Lakehead University

Knowledge Commons,http://knowledgecommons.lakeheadu.ca

Electronic Theses and Dissertations | Retrospective theses

1993

Object-oriented implementation of Prolog

Fan, Wei

http://knowledgecommons.lakeheadu.ca/handle/2453/2195

Downloaded from Lakehead University, KnowledgeCommons

LAKEHEAD UNIVERSITY

OBJECT-ORIENTED IMPLEMENTATION OF PROLOG

BY

Wei Fan ©

A THESIS SUBMITTED TO

LAKEHEAD UNIVERSITY

IN PARTIAL FULFILLMENT OF THE REQUIREMENTS

FOR THE DEGREE OF

MASTER OF SCIENCE

DEPARTMENT OF MATHEMATICAL SCIENCES

THUNDER BAY, ONTARIO

MAY, 1993

© Wei Fan 1993

ProQuest Number: 10611855

All rights reserved

INFORMATION TO ALL USERS
The quality of this reproduction is dependent upon the quality of the copy submitted.

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

ProQuest 10611855

Published by ProQuest LLC (2017). Copyright of the Dissertation is held by the Author.

All rights reserved.
This work is protected against unauthorized copying under Title 17, United States Code
Microform Edition © ProQuest LLC.

ProQuest LLC.
789 East Eisenhower Parkway
P.O. Box 1346
Ann Arbor, MI 48106 - 1346

National Library of Canada

Bibliothèque nationale du Canada

Acquisitions and Bibliographic Services Branch

Direction des acquisitions et des services bibliographiques

395 Wellington Street
Ottawa, Ontario
K1A 0N4

395, rue Wellington
Ottawa (Ontario)
K1A 0N4

Your file Votre référence

Our file Notre référence

The author has granted an irrevocable non-exclusive licence allowing the National Library of Canada to reproduce, loan, distribute or sell copies of his/her thesis by any means and in any form or format, making this thesis available to interested persons.

L'auteur a accordé une licence irrévocable et non exclusive permettant à la Bibliothèque nationale du Canada de reproduire, prêter, distribuer ou vendre des copies de sa thèse de quelque manière et sous quelque forme que ce soit pour mettre des exemplaires de cette thèse à la disposition des personnes intéressées.

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

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

ISBN 0-315-86175-4

Canada

ABSTRACT

Logic programming is a discipline of describing problems in high-level abstraction by separating logic from control. Conventional Prolog interpretation or compilation models take a procedural view of Prolog programs. A description of interpretation models was summarized by Bruynooghe[Bru82] and a well-known compilation model was introduced by Warren[War83].

The goal of this study is to present an alternative approach to construct Prolog execution model to tackle the complexities caused by conventional Prolog execution models. By taking the advantage of object-oriented techniques, a new model – object-oriented model is proposed. Instead of decomposing a given Prolog program into a set of procedures, the model translates it into a collection of coordinated objects which simulate components of the problem to be solved.

First, the object-oriented model is described in terms of the object base and inference engine. The object base represents the components of Prolog programs naturally with corresponding objects in terms of AND/OR network. The inference engine, which specifies the operational behaviour of the objects, is embedded in the object base and independent of any specific Prolog program.

Secondly, implementation issues of a Prolog system based on the object-oriented model are presented. A transformation program is developed to translate any given Prolog program into a set of objects and assign the corresponding relations among them. The implementation of the inference engine adopts Robinson's resolution [Rob79] which consists of two major algorithms: unification and backtracking.

Finally, the first parameter hashing optimization and a uniform interface to adopt

new built-in predicates are addressed to show the extensibility of proposed Prolog system.

An experimental object-oriented Prolog system, LU-Prolog, has been developed based on the proposed model. An evaluation of the performance of LU-Prolog and its future directions are also presented in this thesis.

ACKNOWLEDGEMENTS

I heartily thank, first of all, my supervisor, Dr. Xining Li, for his continued guidance, inspiration and patience which made this thesis possible. His interest and excitement over the topics were often stimulating in the course of this study. His editorial suggestions led to substantial elaboration of this thesis.

I would like to express my gratitude for my parents and sisters. Although I have not seen them for two years, I could still feel their encouragement and love over the miles.

I am indebted to many friends of mine who support me, directly or indirectly, over the long time of this study.

I would like to thank Ms. Hedi Kogel for her comprehensive comments on the literacy contents of this thesis.

Finally, special appreciation is heartily extended to my external examiner Dr. John Cleary and internal examiner Dr. Yiyu Yao.

Contents

Approval Page ii

ABSTRACT iii

ACKNOWLEDGEMENTS v

List of Figures viii

1 INTRODUCTION 1
- 1.1 Logic Programming 1
- 1.2 Related Work 8
- 1.3 Motivation and Thesis Outline 17

2 OBJECT-ORIENTED PROLOG EXECUTION MODEL 20
- 2.1 The Comparison of Prolog Execution Models 20
- 2.2 The Object Concepts 23
- 2.3 Term Object Representation 25
- 2.4 Horn Object Representation – AND/OR Network 29

3 GENERATING THE OBJECT BASE 35
- 3.1 Structure of the Transformation Program 35
- 3.2 Object-oriented Approach of the Transformation Program 37
- 3.3 Creating Internal Term Objects 40
- 3.4 Creating Internal Horn Objects 42
- 3.5 Output Object Base 47

4 THE INFERENCE ENGINE 50
- 4.1 Term Object Sharing 50
- 4.2 Variable Object Manipulation 52
- 4.3 Term Object Unification 54
- 4.4 AND/OR Network Search Strategy 58

5 OPTIMIZATION AND BUILT-IN PREDICATES 64
- 5.1 First Parameter Hashing Optimization 65
- 5.2 Implementing Built-in Predicates 72

6 CONCLUSION 77

Bibliography **82**

List of Figures

1.1 An example of AND/OR graph 7
1.2 The simplified AND/OR graph 7
1.3 The transformation process of Aquarius Prolog 14

2.1 Prolog interpretation model 21
2.2 Prolog compilation model 22
2.3 The object-oriented Prolog model 24
2.4 The structure of term 26
2.5 Example: graphic representation of a term 27
2.6 Example: graphic representation of a term object 29
2.7 The hierarchy structure of HON 30
2.8 A horn object 31
2.9 AND/OR network of the program 33

3.1 The structure of transformation program 36
3.2 A skeleton of internal function object 39
3.3 The tree representation of a term 42
3.4 Internal goal objects 46
3.5 Example: internal Horn objects 48

4.1 A snapshot of term objects in execution 53

5.1 An original OR chain and the hash table 67
5.2 An example of first-parameter hashing optimization 71

6.1 General architecture of object-oriented Prolog machine 80

Chapter 1

INTRODUCTION

Logic programming is a discipline of describing problems in high-level abstraction by separating logic and control aspects of an algorithm. Programmers only declare relationships among values rather than how programs will be executed by a machine. Prolog, a logic programming language based on first order logic, was founded by Alain Colmerauer and his colleagues [SS86]. Conventional Prolog execution models, either in interpretation or in compilation, take a procedural view of Prolog programs. The aim of this study is to explore an alternative approach of modeling and implementing Prolog by using object-oriented techniques to achieve efficiency, extensibility and flexibility. This chapter begins with a discussion of some key concepts and issues on logic programming, followed by a review of related work. Finally, the motivation and objectives of this thesis are presented.

1.1 Logic Programming

The often-cited equation:

$$Algorithm = Logic + Control$$

was first introduced by Kowalski [Kow79]. It has been observed that significant man-power is used in handling control details in the course of designing algorithms and describing them by conventional procedural languages [SS86]. This phenomenon originates from von Neumann architecture which is composed of CPU, registers,

memory and input/output devices. The operations in a von Neumann computer fall into the following categories:

1. load data from memory to a register;
2. carry out arithmetic operation in the registers;
3. store data from registers to memory;
4. input/output operations.

It is critical in von Neumann computers that the execution sequence of instructions has to be specified in order to obtain the desired results. Although the development of high-level procedural programming languages provides more and more abstractions, control specifications are still necessary to make programs run correctly under the von Neumann architecture.

On the other hand, logic programming separates the logic and control aspects of an algorithm, hides control details from programming, and allows a very high-level description of relationships among values. The declarative aspect of logic programming is ideal, enabling programmers to concentrate on their goals rather than the ways how to achieve these goals.

In Prolog programs, *term* is the only data structure. A term can be a constant, a logical variable, or a function (structure) with terms as its arguments. For example, *123* and *abc* are *constant* terms, *f(a, 1)* is a *function* term with two constant arguments.

A *logical variable*, usually denoted by a character string starting with a capital letter, is a *term* whose value is determined dynamically in the course of execution.

For example, in a function: *father(X, Y)*, X and Y are both logical variables. A *free variable* is one that has an unknown value. As a computation proceeds, a free variable may be *instantiated* (or *bound*) to another term which is called the *binding* of the variable. An instantiated variable is identical with its binding (the term it is bound to) and maintains the same binding throughout the computation. This property of the logical variable is known as *single-assignment.*

A Prolog program consists of a set of *Horn clauses.* A Horn clause is an implication of the form

$$A \leftarrow B_1, \ldots, B_n.$$

where A represents the conclusion of the implication and is called the head, and $B_1, \ldots, B_n$ indicates the conjunction of conditions of the implication and is called the body of the clause. The clause is a *query* when A is absent, a *fact* when $n = 0$, or a *rule* otherwise. Clauses with the same name and arity define a *predicate.*

For example, the following clauses define the predicate *append*:

```
append([], L, L).

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

An informal reading of a Horn clause is "for each assignment of each variable, if $B_1, \ldots, B_n$ are all true, then A is true". It can also be interpreted as: to prove A, prove $B_1, \ldots, B_n$.

Prolog computation is initiated by a query in which conditions in conjunction are called *goals.* The computation of a Prolog program proceeds through a series of reductions, that is, a Prolog execution system tries to solve a given query by selecting

a goal from left to right and following a depth first search through its program looking for a matching clause, backtracking when it encounters a failure. The computation may terminate by answering "*yes*", which means the query is satisfied by the Prolog program, or "*no*" otherwise.

Theoretically, a logic programming language based on Horn clauses should be nondeterministic. The choice of goals to solve (AND-decision) is arbitrary. Success of the computation does not depend on the choice of goal. The choice of matching clauses (OR-decision) is also arbitrary provided it leads to a successful computation.

Prolog mimics the nondeterministic feature by imposing a fixed execution order plus backtracking on conventional machines. The imposed execution order makes Prolog an efficient logic programming language. However, such imposition incurs a penalty - any correct Horn clause program is a correct Prolog program, but not *vice versa.*

Prolog computation is based on unification among terms. Unification is a process to check whether two terms are unifiable with each other. Basically, unification is done through string matching and substitution. For example, we say that the unification between (X, Y), and *(bob,tom)* succeeds, in the sense that, with the substitution $\{X{=}bob,\ Y{=}tom\}$, (X, Y) and *(bob, tom)* shall become identical. Such a substitution is called the most general unifier (MGU) of these terms. In the case that no variables occur in two terms to be unified, the result of unification depends on their syntactic identity.

A *resolvent* is a set of conjunctive goals obtained by one step of computation. For example, consider the following goal list and Horn clause:

$$: -A_1, A_2, \ldots, A_i, \ldots, A_n.$$

$$A_i : -B_1, \ldots, B_k.$$

Suppose A_i is the current goal to be computed. If goal A_i and clause A_i are unifiable and the MGU of the unification is θ, then the resolvent of this computation is obtained by applying θ to the new goal list $A_1, A_2, \ldots, A_{i-1}, B_1, \ldots, B_k, A_{i+1}, \ldots, A_n$.

In general, a standard Prolog computation model, called the abstract interpreter [SS86], can be described as follows:

Input:
    A query $A_1, \ldots, A_n$ and a program P;
Output:
    yes if a proof of $A_1, \ldots, A_n$ from P is found; *no* otherwise;
Algorithm:
    begin
      T: resolvent
      θ: MGU
      Initialize T to $A_1, \ldots, A_n$
      while T is not empty
      begin
        choose a goal A from T, and a clause $A' \leftarrow B_1, \ldots, B_k$, from P
          such that A' and A are unifiable (if no such clause exists,
          exit the while loop);
        $\theta = unify(A, A')$;
        remove A from T and add $B_1, \ldots, B_k$ to T;
        apply θ to T, i.e., $A_1, \ldots, A_{i-1}, B_1, \ldots, B_k, A_{i+1}, \ldots, A_n$
      end
      If T is empty, output *yes*; otherwise output *no*.
    end.

Semantically, the relationship among components of Prolog program can also be viewed in terms of an *AND/OR graph.* The definition of an AND/OR graph can be defined recursively [Nil71] as follows:

1. The terminal nodes are solved nodes (since they are associated with primitive problems);
2. If a nonterminal node has OR successors, then it is a solved node if and only if at lease one of its successors is solved;
3. if a nonterminal node has AND successors, then it is a solved node if and only if all of its successors are solved.

For example, with a Prolog program:

