costly but fastest.

At present, a LVM emulator (in C) has been implemented. A SPARC workstation with 8MB RAM and 64KB virtual cache is used to run the test. Four benchmark programs have been used to test the LVM system performance. For the "traveling salesman problem", tours of 30 cities are computed (*tsp30*). The *dna20* program implements a dynamic algorithm for comparing DNA sequences. One sequence of length 32 is compared with other 20 sequences. The *tak22* program performs a recursive arithmetic computation with input (22,16,8). In the execution of this program, no-long lived (heap) objects are involved. The *qsnv1000* is a program of *quick-sort* followed by *naive reverse* for a list of 1000 integers. This benchmark is very interesting because a fraction of collected objects may survive through many garbage collections. Table 9.1 gives the statistics of these benchmarks.

program	gc-enabled	gc-disabled
<i>tsp30</i>	9.35	10.63
<i>dna20</i>	7.23	7.45
<i>tak22</i>	22.1	22.9
<i>qsnv1000</i>	5.57	5.50

Table 9.1 Execution time (sec)

In Table 9.1, the execution time is the user elapsed times in second, measured by the UNIX timing facility. All listed values are the average values of execution time over 100 runs. The gc-disabled run is to execute a program with CGC instructions disabled. To fit the capacity of the virtual memory of the SPARC workstation, the proper inputs for these programs are chosen. From the Table 9.1, we can find that the execution times of the first three programs running under gc-enabled are shorter than those under gc-disabled. These results are beyond our usual expectation. In other words, their performance with CGC is better than, or at least as good as that when they run on a machine with infinite virtual memory. This result indicates that the memory management system in LVM is highly successful.

There are several factors to explain the excellent behavior of the LVM memory management system. First, the GC algorithm is very efficient. By contract to traditional

idea, this algorithm collects useful objects instead of garbage with respect to dynamically partitioned generations. More than 85% of all objects in young generation die within one GC cycle. Secondly, the single stack paradigm incorporated with GC collector improves the data locality during execution so that the program cache performance is improved by greatly reducing the cache misses. A detailed analysis of the cache performance during GC process is presented [32]. This study shows that the cost of GC could be paid by the saving from cache performance improvement.

For full evaluation of the performance of LVM system, the comparison between LVM system and other Prolog systems should be carried out. Unfortunately, these results are not yet available. At present, work continues on the project.

9.2 Compiler Features

There are seven main features in the LVMC design [33]:

(1) **Global analysis.** In order to generate optimized LVM code, the LVMC extracts the necessary properties from the source programs such as input mode, determinacy and garbage estimation. Alternatively, the input mode and determinacy could be declared by programmers.

(2) **Last argument dispatching.** At the implementation level, the LVM uses the last argument as the first-order discriminator for the purpose of partial unification and control transfer. There is no constraint on the programmer's programming style. The LVMC automatically reorders the clause arguments in the reference to LVM.

(3) **Determinism transformation.** During compilation, shallow backtracking is replaced by conditional branching when it is possible, and the implicitly deterministic predicates are rewritten to make its determinism explicit.

(4) **Garbage collection assistance.** The LVMC inserts some special LVM instructions into the compiled codes to trigger the garbage collector and also specifies the initial root set to guide the collector.

(5) **Uninitialized variable identification.** In order to eliminate the unnecessary dereferencing or trailing operations, the LVMC distinguishes the variables in a clause into L-type and V-type. The variables of V-type are bound by destructive assignment.

(6) **Register allocation.** To speed up the arithmetic computation and term unification of deterministic clauses, soft registers are scheduled in calculation and parameter passing. The register conflict is avoided.

(7) **Special optimization.** For a special set of nondeterministic recursive predicates, the LVMC provides two code entries (nonrecursive and recursive entry) for the purpose of stack frame reuse.

9.3 Future Work

Although the current LVM compiler implements a series of optimizations on the generated bytecode, there are many possibilities for future improvement. For example, they are:

- (1) better mode analysis;
- (2) multiple argument dispatching;
- (3) using matching tree for conditional dispatching;
- (4) garbage estimation refinement and root set minimization.