```
a :- b,c,d.
a :- e,f,g.
c :- h,i.
c :- j.
f :- l.
f :- m.
f :- n.
```

Its AND/OR graph is illustrated in Figure 1.1 and Figure 1.2, where Figure 1.1 shows both AND nodes and OR nodes. The AND/OR graph can be simplified by combining AND- and OR-nodes and lining up AND branch as shown in Figure 1.2 [USL93].

It is clear, from the view point of an AND/OR graph, that a Prolog program is simply an AND/OR network connecting all Horn clauses. When a query is given, a

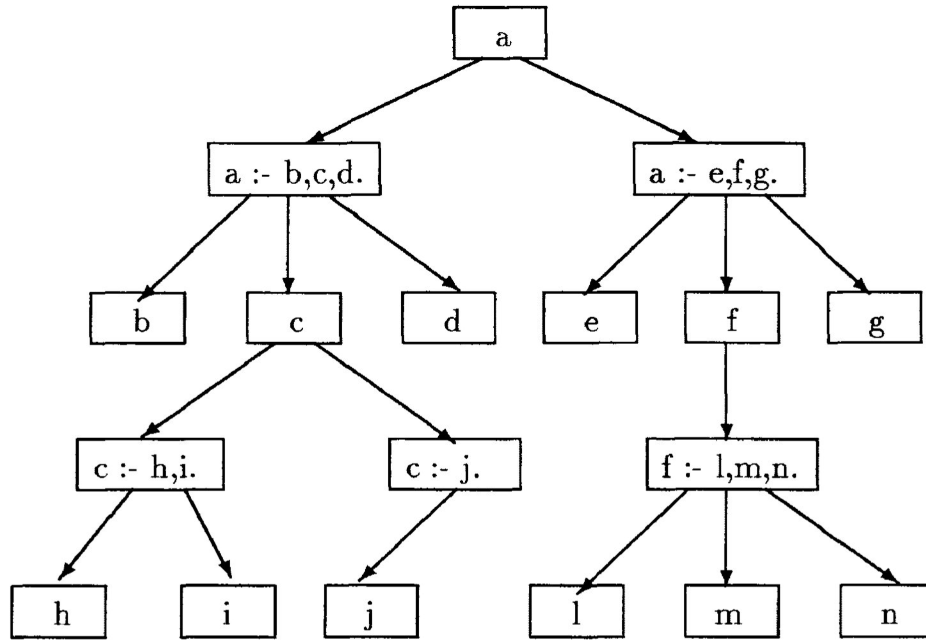


Figure 1.1: An example of AND/OR graph

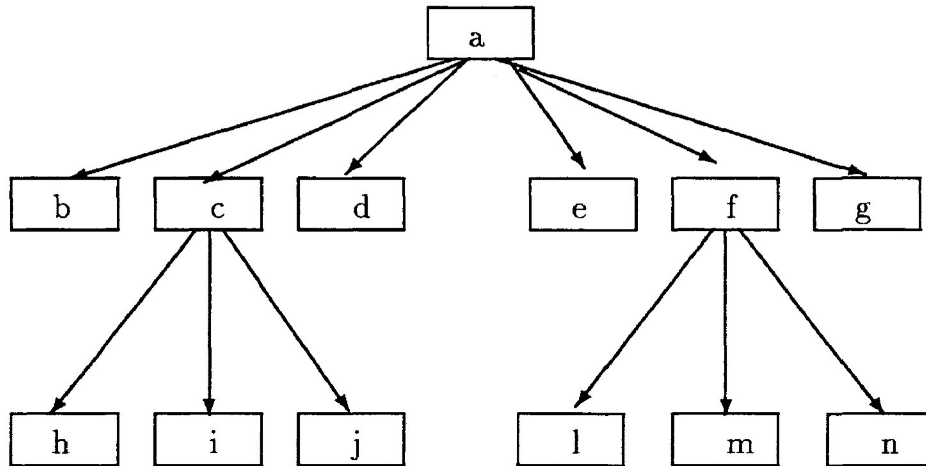


Figure 1.2: The simplified AND/OR graph

search space which provides all possible computation paths with respect to the given query is then formed.

Prolog, with its both solid theoretical foundation on logic and expressive power, has been used in a wide spectrum of applications, such as symbolic processing, artificial intelligence, expert systems, simulation, planning and deductive databases.

1.2 Related Work

The development of Prolog system has been driven by both improving the running efficiency [RD92, Con87, Mel85, HS84, TW84] and achieving the best use of memory [Mel82, Bru82]. Since the nondeterministic feature is simulated or approximated in terms of depth-first search and backtracking in most Prolog systems, it results in much more time and space overhead than those in conventional programming systems.

Although term is the only data structure in Prolog programs, this simple data structure is made complicated by logical variable binding. Consider the following example:

```
?- the_first(X).

the_first(Y) :-
    try_first(Y).
try_first(Z) :-
    same(Z, f(A,g(B))),
    try_other(A,B).
same(X,X).
...
```

In the course of execution, variable X is bound to another variable Y, and Y is bound to Z, then Z is bound to a function f(A,g(B)), where A and B can again be bound to any term. This unrestricted binding feature provides Prolog programmers with the great power to describe desired relationship. However, it also leads to a great difficulty in handling these variables. For example, it is often the case that a variable has to be dereferenced, that is, to follow its binding chain for its value. Moreover, the binding value of a variable can be a function term in which one or more parameter variables can again be bound to some other terms, ... and so on. It is necessary that a Prolog system maintains all of previous bindings for further reference and updates them during backtracking. Backtracking is the operation of a goal to undo the current unification and try another alternative clause if it fails to match a current clause, or to undo the current unification and go back to the previous goal when the failure goal has no alternative to try. Since binding chains and structures of binding values are determined at runtime according to different programs involved, to maintain and update these bindings can be complicated and expensive. Extensive researches have been done trying to find a proper term representation so that terms can be managed efficiently.

There are two commonly used term representations in Prolog implementation: Structure Sharing (SS) and Non-Structure Sharing (NSS). SS was first introduced by Boyer and Moore [BM72] and was used in earlier Prolog implementations, such as Marseilles interpreter and DEC-10 Prolog. The idea of NSS came from Bruynooghe [Bru76] and was adopted by several later Prolog interpreters and compilers. Comparisons of SS *vs.* NSS can be found in [Mel82] and [Bru82].

In a SS system, all terms are represented by *molecules*. A molecule consists

of two pointers to a *skeleton* and an *environment* respectively. A skeleton is the internal representation of a term description in which variables are referenced by their locations in environment - a chunk of memory holding the values of these variables. Inside a clause, all occurrences of a variable, no matter whether they appear in one term or in multiple terms, are designated by a single location in the environment. The idea of the SS scheme is that different instances of the same term share a single skeleton and differ only in their environments. Therefore, the cost of constructing a new term instance is quite low: it only needs an environment allocation plus two pointer assignments. On the other hand, to construct a new term instance in a NSS system, a concrete copy of the term must be created. The construction process needs to copy the ground description of the term and to allocate a location for every occurrence of a variable in the term. Newly created terms are retained in a memory area called *global stack* and variables in the global stack are called *global* variables. The NSS approach seems to require more time and space for constructing complex structures.

The SS scheme tends to be faster. However, a disadvantage is that all terms (molecules) must be retained in the forward execution of a Prolog program. It has no knowledge of whether a term is used as a selector (local instance) or a constructor (global instance). On the other hand, the NSS scheme only creates new terms when they become variable instantiations and may deallocate *local* variables - variables which do not carry information outside of their clauses - upon the termination of the clause's activations. Properly speaking, variables are all local to the clauses in which they appear. The terminology here is to denote that a global variable represents a long-lived object, while the life cycle of a local variable is determined

by the invocation of its clause - it may be discarded if the execution of the clause succeeds deterministically, or be frozen when goals in the clause body have more choices to be tried. Since terms often behave as selectors rather than constructors and the majority of predicates (clauses with the same name and arity) are mostly deterministic, the global space used by the NSS scheme is usually smaller than that used by the SS approach. Even though applying *mode declaration* [War77] or *global flow analysis* [Mel81] can reduce the amount of global variables in a SS system, it is still hard to compare the memory utilization of these two approaches since programs can be written such that any one method is worse than the other. Mellish [Mel82] comments that neither of the approaches is optimal in its use of the local and global stacks, and new methods are expected which have the advantages of both.

Another problem with respect to the operational behaviour of terms during unification is called *occur-check* - a time-consuming operation which tries to avoid unifying a variable with a term in which it occurs. Efficient solutions of the occur-check problem were studied in [Pla84, Bee88, Apt92] *etc.* The general idea of these solutions is to detect places where occur-check may be safely omitted and where it must be made. However, these proposals involve problems such as complicated global analysis, sophisticated mode declaration, and unnecessary occur-checks. On the other hand, Colmerauer[Col82] proposed a novel model of Prolog which does not perform occur-check in execution. He claimed that the unification overhead for handling cyclic structures (called *rational trees* in his paper) is less than that required by occur-check, and more important, cyclic structures may become very useful in representing static inter-linked data, such as graphs, grammar, and flow-charts.

Prolog execution models usually involve a kind of *transformation program.* Trans-

formation is a process of converting one program into another equivalent program in a different language. The scheme to make a Prolog program run under a computer system is to transform the Prolog program (referred as *source code*) to another equivalent set of internal representations or abstract instructions (called *target code*) which can be interpreted or compiled. The number of passes of transformation depends on the overall arrangement as well as the complexity of the task, such as the degree of difference between source code and target code and whether or not optimization is involved during the transformation. If more than one pass of transformation is required, the output of transformation before target code are referred to as intermediate programs.

From the procedural point of view, a Horn clause $A \leftarrow B_1, B_2, \ldots, B_n$ is translated to [SS86]:

```
procedure A
    call B1,
    call B2,
    ...
    call Bn,
end.
```

This attractive concise translation has inspired many dreams about the implementation of Prolog system. Unfortunately, it is not that simple. A Prolog system has to maintain and update all of necessary values bound by logical variables and other calling environment information in order to do backtracking once a goal encounters failure. Therefore, it is usually impossible to use the conventional subroutine call-return mechanism [Mel85].

To make Prolog more practical, Warren developed an execution model, the Warren Abstract Machine (WAM), for running Prolog efficiently on conventional hard-

ware [War83]. The *abstract instructions* of WAM can be classified into the following categories:

1. register manipulation;
2. stack operations;
3. execution control instructions;
4. unification and backtracking operations;
5. special instructions for maintaining calling environment and global variables.

The abstract instructions are machine independent. They can be implemented by assembly language on target machines or directly by hardware. The idea of WAM is to translate a Prolog program to a set of sequential WAM's abstract instructions which can be implemented and executed particularly efficiently by using machine code, trading speed for memory and performing special case analysis to simplify recursion and unification operations. It is obvious that the abstract instruction set is the one which closely depends on von Neumann architecture, in other words, Prolog is viewed by WAM mainly from conventional von Neumann machine, rather than its original logic foundation.

The first Prolog interpreter was developed in the early 70's by Colmeraurer and his colleagues [SS86]. Other better known variations include DEC-10 Prolog, Micro-Prolog and IC-Prolog. Following Warren's first Prolog compiler, more efficient compiling systems have been developed, such as BIM Prolog, Quintus Prolog and Aquarius Prolog. Most existing Prolog compilation models are based on WAM with extensions on instruction granularity, global analysis, extracting determinism, *etc.* For example, the Aquarius Prolog [RD92] is based on the Berkeley Abstract

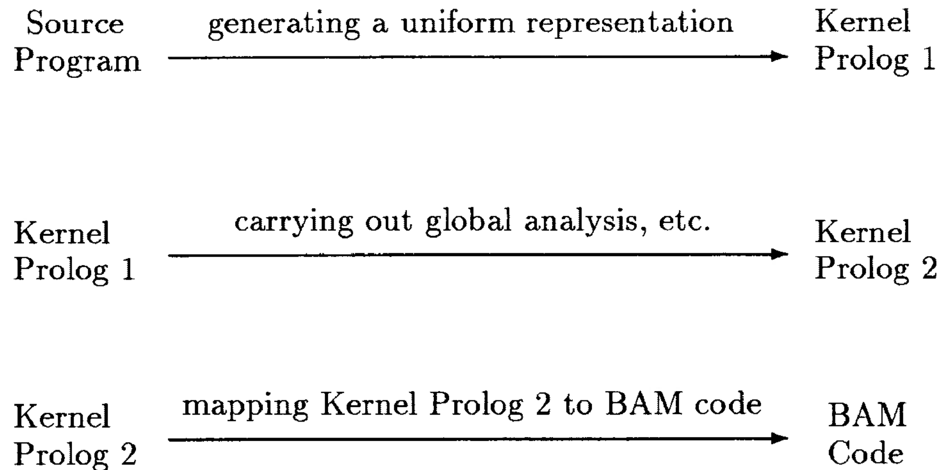


Figure 1.3: The transformation process of Aquarius Prolog

Machine (BAM) which extends WAM by adding more instructions to handle special operations, such as dereferencing operation. The transformation scheme adopted by Aquarius Prolog is shown in Figure 1.3.

It is perhaps the most complex part in the whole compilation to perform transformation from Horn clauses into BAM code. For example, it takes 36 minutes for Aquarius Prolog to compile a 1,138-line Prolog program [RD92]. To do this kind of compilation can be a painful experience. Tick and Warren [TW84] went further by employing a pipeline Prolog processor to speed up the execution of WAM's instruction set. However, it did not reduce the complexity of the transformation process. The main reason for such a complicated transformation from a Prolog program into a target code is that WAM and its variants mix-up the logic and control aspects of target code to which a declarative Prolog program is supposed to separate.

Prolog could still be impractical without efficient implementation of Prolog system by means of optimization [SS86]. Much of recent research on the implementa-

tion of Prolog system focus on various schemes of optimization to achieve efficiency. However, it is realized that to generate efficient code requires a more sophisticated instruction set which makes a further complication of the already complex instructions and run-time data structures.

As researchers face with increasing complexity in an attempt to bind more optimization schemes into a Prolog system to improve its efficiency, Mellish argues that it seems unlikely that conventional machines can be efficiently used by Prolog programs without the use of a complicated compiler [Mel85]. This situation remains unchanged by applying WAM since WAM itself is strongly characterized by von Neumann architecture.

Since the terms and clauses are not self-maintained objects in WAM, any operation associated with an optimization has to be arranged by the system. The complex states, behaviour of terms and clauses make it a difficult task to add various optimization features into such a mixed-up system. Even though the set of abstract instructions of WAM are subsequently augmented in order to support PROLOG [Mel85], the result is still unsatisfactory. For example, with the following program:

```
nrev([X|Y],Z) :-
        !,
        nrev(Y,Z1),
        append(Z1,[X],Z).
nrev([],[]).

append([X|Y],Z,[X|Y1]) :-
        !,
        append(Y,Z,Y1).
append([],X,X).
```

A part of its generated target code under WAM [Mel85] is shown in the following:

```
            procedure nrev/2
label_27:
            pop loc1
            push loc1
            ispair
            ifnot label_24
            push loc1
            back
            call nrev/2
            pop loc2
            push loc2
            push loc1
            ...
label_24:
            ...
label_29:
            ...
            endprocedure