The mode analysis of program arguments plays an important role for the optimized code generation. A better algorithm of mode analysis will reduce the number of (?) mode. If (?) mode of a structure argument could be reduced to (-) mode, it would save the initialization operation of these variables of the structure. On the other hand, if (?) mode of a structure argument becomes (+) mode, the selector codes of the structure will be generated instead of constructor code. The registers may be used instead of stack variables and the code efficiency will be increased. Furthermore, the (+) mode of structure arguments can be classified into a deeper level, i.e., +/0, +/1, +/2, ... +/n modes [34]. Here (+/0) mode is designed to the ground structure. The index (n) denotes the minimum depth of variables inside the structure tree. This extra information can speed up the unification process through comparing two selector depths before full unification.

The indexing method is critical to determine whether a choice point needs to be set up in some cases. At present, only one head argument with (+) mode is used as the clause dispatching discriminator. When there is more than one selector in the head argument, multiple argument dispatching algorithm can be implemented. Classifying a program from nondeterministic predicate to deterministic can really shorten the program execution time. The LVMC explores the first body goal in a clause as the conditional dispatching discriminator, but there is still more work to do. A more sophisticated indexing technique could be applied to determine the predicate determinacy.

In order to reduce the number of GC instructions, two practical issues require further investigation: how to minimize the initial root sets of clauses and how to refine the garbage estimation of predicates. If the amount of garbage of a predicate may be classified into more than three levels, then the rules of setting GC points will be subtler.

Appendix A. LVM Instruction Set

Table A1.1 LVM instruction set (I)

modification	instructions
VAR v	an uninitialized stack variable with relative offset v
VAL v	an initialized stack variable with relative offset v
RGV r	an uninitialized register variable with numbering r
RGL r	an initialized register variable
SSI v	static shared instance with offset v relative to stub
SCI v	static copied instance with offset v relative to code entry
DCI e	direct instance with code entry e
FUNC f/n	functor f with arity n
VOID	void variable
NIL	null list
INT n	integer n
CON c	constant c
control	instructions
ALC n	allocate a n-cells of C-frame including 3 environment parameters
ALV n	allocate a n-cells of V-frame
UNI n e	unify the number of instruction (n) with code entry e
ACU n1 n2 e	equal to two successive instructions: ALC n1 and UNI n2 e
AVU n1 n2 e	ALV n1 and UNI n2 e
CAL r/v u e	call procedure (e) with head U-code (u) and dispatching flag r/v
CCL r/v u e	chain call
LAC r/v u e	last call
PCD	return to a continuation entry
SWT e1 e2 e3 e4	dispatch on the type of R0 value type (e1-var, e2-con/int, e3-nil, e4-structure)
SHS e	dispatch on (CON/INT) value of R0 with hash table e
THS n a1/e1..an/en e	marco for hashing table
CGC e	collect garbage if necessary with root set entry e
MGC e	must collect garbage with root entry e
TGC n v1 ...vn	macro for gc root set
TRY e	create a B-frame, try this else e (code entry)
RTY e	retry this else e (code entry)
TST	last alternative
OTR e	recursive try with optimization of reusing choice frame
ORT e	optimized re_try instruction
OTS	optimized last_call instruction
STA n	start execution with n-cells for query
FIN	terminate execution
SUC	succeed
FAL	fail
NCT	neck cut
LCT v	set up deep cut level to v
DCT v	deep cut with cut level (v)

Table A1.2 LVM instruction set (II)