            procedure append/3
            deref
            pop loc2
            deref
            pop loc1
            push loc1
            ispair
            ifnot label_33
            push loc1
            back
            ...
label_33:
            ...
label_38:
            endprocedure
```

where the meaning of some abstract instructions [Mel85] are explained as follows:

call < *name* >	Call the named procedure.
deref	Dereferences a Prolog term.
endprocedure	Return.
ifnot < *label* >	Transfer control to < *label* > if the top item on the user stack is non-FALSE.
ispair	Test whether an item is a list pair.
pop < *name* >	Pop the top element of execution into named variable.
procedure < *name* >	Mark the start for the named procedure.
push < *name* >	Push the value of named variable onto the execution stack.
...	...

Such a target code is far beyond the declarative semantics of Prolog programs. This results in complex schemes to perform the transformation. Furthermore, as other new optimization schemes are introduced, they usually heavily influence the built-up system. The man-power to modify such a system would be expensive.

1.3 Motivation and Thesis Outline

Conventional Prolog execution models are characterized by their procedural view of Prolog programs. One shortcoming of the standard interpretation model is that the components of Prolog programs are treated as passive data. The logic relations among these components are explored in runtime only. On the other hand, the con-

ventional compilation models are realized in the manner away from the spirit of logic programming for separating logic from control to achieve higher abstraction. Specifically, the target code is generated by abstract instructions which mix-up control details and logic relations with respect to the corresponding Prolog program.

The main objective of this study is to present an alternative approach to construct Prolog execution model to tackle the complexities caused by conventional Prolog execution models. Instead of being taken from a procedural point of view, as in the standard interpreter and WAM, Prolog programs are modelled and implemented in terms of a collection of collaborated objects. By taking the advantage of object-oriented techniques, a new model – Object-Oriented Model (OOM), is proposed. With OOM, an object base is constructed which reflects the components and the relationships among them with respect to the corresponding Prolog program. The inference engine is embedded in each object to simulate the operational behaviour of Horn clauses and is independent of any specific Prolog program. Such an approach of the Prolog execution model provides the framework with extensibility and flexibility to implement and improve a Prolog system.

Reaching our objectives involves two major steps: constructing a Prolog execution model and translating any given Prolog program into a corresponding object base. Chapter 2 compares conventional models with our proposal. From this investigation and discussion, we conclude that, by viewing components of a Prolog program as a set of collaborated objects, object-oriented Prolog execution model offers a greater potential to achieve our goals than conventional procedural models.

Chapter 3 discusses the issues of generating the object base, which particularly concerns with representing the components of a given Prolog program in terms of

objects and assigning the corresponding relationships to them. A practical transformation program to generate the object base is addressed.

Chapter 4 describes the inference engine which is embedded in the object base. The inference engine facilitates objects in the object base with operational behaviour based on the standard Prolog semantics. Most importantly, with the logic relations specified in the object base with respect to a given Prolog program, the inference engine only describes the control aspects of Prolog and is independent of any specific Prolog program.

The purpose of Chapter 5 is to show the extensibility of our model to adopt new schemes. The first parameter hashing optimization, which is the scheme to avoid unnecessary backtracking by looking at the hash value of the first parameters of current goal and Horn clause, is described. Other extensions such as the uniform interface to introduce new built-in predicates are also covered.

The framework described in the previous chapters provide great scope for further investigation such as the possibility to develop parallel mechanisms or object-oriented hardware. Concluding remarks and possibilities for future research are presented in Chapter 6.

Chapter 2

OBJECT-ORIENTED PROLOG EXECUTION MODEL

2.1 The Comparison of Prolog Execution Models

Although Prolog execution models may vary more or less in their details, the present concern will focus on their fundamental features. Prolog models are used to implement Prolog systems either by interpretation or by compilation. A typical Prolog *interpretation model* is shown in Figure 2.1.

The characteristics of the interpretation model is that it treats a Prolog program as a set of passive data. A Prolog interpreter is designed to perform all operations on these data. This approach results in considerable difficulties in manipulating terms and Horn clauses in the Prolog program, especially those which are required by backtracking. As a result, great efforts have to be made to keep track of these passive data.

The compilation system generates a target code which is composed of abstract instructions, and which is then compiled by the existing system to produce final executable code. The structure of a Prolog *compilation model* is depicted in Figure 2.2. The strategy is adopted by most conventional compilers. However, since Prolog differs from any procedural programming language by its separation of logic from control, the traditional approach to Prolog compilers enforces procedural view

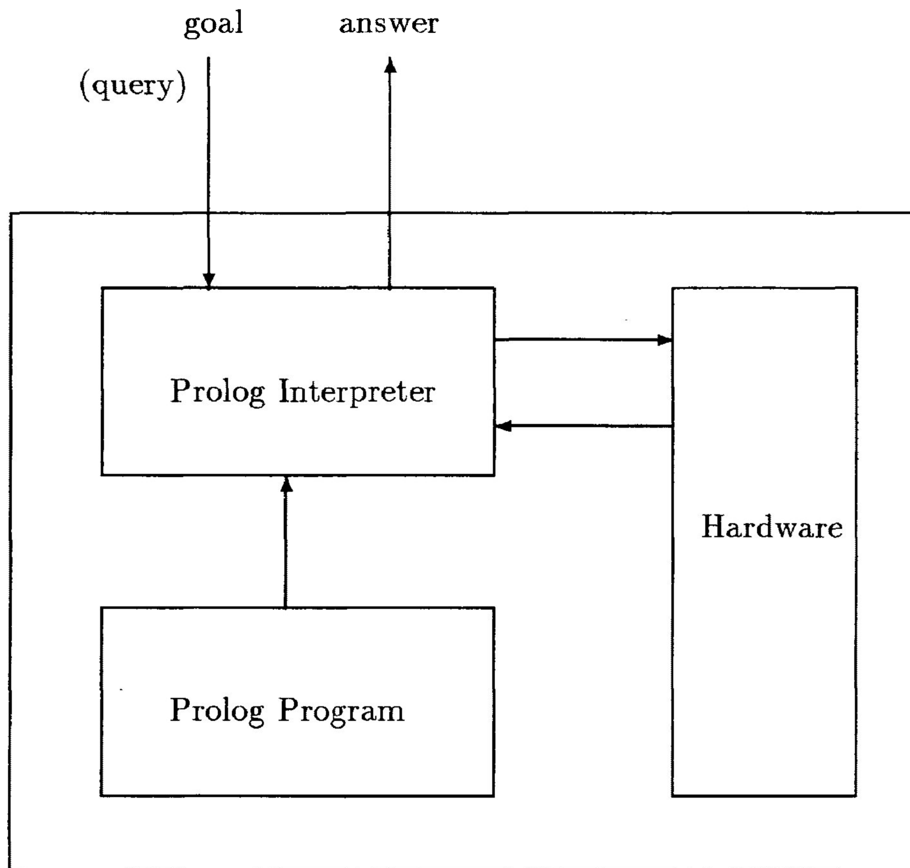


Figure 2.1: Prolog interpretation model

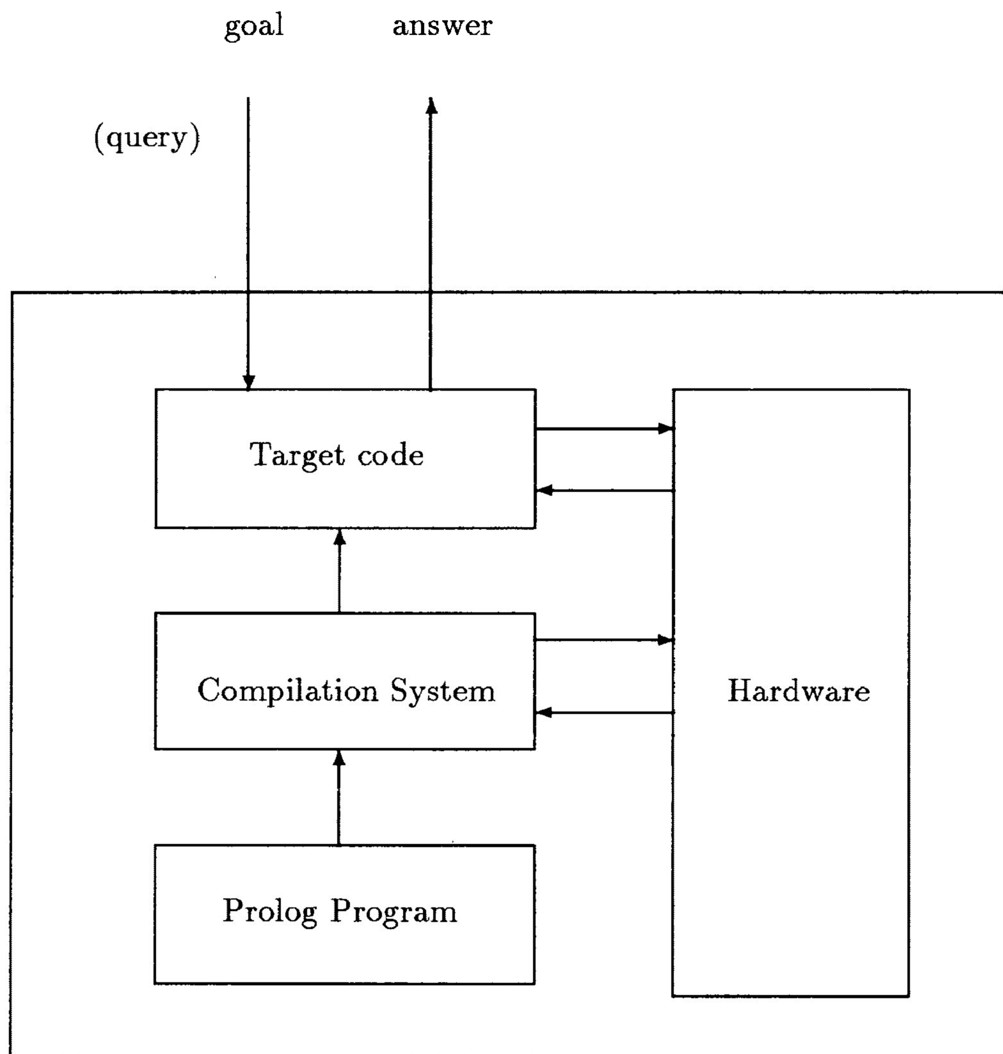


Figure 2.2: Prolog compilation model

on Prolog programs. This is contrary to the primary feature of Prolog, and leads to a complicated compilation process. On the other hand, the object-oriented Prolog model represents the components of Prolog programs naturally with corresponding objects. Figure 2.3 shows the structure of object-oriented Prolog model.

The *object base* is organized in the form of AND/OR network, where each node is represented by the corresponding Horn object. The advantages of this approach are that the target code, which is generated by a transformation program, only represents the relationships assigned by the Prolog program, whereas the *inference engine*, which provides the mechanisms for the objects to facilitate their behaviours, is embedded in each object. One important property of the inference engine is that it is predefined and independent of any specific Prolog program.

2.2 The Object Concepts

Since object-oriented techniques play a critical role in our model, we shall first to discuss some of concepts and features of object-oriented paradigms.

An *object* is a run-time entity encapsulating both state and behaviour. Each object has a name as its identity. Objects with similar characteristics are grouped into a *class* which specifies the state as an abstract data structure and the behaviour as a set of services. The services are appropriate computations applied thereto. An object may issue requests for services performed by other objects. A *request* is a message to the object which provides the service. Performing a request involves executing some code (a method or a member function) on the associated data. The definition and implementation of one class can be inherited by other newly defined

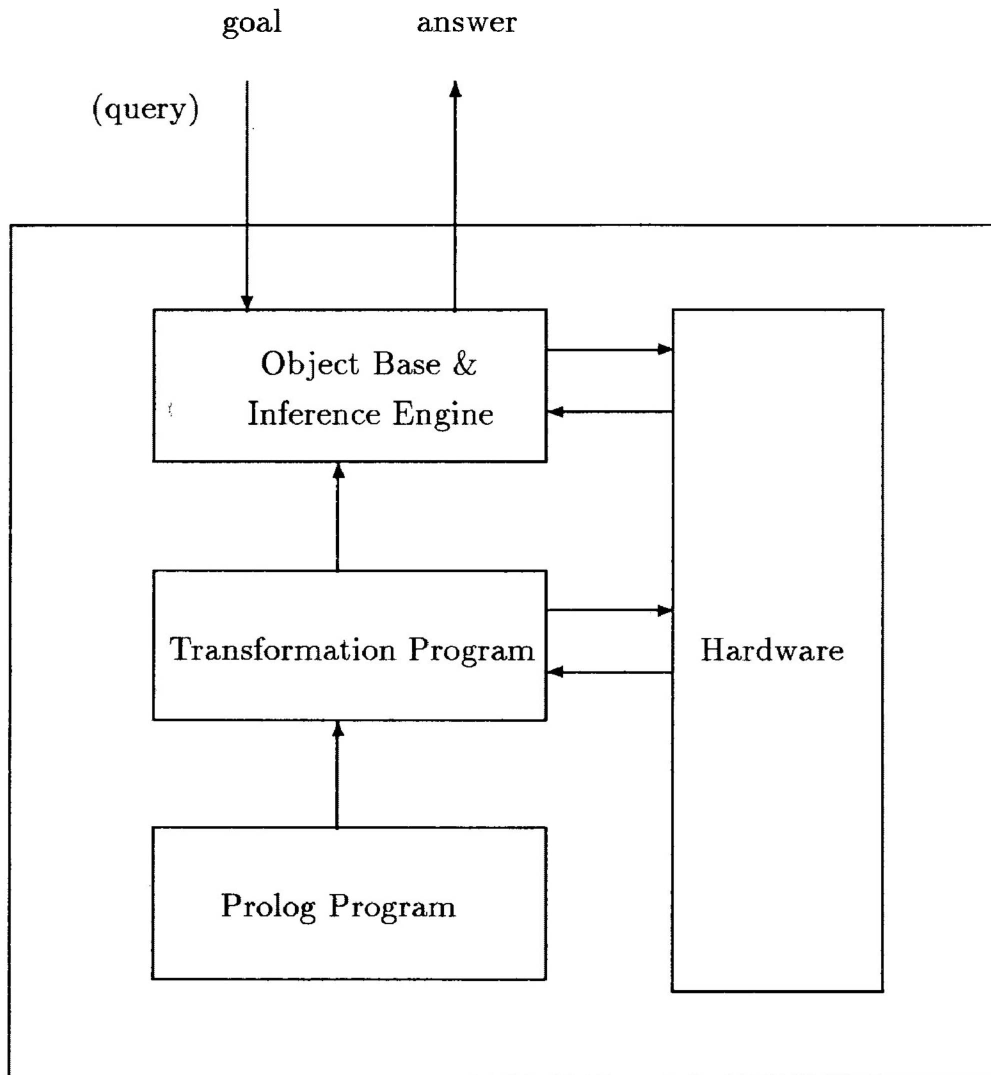


Figure 2.3: The object-oriented Prolog model

classes. The *inheritance relation* facilitates reusability and extensibility of software systems. An object reference is polymorphic if it refers to objects of more than one class over time. Dynamic binding provides a powerful tool for associating code with a given request at runtime.

The object-oriented paradigm takes a modelling point of view of a complex system. It specifies collections of coordinated entities to simulate components of the problems to be solved. This paradigm offers a way to analyze Horn clauses, by decomposing them as a set of interacting objects. The object-oriented programming is somewhat like the conventional procedural one, except that concepts such as object, class, polymorphism, inheritance, and dynamic binding make it more powerful and flexible.

The object base reflects the relationships among values specified by a Prolog program. In order that a Prolog program can be expressed in terms of objects, the *hierarchical structures* among different kinds of terms and Horn clauses have been developed according to their degree of similarities. The object base is then constituted by allied representation of term objects and Horn objects with respect to a Prolog program.

2.3 Term Object Representation

The basic data structure in Prolog programming language is called *term*, represented by several concrete forms such as *atom, integer, variable* and *function.* Accordingly, the hierarchy structure of term objects is organized in their natural way such that atom, integer, variable and function are classes derived from a same common base

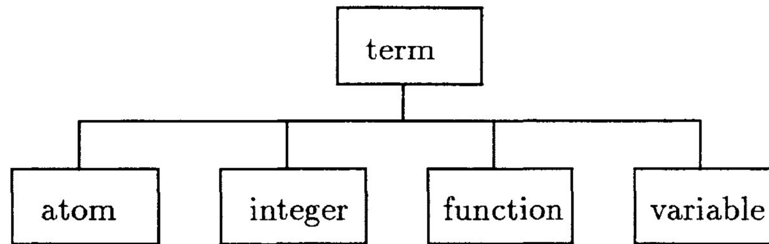


Figure 2.4: The structure of term

class - *term*, as shown in Figure 2.4.

In the following discussion, we introduce a notation: *obj* : $p_1, p_2, \ldots$ to represent the initialization of *obj* by $p_1, p_2, \ldots$. Thus, the term objects in an object base look like as follows:

```
integer objects: {
        int_obj_1: integer_1;
        int_obj_2: integer_2;
        ...};

atom objects: {
        atom_obj_1: symbolic_name_1;
        atom_obj_2: symbolic_name_2;
        ...};

variable objects: {
        var_obj_1: symbolic_name_1;
        var_obj_2: symbolic_name_2;
        ...};

argument list objects: {
        arg_obj_1: term_obj_11, term_obj_12,...;
        arg_obj_2: term_obj_21, term_obj_22,...;
        ...};

function objects: {
```

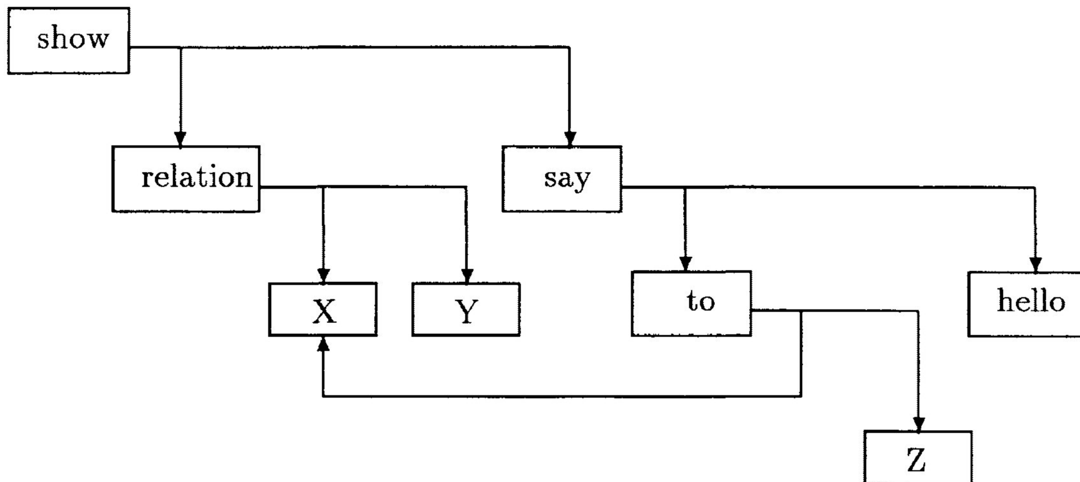


Figure 2.5: Example: graphic representation of a term

func_obj_1: functor_1, argument_list_obj_1;
func_obj_2: functor_2, argument_list_obj_2;
...};

Each term object is attached by one or more parameters which represent the corresponding relationships defined by a Prolog program. The definitions of objects with respect to integer, atom and variable, are straightforward. The class of argument list objects, is introduced to form the argument lists of function term (refer to page 28). For example, suppose that we have the following term:

$$show(relation(X, Y), say(to(X, Z), hello))$$

The implied relationship among the inner terms can be illustrated graphically in Figure 2.5.

The corresponding term objects can be described as follows:

atom objects: {

```
        atom_obj_1: "hello"
    };

variable objects: {
        var_obj_1: "X";
        var_obj_2: "Y"
        var_obj_3: "Z"
    };

//  in the following, the argument list objects specified
//  are corresponding to:
//          (X, Y);
//          (X, Z);
//          (to(X, Z), "hello");
//          (relation(X, Y), to(X,Z), "hello"))
//  respectively.

Argument list objects: {
        arg_obj_1: var_obj_1, var_obj_2;
        arg_obj_2: var_obj_1, var_obj_3;
        arg_obj_3: func_obj_2, atom_obj_1;
        arg_obj_4: func_obj_1, func_obj_3
    };

function objects: {
        func_obj_1: "relation", arg_obj_1;
        func_obj_2: "to", arg_obj_2;
        func_obj_3: "say", arg_obj_3;
        func_obj_4: "show", arg_obj_4
    };
```

It should be noted that the two occurrences of X are represented by a single object *var_object_1*. The relationships among the above objects can be shown graphically in Figure 2.6. It is obvious by comparing Figure 2.5 with Figure 2.6 that each term has been represented by its corresponding object.

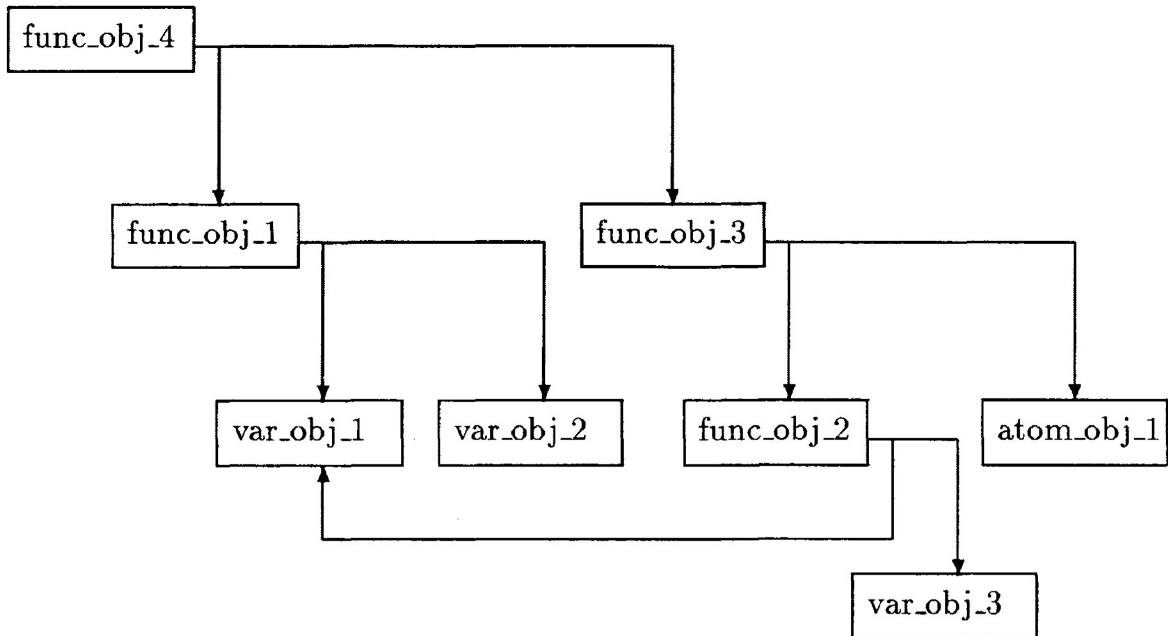


Figure 2.6: Example: graphic representation of a term object

2.4 Horn Object Representation – AND/OR Network

Essentially, we view a Prolog program as an AND/OR network in terms of Horn objects. Having analyzed the degree of similarity among different kinds of clauses, we introduce a hierarchy structure in Figure 2.7.

The inheritance relation in the object-oriented paradigm is often called the "is a" relation. From Figure 2.7, it is easy to see that a rule is a clause, and in turn a clause is a horn. Two classes, root and stub, are used for optimization and will be discussed later. The body of a Horn clause is a sequence of goals. Correspondingly, a goal class is defined in our model for creating goal objects. Horn objects can be specified as follows:

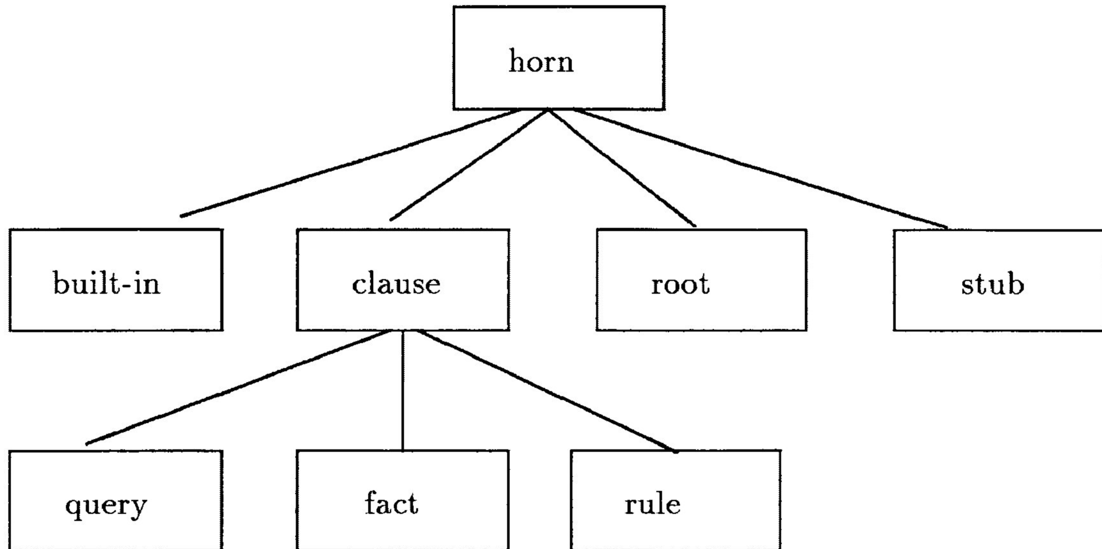


Figure 2.7: The hierarchy structure of HON

```
fact objects: {
      ft_obj_1: functor_1, arg_list_1, next_1;
      ft_obj_2: functor_2, arg_list_2, next_2;
      ...};

goal objects: {
      gl_obj_1: functor_1, horn_1, arg_list_1, pre_gl_1,nt_gl_1;
      gl_obj_2: functor_2, horn_2, arg_list_2, pre_gl_2,nt_gl_2;
      ...};

rule objects: {
      rl_obj_1: functor_1, arg_list_1, next_1,gl_1;
      rl_obj_2: functor_2, arg_list_2, next_2,gl_2;
      ...};
```

It is imperative that Horn objects are connected with each other to represent the AND/OR network. This is achieved by the appropriate specification of parameters for each object. Goal objects which are embedded in a rule object are double-linked

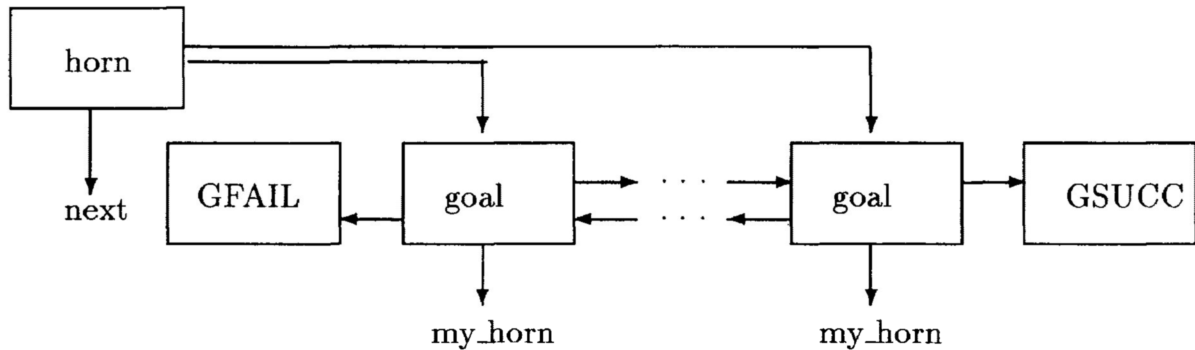


Figure 2.8: A horn object

by the pointers *pre_gl_i* and *nt_gl_i* in the same order assigned by the Prolog program to form AND successors of the rule object. Furthermore, fact or rule objects with the same predicates are linked by the pointer *next_i* to construct OR successors, where the chain of OR successors can be accessed by the goal object in a depth-first manner. The OR successors are organized in the same sequence as their corresponding Horn clauses in the Prolog program.

A graphic representation of a Horn object is shown in Figure 2.8. It should note that Horn objects which define a same predicate are linked together by the pointer next in their original program context order to form OR-chain. If there is no more Horn object to be linked, the *next* will be assigned as NULL. Further, two special goal pointers, GFAIL and GSUCC, are used to be the sentinels of the goal list: GFAIL is assigned to the *pre_gl_i* pointer of the first goal and GSUCC is assigned to the *nt_gl_i* pointer of the last goal in the list.

To illustrate the AND/OR network established in the object base, we consider the following example.