arithmetic	instructions
LOD v i	dereferencing v-th cell and loading it to register Ri
STO v i	store (Ri) to v-th cell
MOV i j	move (Ri) to Rj
ADD i j	$(Ri) + (Rj) \Rightarrow R8$
SUB i j	$(Ri) - (Rj) \Rightarrow R8$
MUL i j	$(Ri) * (Rj) \Rightarrow R8$
DIV i j	$(Ri) / (Rj) \Rightarrow R8$
MOD i j	$(Ri) \text{ mod } (Rj) \Rightarrow R8$
SFL i j	$(Ri) \ll (Rj) \Rightarrow R8$
SFR i j	$(Ri) \gg (Rj) \Rightarrow R8$
INC i	$(Ri) + 1 \Rightarrow Ri$
DEC i	$(Ri) - 1 \Rightarrow Ri$
AND i j	$(Ri) \& (Rj) \Rightarrow R8$ (bitwise)
ORR i j	$(Ri) (Rj) \Rightarrow R8$ (bitwise)
XOR i j	$(Ri) \wedge (Rj) \Rightarrow R8$ (bitwise)
CMP n i	$n == (Ri) \Rightarrow R8$
CMG i j	generic comparison of (Ri) and (Rj) $\Rightarrow R8$ (-1,0,1)
MIN n i	$n \Rightarrow Ri$ (making an integer)
MDC v i	$(\text{address}(v), dc) \Rightarrow Ri$
MDS v i	$(\text{address}(v), ds) \Rightarrow Ri$
branching	instructions
JMI e i	jump to entry code (e) if (Ri) is an integer
JMC e i	jump to entry code (e) if (Ri) is a constant
JMN e i	jump to entry code (e) if (Ri) is a null list
JMS e i	jump to entry code (e) if (Ri) is a structure
JZE e i	jump to entry code (e) if $(Ri) == 0$
JNE e i	jump to entry code (e) if $(Ri) < 0$
JPO e i	jump to entry code (e) if $(Ri) > 0$
JLE e i	jump to entry code (e) if $(Ri) \leq 0$
JGE e i	jump to entry code (e) if $(Ri) \geq 0$
JMP e	jump to entry code (e) unconditionally
initialization	instructions
IV1 n	initialize a n-th cell unbound
IV2 n	initialize two successive cells starting n-th cell unbound
IV3 n	initialize three successive cells starting n-th cell unbound
IVN k n	initialize n successive cells starting k-th cell unbound
IT1 k e	assign entry point (e) to k-th cell
IT2 k e1 e2	assign entry points(e1,e2) to two successive cells starting k-th cell
IT3 k e1 e2 e3	...
ITN k n e1... en	assign e1...en to n successive cells starting k-th cell

Table A2. List Initialization Instructions

Operator	Operands	Meaning
ISL	I I I2	[SSI SCI]
ISG	I I I2 E	[SSI DCI]
ISN	I I I2	[SSI NIL]
ISU	I I I2	[SSI U VAR]
ISE	I I I2 I3	[SSI E VAR]
ISF	I I I2	[SSI F VAR]
ILL	I I I2	[SCI SCI]
ILG	I I I2 E	[SCI DCI]
ILN	I I I2	[SCI NIL]
ILU	I I I2	[SCI U VAR]
ILE	I I I2 I3	[SCI E VAR]
ILF	I I I2	[SCI F VAR]
IGL	I E	[DCI SCI]
IGU	I E	[DCI U VAR]
IGE	I I E I2	[DCI E VAR]
IGF	I E	[DCI F VAR]
ICL	I C	[CN SCI]
ICU	I C	[CN U VAR]
ICE	I I C I2	[CN E VAR]
ICF	I C	[CN F VAR]
INL	I	[NIL SCI]
INU	I	[NIL U VAR]
INE	I I I2	[NIL E VAR]
INF	I	[NIL F VAR]
IUL	I	[U VAR SCI]
IUG	I E	[U VAR DCI]
IUN	I	[U VAR NIL]
IUU	I	[U VAR U VAR]
IUE	I I I2	[U VAR E VAR]
IUF	I	[U VAR F VAR]
IEL	I I I2	[E VAR SCI]
IEG	I I I2 E	[E VAR DCI]
IEN	I I I2	[E VAR NIL]
IEU	I I I2	[E VAR U VAR]
IEE	I I I2 I3	[E VAR E VAR]
IEF	I I I2	[E VAR F VAR]
IFL	I	[F VAR SCI]
IFG	I E	[F VAR DCI]
IFN	I	[F VAR NIL]
IFU	I	[F VAR U VAR]
IFE	I I I2	[F VAR E VAR]