```
nrev([X|Y],Z) :-
        nrev(Y,Z1),
        append(Z1,[X],Z).
nrev([],[]).

append([X|Y],Z,[X|Y1]) :-
        append(Y,Z,Y1).
append([],X,X).
```

The objects is then described in the following form:

```
.../* definitions of the term objects */

argument list objects: {
        arg_obj_1:      ... ;   // specify [X|Y]
        arg_obj_2:      ... ;   // specify ([X|Y],Z)
        arg_obj_3:      ... ;   // specify (Y,Z1)
        arg_obj_4:      ... ;   // specify [X]
        arg_obj_5:      ... ;   // specify (Z1,[X],Z)
        arg_obj_6:      ... ;   // specify ([],[])
        arg_obj_7:      ... ;   // specify [X|Y]
        arg_obj_8:      ... ;   // specify [X|Y1]
        arg_obj_9:      ... ;   // specify ([X|Y],Z,[X|Y1])
        arg_obj_10:     ... ;   // specify (Y,Z,Y1)
        arg_obj_11:     ... ;   // specify ([],X,X)
    };

fact objects: {
        ft_obj_1: "nrev", arg_obj_5, NULL;
        ft_obj_2: "append",arg_obj_11, NULL
    };

goal objects: {
        gl_obj_1: "nrev", rl_obj_1, arg_obj_3, GFAIL, gl_obj_2;
        gl_obj_2: "append", rl_obj_2, arg_obj_5, gl_obj_1, GSUCC;
        gl_obj_3: "append", rl_obj_2, arg_obj_10, GFAIL, GSUCC
    };
```

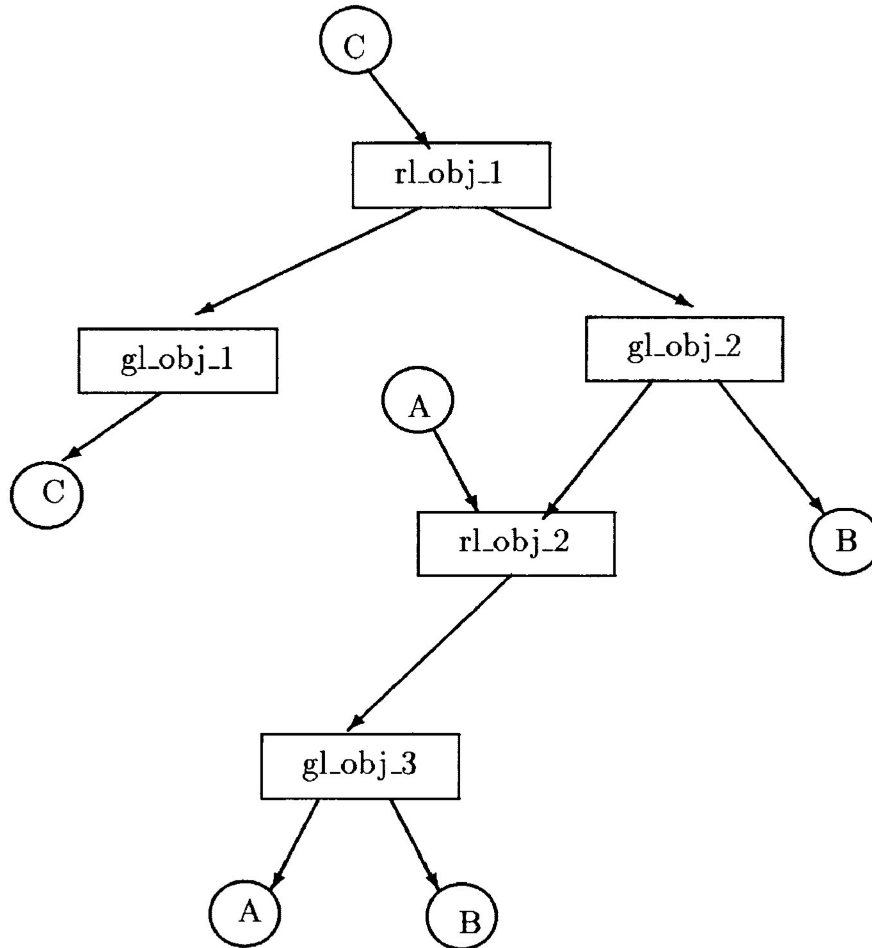


Figure 2.9: AND/OR network of the program

```
rule objects: {
        rl_obj_1: "nrev", arg_obj_2, ft_obj_1, gl_obj_1;
        rl_obj_2: "append", arg_obj_9, ft_obj_2, gl_obj_3
    };
```

The representation of the corresponding AND/OR network is shown in Figure 2.9. Finally, it should be emphasized that this network only presents the data and their relationships, and no execution instructions are involved in the generated object

base.

Chapter 3

GENERATING THE OBJECT BASE

This chapter focuses on the object-oriented approach of the transformation program, which translates a given Prolog program into an object base. To form the object base, the particular concern is the way of various objects, i.e., term objects and Horn objects being represented, and the way the relationships among them being set up properly with respect to the Prolog program. A set of objects, referred to as *internal objects*, are created by the transformation program according to the components of the Prolog program. With the coordination of the object generator, the object base is obtained through the corresponding output of *internal objects.* Such a transformation approach, by taking advantage of object-oriented techniques, provides the transformation program with flexibility to make further extension.

3.1 Structure of the Transformation Program

The structure of the transformation program, as shown in Figure 3.1, is constituted by the lexical analyzer, parser, table handler and object generator.

The *lexical analyzer* decomposes a Prolog program into lexical elements, i.e. basic syntactic units, such as atoms, functions, predicates, *etc.* The lexical elements are represented by corresponding tokens which consist of keywords, identifiers and symbols found in the program [TS82]. The *parser* then carries out syntactic analysis of the tokens produced by the lexical analyzer. In the course of parsing, the information

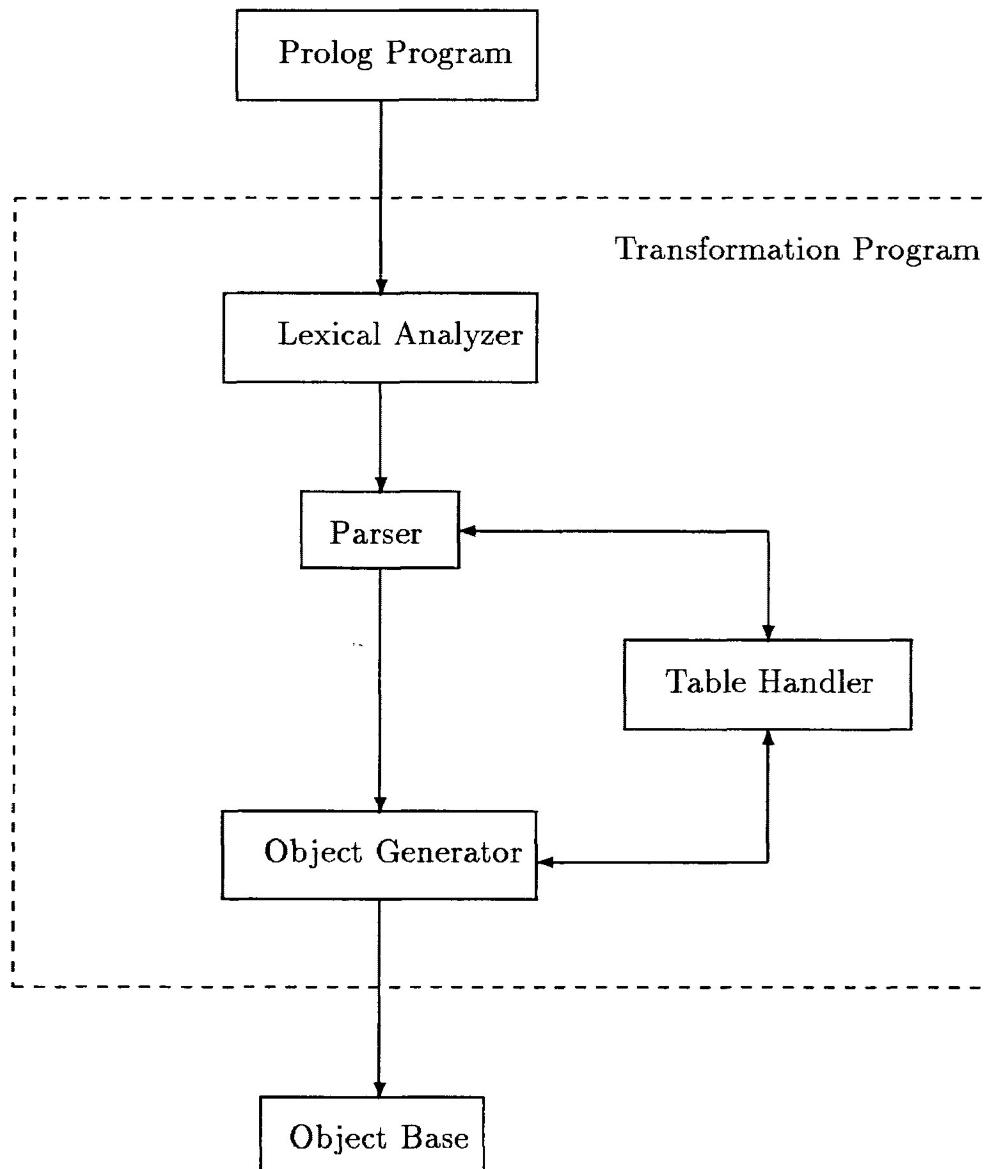


Figure 3.1: The structure of transformation program

which will be used at final phase to generate the object base, is stored in tables. The *object generator*, with the information in the tables, is to form the object base by specifying corresponding objects and by imposing proper relationships among them.

The traditional way of implementing transformation process, either in interpreter or compiler, is through a group of subroutines which are activated by a main routine. However, our transformation program is characterized by its object-oriented manner. The transformation program creates a collection of objects, referred to as internal objects, to represent the corresponding components of the Prolog program. The internal objects are characterized by their one-to-one mapping relationship with the objects in the object base. Thus, the internal objects serve as a bridge between the source program and the object base. By incorporating the features of an object-oriented paradigm such as abstraction, inheritance, *etc.*, the object-oriented approach provides a different and, to some extent, a more powerful way to implement the transformation process.

The transformation program was originally developed, as discussed in this chapter, without involving any optimization scheme, However, the extensibility of the transformation program makes it possible to be easily upgraded when optimization schemes are considered. This will be discussed later.

3.2 Object-oriented Approach of the Transformation Program

Conventional implementation takes a procedural view of the transformation process by decomposing the task into a collection of procedures or functions. In the course of

transformation, symbolic tables are used to maintain the results of parsing. Finally, the object base is produced by using the information in the symbolic tables.

Instead of dealing with passive symbolic strings, the object-oriented paradigms provides a means to represent the results of parsing by a set of active objects, i.e., the internal objects. The term internal objects is used to indicate the objects employed by the transformation program, and to distinguish them from objects in the object base. The transformation program is developed to allow the use of the same hierarchical structures of *term objects* and Horn objects as described earlier. Two classes of internal objects, i.e., internal term objects and internal Horn objects, are created by the transformation program to represent the components in the Prolog program. Accordingly, tables are used to maintain various objects as well as other information. It should be clear that the internal objects are intended to form the object base, rather than to perform a Prolog computation.

The object-oriented paradigm makes it possible to decentralize the task among related objects. For example, assuming that all objects have been created and other necessary information has been obtained, each internal object will output its specific format to the object base, which is under the coordination of the object generator.

An internal objects is created according to its specific type, such as atom, function, fact, *etc.* In order to generate the appropriate objects in the object base, the key point is that each internal object obtains sufficient information such as its symbolic name and its relationship with other objects. For example, a function object should know which term objects serve as its parameters. To provide a concrete picture of the internal objects, we consider one of internal objects – *internal function objects.*

The skeleton of an internal function object is shown in Figure 3.2, which includes

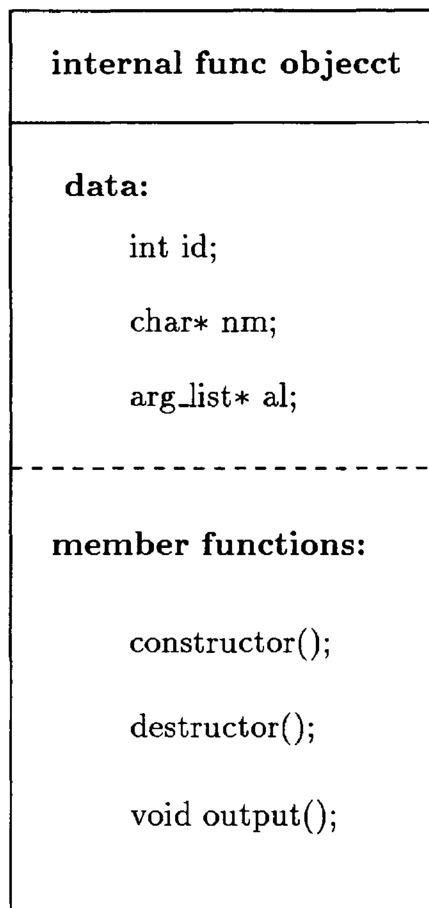


Figure 3.2: A skeleton of internal function object

an identification number, id; a symbolic name of functor, nm; and an argument list, al. These data items are used to record the information necessary to specify corresponding function objects in the object base. In the course of parsing, a global counter *gfid* is used to keep the sequence number of the internal function objects. The value of *gfid* is attached to the internal function object being created, and then *gfid* is increased by 1. The member function output(), which is specified as a virtual function in the *internal term class*, i.e., its base class, is defined here to meet the specific requirement to format the corresponding function object in the object base.

The advantage of this decentralized approach is that it allows for the design of the transformation process in the natural way with respect to the Prolog program and object base. It makes easier to modify some parts of the transformation program when optimization schemes are introduced, to add new term structures to an extended Prolog, and to enhance better maintainability and understandability of the transformation program.

3.3 Creating Internal Term Objects

Prolog adopts a unique name, i.e., term, for representing values involved in a computation. A term can be specified recursively as:

- term ← integer, atom, or variable;
- term ← functor + parameter (one or more terms).

Two important issues need to be addressed in dealing with terms. The first issue concerns the object sharing:

1. atoms with the same symbolic name in a Prolog program are represented by a single internal atom object;

2. variables with the same symbolic name share one internal variable object only when they occur in the same Horn clause.

For example, consider the following two predicates:

```
share_1(X,X,foo).
share_2(X,X,foo).
```

Although all of variables have the same symbolic name X, they are two different variables local to their clauses. Therefore, both X's in share_1 are represented by one variable object, whereas both X's in share_2 are represented by another variable object. On the other hand, only one internal atom object is created with respect to two occurrence of foo, no matter in which Horn clauses they appear.

The second issue deals with the criterion to determine the sequence of term objects to be generated. This specifically concerns with function terms, where other terms are embedded as parameters. It is noted that the structure of any function term is simply a tree, which can be specified recursively:

- the root of a tree represents the functor of the term;
- the children of the root correspond to the parameters of the term respectively;
- a leaf node may represent an atom, integer, or a variable.

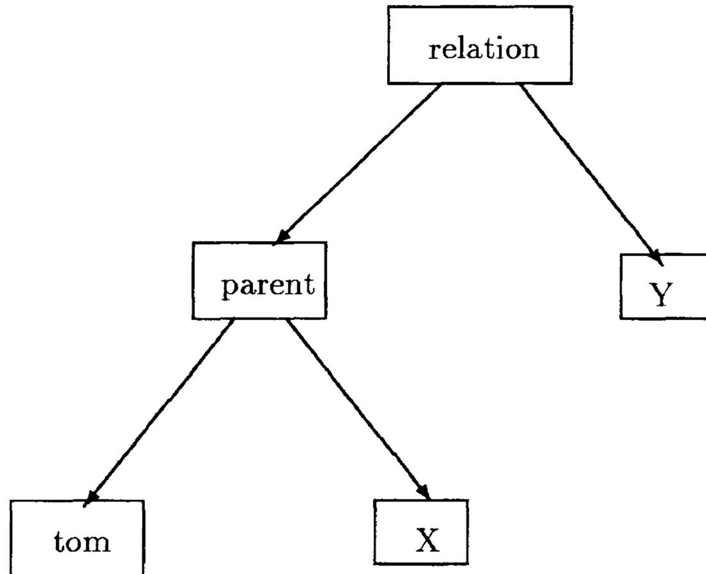


Figure 3.3: The tree representation of a term

For example, the tree representation of the function term: *relation(parent(tom,X), Y)* is shown in Figure 3.3.

With this observation, the transformation program generates the term objects in the sequence according to the postorder traversal. The reason is that, whenever a function is created its parameters can always be referred to, so that the relationship between the function and its parameters can be set up properly.

3.4 Creating Internal Horn Objects

A Prolog program is represented by an AND/OR network. In the course of parsing, a set of objects are generated to represent the components of a Prolog program. These objects also reflect the same relationships specified by the Prolog program. The major procedures to form an AND/OR graph among internal Horn objects are

shown in the following code, where *h_str* and *g_str* represent a goal and a Horn in symbolic form respectively; accordingly, *h_obj* and *g_obj* represent their corresponding internal objects.

```
Part I:

input:
        a list of goals in symbolic form which embedded in a rule;
output:
        an AND chain;
algorithm:
        begin
           glid = 1;
           while (g_str)
              begin
                 g_str = g_str→next;
                 /* processing the argument of the goal */
                 g_obj = create(g_str,...);
                 /* link with the previous goal objects */
                 table_handler = g_obj;
                 glid = glid + 1
              end
        end. { Part I }


Part II:

input:
        a Prolog program;
output:
        OR chains;
algorithm:
        begin
           ftid = 1; rlid = 1;
           while (h_str)
              begin
                 /* processing the argument of the Horn */
                 switch:
```

```
            case FACT:
               h_obj = create(h_str,...);
               ftid = ftid + 1;
            case RULE:
               /* generate AND chain: AND_chain */;
               h_obj = create(AND_chain,h_str,...);
               rlid = rlid + 1;
        end { while }
        table_handler = h_obj;
        /* link h_obj with other internal Horn objects
        with the same predicate */
    end. { Part II }
```

Part III:

```
input:
    table handler;
output:
    AND/OR graph;
algorithm:
    begin
    while (g_obj)
       begin
          The_OR_chain = table_handler(g_obj);
          g_obj.get(The_OR_chain);
          g_obj = g_obj→next;
       end
    end. { Part III }
```

The procedure in Part I assumes that the input, i.e., the set of goals embedded in a rule clause, have been converted into the uniform format by previous processing. Arithmetic expressions are all represented by prefix notation. For instance, X is (3+5)*4 has been changed into is(X,*(+(3,5),4)). In a clause, AND is the only relation among goals. The clause, such as,

a :- (b,c); (d,e); (f,g,h).

has been changed equivalently into the following clauses:

```
a :- b,c.
a :- d,e.
a :- f,g,h.
```

The procedure in Part I creates a set of internal goal objects attached to an internal Horn object, and each internal goal object is identified by a unique ID number. By means of message passing, each internal goal object also obtains the ID numbers of its connected neighbours. For example, with respect to the Horn clause H = A,B,C., its internal goal objects are created as shown in Figure 3.4. The internal goal object B knows that the ID numbers of its previous and next neighbours are 10 and 12 respectively; the internal goal object A knows it only has a next neighbour with ID number 11; and so on.

Part II is used to construct OR chains connecting internal Horn objects. An OR chain links all Horn objects with the same predicate and arity. As the matter of fact, an OR chain is not linked by pointers, but through the ID numbers assigned to the internal Horn objects. Therefore, an internal Horn object recognizes its next OR candidate by its ID number. After being created, all internal Horn objects are stored in a table to be manipulated by the table handler.

The last step is to link each internal goal object to its OR chain, as described in Part III. The corresponding OR chain is obtained through looking up tables. The table handler first directs the internal goal objects to search the built-in predicate table. The table which contains OR chains from the Prolog program, is searched only when the internal goal object encounters failure in an attempt to find a match among built-in predicates.

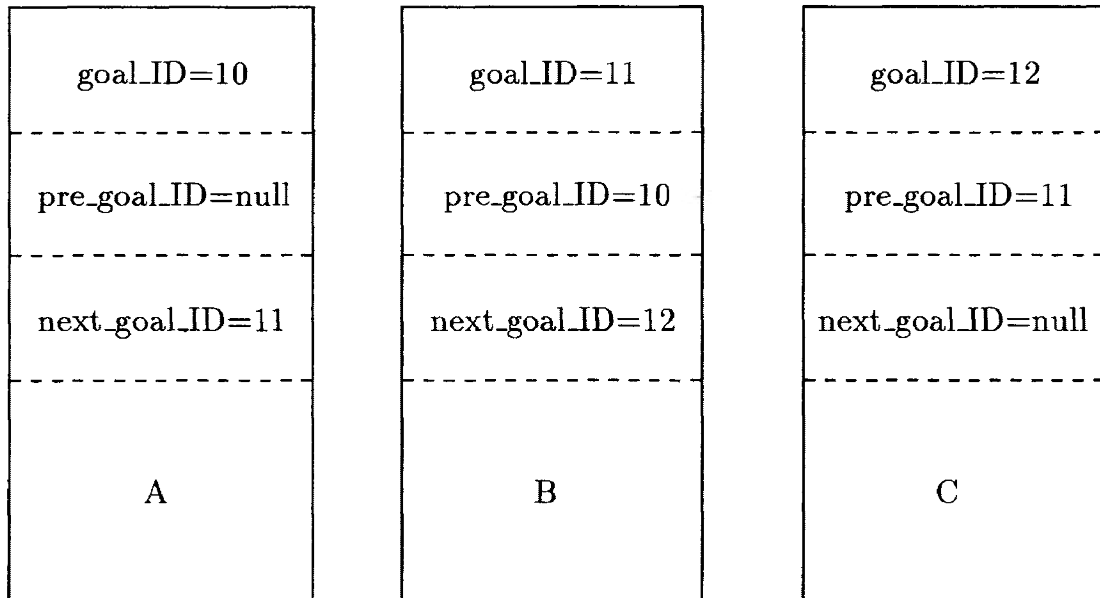


Figure 3.4: Internal goal objects

To illustrate, we assume that the following program *quicksort* is processed:

```
(Q1)      quicksort([X|Xs],Ys) :-
(QG1)            partition(Xs,X,Littles,Bigs),
(QG2)            quicksort(Littles,Ls),
(QG3)            quicksort(Bigs,Bs),
(QG4)            append(Ls,[X|Bs],Ys).
(Q2)      quicksort([],[]).