Appendix B. LVM Built-in Instructions

Table B.1 LVM built-in instructions for built-in predicates (1)

predicate (name/arity)	LVM instruction
bagof/3	B00
findall/3	B01
setof/3	B02
=:=, xfx, 700	B03
= =, xfx, 700	B04
>, xfx, 700	B05
>=, xfx, 700	B06
<, xfx, 700	B07
<=, xfx, 700	B08
is, xfx, 700	B09
@>, xfx, 700	B10
@>=, xfx, 700	B11
@<, xfx, 700	B12
@<=, xfx, 700	B13
=, xfx, 700	B14
\=, xfx, 700	B15
\.=, xfx, 700	B16
=, xfx, 700	B17
unify_with_occurs_check/2	B18
atom/1	B19
atomic/1	B20
compound/1	B21
number/1	B22
float/1	B23
integer/1	B24
nonvar/1	B25
var/1	B26
atom_char/2	B27
atom_codes/2	B28
atom_concat/3	B29
atom_length/2	B30
char_code/2	B31
number_chars/2	B32
number_codes/2	B33
sub_atom/5	B34
get-byte/1	B35
gct-bytc/2	B36
peek_byte/1	B37
peek_byte/2	B38
put_byte/1	B39
put_byte/2	B40
get_char/1	B41
get_char/2	B42
get_code/1	B43
get_code/2	B44

Table B.2 LVM built-in instructions for built-in predicates (II)

predicate (name/arity)	LVM instruction
peek_char/1	B45
peek_char/2	B46
peek_code/1	B47
peek_code/2	B48
put_char/1	B49
put_char/2	B50
put_code/1	B51
put_code/2	B52
nl/0	B53
nl/1	B54
char_conversion/2	B55
current_char_conversion/2	B56
current_op/3	B57
op/3	B58
read/1	B59
read/2	B60
read_term/2	B61
read_term/3	B62
write/1	B63
write/2	B64
write_term/2	B65
write_term/3	B66
write_canonical/1	B67
write_canonical/2	B68
writcq/1	B69
writcq/2	B70
arg/3	B71
copy_term/2	B72
functor/ 3	B73
abolish/1	B75
asserta/1	B76
assertz/1	B77
retract/1	B78
clause/2	B79
current_predicate/1	B80
current_prolog_flag/2	B81
set_prolog_flag/2	B82
fail/0	B89
true/0	B90
call/1	B91
catch/3	B92
once/1	B93
repeat/0	B94
throw/1	B95
halt/0	B96
halt/1	B97
open/3	BA0
open/4	BA1

Table B.3 LVM built-in instructions for built-in predicates (III)

predicate (name/arity)	LVM instruction
close/1	BA2
close/2	BA3
at_end_of_stream/0	BA4
at_end_of_stream/1	BA5
set_input/1	BA6
set_output/1	BA7
current_input/1	BA8
current_output/1	BA9
set_stream_position/2	BB0
stream_property/2	BB1
flush_output/0	BB2
flush_output/1	BB3

Bibliography

1. D. H. D. Warren, An Abstract Prolog Instruction Set, Technical Note 309, SRI International, Menlo Park, CA, 1983.
2. X. Li, A New Term Representation Method for Prolog, *J. Logic Programming*, Vol. 34(1), 43-58(1998).
3. P. Van Roy and A. M. Despain, High-Performance Logic Programming with the Aquarius Prolog Compiler, *IEEE Computer*, Vol.25(1), 54-68(1992).
4. A. Tayler, Parma-Bridging the Performance Gap between Imperative and Logic Programming, *J. Logic Programming*, Vol.28, 5-16(1996).
5. L. Sterling and E. Shapiro, *The Art of Prolog*, MIT Press, Cambridge, Mass., 1986.
6. M. Bruynooghe, An Interpreter for Predicate Programs: Part 1, Technical Report, CW 16, Katholieke Universiteit Leuven, 1976.
7. C. S. Mellish, An Alternative to Structure Sharing in the Implementation of a Prolog Interpreter, *Logic Programming*, Academic Press, 1982.
8. R. S. Boyer and J. S. Moore, The Sharing of Structure in Theorem Proving Programs, *Machine Intelligence 7*, Edinburgh University Press, 101-116 (1972).
9. D. H. D. Warren, Logic Programming and Compiler Writing, Technical Report, DAI 44, University of Edinburg, 1977.
10. H. Ait-kaci, Warren's Abstract Machine: A Tutorial Reconstruction, MIT Press, Cambridge. Mass., 1991.
11. A. Krall, The Vienna Abstract Machine, *J. Logic Programming*, Vol.29, 95-106 (1996).
12. A. Krall, Implementation Techniques for Prolog, Workshop Logische Programmierung, Berchit, 1-15, 1994.
13. N. F. Zou, T. Takagi and K. Ushijima, A Matching Tree Oriented abstract Machine for Prolog, In *Logic Programming: Proceedings of the Seventh International Conference*, MIT Press, Cambridge, Mass., 159-173, 1990.
14. N. F. Zou, Parameter Passing and Control Stack Management in Prolog Implementation Revisited, *ACM Trans. on Programming Languages and Systems*, Vol.18, 752-779(1996).
15. X. Li, Structure Sharing and Structure Copying Revisited, *Proceedings of 1996 Computer and Net Workshop on Parallelism and Implementation Technologies for (Constraint) Logic Languages*, 119-130, 1996.
16. X. Li, Program Sharing: A New Implementation Approach for Prolog, *PLILP'96*, LNCS, Springer, 1140, 259-273, 1996.

17. X. Li, *The LVM Specification*, Technical Report, Lakehead University, 1997.
18. C. S. Mellish, *Some Global Optimizations for a Prolog Compiler*, *J. Logic Programming*, Vol.1, 43-66(1985).
19. P. Deransart, L. Cervoni and A. Ed-dbali, *Prolog: The Standard: Reference Manual*, Springer-Verlag, New York, 1996.
20. A. V. Aho, R. Sethi and J. D. Ullman, *Compilers, Principles, Techniques, and Tools*, Addison-Wesley, 1988.
21. J. Beer, *The Occur-Check Problem Revisited*, *J. Logic Programming* Vol.5, 243-261(1988).
22. P. Van Roy, *Can Logic Programming Execute as Fast as Imperative programming?*, Ph.D. Thesis, University of California at Berkeley, 1990.
23. T. Lindgren, *Polyvariant Detection of Uninitialized Arguments of Prolog Predicates*, *J. Logic Programming*, Vol.28, 217-229(1996).
24. R. M. Colomb, *Enhancing Unification in Prolog through Clause Indexing*, *J. Logic Programming*, Vol.10(1), 23-44(1991).
25. B. Demoen, A. Marien and A. Callbaut, *Indexing prolog clauses*, North American Conf. on Logic Programming, 1989.
26. R. Jones and R. Lins, *Garbage Collection: Algorithms for Automatic Dynamic Memory Management*, John Wiley and Sons., 1996.
27. H. Schorr and W. M. Waite, *An Efficient Machine-Independent Procedure for Garbage Collection in Various List Structures*, *Communications of the ACM*, Vol.10(8), 501-506 (1967).
28. C. J. Cheney, *A Nonrecursive List Compacting Algorithm*, *Communications of the ACM*, Vol.13(11), 677-678(1970).
29. J. Bevemyr and T. Lindgren, *A Simple and Efficient Copying Garbage Collector for Prolog*, *Proceedings of Programming Language Implementation and Logic Programming*, LNCS 844:88-101, Springer-Verlag (1994).
30. H. Lieberman and C. Hewitt, *A Real-Time Garbage Collector Based on the Lifetimes of Objects*, *Communications of the ACM*, Vol.26(6), 419-429(1983).
31. A. W. Appel, *Simple Generational Garbage Collection and Fast Allocation*, *Software-Practics and Experience*, Vol.19(2), 171-183 (1989).
32. Y. Ding and X. Li, *Cache Performance of Chronological Garbage Collection*, *IEEE Canadian Conference on Electrical and Computer Engineering*, Waterloo, Vol.1, 1-4(1998).

33. **Y. Wang and X. Li, Compiling Prolog to Logic Virtual Machine, IEEE Canadian Conference on Electrical and Computer Engineering, Waterloo, Vol.1, 317-320(1998).**
34. **J. Tan and I.-P.Lin, Compiling Dataflow Analysis of Logic Programs, Proc. of '92 ACM/SIGPLAN Conf. on Programming Language Design and Implementation, San Francisco, June 1992.**