(P1)      partition([X|Xs],Y,[X|Ls],Bs) :-
(PG1)            X =< Y,
(PG2)            partition(Xs,Y,Ls,Bs).
(P2)      partition([X|Xs],Y,Ls,[X|Bs]) :-
(PG3)            X > Y,
(PG4)            partition(Xs,Y,Ls,Bs).
(P3)      partition([],Y,[],[]).

(A1)      append([],Ys,Ys).
(A2)      append([X|Xs],Ys,[X|Zs]) :-
```

(AG_1) append(Xs,Ys,Zs).

Figure 3.5 illustrates the relationships among the Horn Objects in terms of ID numbers, which are indicated by dotted arrow lines. However, when mapped into the object base, the corresponding relationships are represented by actual pointers in the object base. Figure 3.5 also shows one related table which maintains the necessary information.

3.5 Output Object Base

After all objects have been created and the necessary information has been obtained, the object generator is used to produce the object base. The table handler groups internal objects created into corresponding internal object lists:

1. internal atom object list;
2. internal variable object list;
3. internal function object list;
4. internal argument object list;
5. internal fact object list;
6. internal goal object list;
7. internal rule object list.

Note that, an abstract list which defines the common states and operations of lists is used in the implementation and all other concrete lists are the classes derived from the abstract list class. As soon as an internal object is created, it is appended to the appropriate list with the same type. After all internal objects have been created, the

A
Q_1
Q_2
predicate
horn
GFAIL
QG_1
QG_2
QG_3
QG_4
GSUCC
A
A
quicksort/2
partition/4
P_1
append/3
A_1

Figure 3.5: Example: internal Horn objects

object base is constructed by the output of these internal objects, which define and initialize horn and term objects in the object base.

Chapter 4

THE INFERENCE ENGINE

The standard Prolog inference engine is based on Robinson's resolution [Rob79]. It consists of two major algorithms: unification and control. The same strategy is used in our implementation. However, instead of separating the inference engine from the logic base, an object-oriented inference engine is embedded in every object in the object base. The control aspects of Horn and term objects are implemented by member functions,

4.1 Term Object Sharing

Term representation is the fundamental concern for constructing the object base. Here, we present a new term representation method - *Term Object Sharing (TOS).* In TOS, all terms are represented in their natural way, that is, they are objects from a common base class - term.

TOS takes an object-oriented view of logic terms. It closely resembles the SS scheme in the sense that different instances of the same term share a single term description and differ only in their variable bindings. However, the most important difference is that TOS classifies variables as *local* or *global* dynamically and achieves optimal use of memory. This means that TOS tends to have the advantages of both SS and NSS without any extra cost such as *mode declaration* [War77] or *global flow analysis* [Mel81].

With TOS, all variable objects are local initially. Some variable objects will be classified as global in the run time. In order to construct or select term instances correctly, special care should be taken in handling *lvar* and *func* terms. First, it is possible that a variable value is bound to another variable value in unification. If this happens, we just let the younger value point to the older one rather than the other way round. Thus, there is no danger of leaving dangling pointers when the values of local variables are deallocated upon the callee's completion. In practice, an *age* is attached to every newly created value and an age counter is increased whenever a caller invokes its callee. Let A and B be two values, we say that A is younger than B if A→age is greater than B→age. Needless to say, if two ages are equal, A and B must be created by a single callee to which they belong.

Secondly, when a *func term* is instantiated to a variable, we have to decide whether the instantiation is served as a selector or a constructor. If the instantiation is a constructor, all variables in the func term must be set to global. A simple rule used in TOS is that a func term is classified as a constructor if it becomes a binding of an *older* value (here, the age of the func is determined by choosing any variable inside the func and returning the age of the referred value).

For example, suppose the caller's argument is X and the callee's argument is f(Y), if the value dereferenced from X is free, then we simply bind the value to term f and set variable Y global. From the discussion of value-value binding, it is certain that the dereferenced value from X must be *older* than term f(Y). Therefore, f(Y) is a constructor and Y must be global. Now, exchange caller and callee's arguments and assume that the dereferenced value from X is still free, then a comparison has to be made between the age of the dereferenced value and f(Y). In most cases, this value

is younger than f(Y), so we just let it point to f and leave Y's state unchanged, i.e., X serves as a selector. However, it is also possible that such a value is older than f(Y). For example, consider the following program:

```
?- p(C)

p(A) :- q(A, f(Y)).
q(X, X).
```

Suppose that we have deduced the query p(C) to the current goal q(A, f(Y)), the final pair of terms to be unified in solving the goal is the pair of f(Y) (caller's argument) and X (callee's argument). Although X looks younger than f(Y), yet the actual dereferenced value from X is a free value in C which is carried down by A and X, and was created before f(Y). Thus, Y must be global. In this example, variable X seems like a bridge to direct a constructor to a free value.

4.2 Variable Object Manipulation

The key issue in dealing with term objects is how to manipulate variable objects. The strategy adopted in the inference engine is to let each variable object keep track of its bindings and discard partial binding records as soon as they become useless. To achieve this, each variable object must maintain a stack to store its own updating records. In our implementation, fixed-size blocks of memory are allocated randomly from a pool and linked together to form a cactus stack for a logical variable, and the binding records stored in the stack are called *values* of the variable. In addition to the normal *push* and *pop* operations, the stack provides a value accessing function which returns a referred value in constant time.

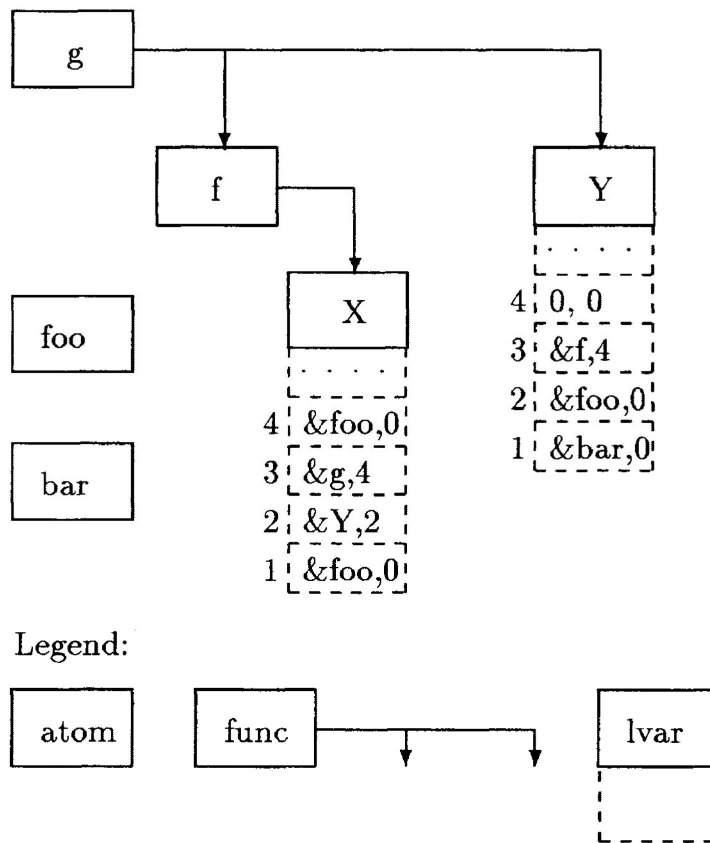


Figure 4.1: A snapshot of term objects in execution

A variable value is defined by a triple: int gref, int lref, and term* bnd, where *bnd* is a pointer to the binding term, lref and gref are integers for dereferencing variables occurred inside the binding term. A value is a *free value*, if its bnd is a null pointer. In fact, a value represents a term instance created in execution. As the term instance may be a structure which involves other variables, these variables must be further dereferenced by either lref or gref according to the status of these variables. For example, Figure 4.1 shows a possible snapshot of term objects in an execution. Suppose that *pg* is a term pointer to function *g*, and we only use one integer to dereference variables. We have the following term instances when different dereferencing integers are passed to function *print*:

pg→print(1): g(f(foo), bar);
pg→print(2): g(f(foo), foo);
pg→print(3): g(f(g(f(foo), _Y)), f(foo));

where _Y stands for a free value of variable Y.

4.3 Term Object Unification

Unification plays a central role during the computation as far as term objects are concerned. *Term object unification* is defined as public member functions in the following classes. Note that the other two operations of dereference and binding are also put in the term class and defined as *virtual*, so that all kinds of term objects are processed uniformly:

term object
data:
int tag;
char* name;
member functions:
constructor();
destructor();
virtual int unification(...);
virtual term* dereference(...);
virtual void binding(...);

arg_list object
data:
link* last;
member functions:
constructor();
destructor();
int unification(...);
...

func object
data:
int vl, vr;
arg_list *arglist;
member functions:
constructor();
destructor();
int unification(...);
...

lvar object
data:
stack s;
/* auxiliary data */
member functions:
constructor();
destructor();
int unification(...);
term* dereference(...);
void binding(...);
/* stack operations */

In order to continue our discussion without too many control details, we assume that both caller and callee have their own arguments objects created from the class *arg_list*. A caller passes three parameters, *ler*, *ger* and arguments to invoke its callee, where *ler* and *ger* are used for accessing terms in the caller's arguments. On the other side, the callee maintains two integer variables *lee* and *gee* for dereferencing its own arguments. Both *lee* and *gee* are increased when the callee is invoked, however, *lee* is decreased at the callee's completion while *gee* is decreased upon backtracking. The unification process starts by the statement:

callee → arguments → unify(lee, gee, ler, ger, caller → arguments);

which will unify in turn each pair of terms in caller and callee's arguments. The function will return 1 if the unification succeeds, 0 otherwise.

Note that, in class term, the member functions for unification, dereference and binding are declared as virtual functions. A virtual function passes through the actual execution to its derived class at runtime, whereas an object of the derived class invokes the operation. For example, whenever an atom object invokes unification, then the actual code to be executed is described as follows:

```
enum {ATOM, INT, FUNC, VAR, SUCCESS, FAIL};

input:
      int lee, int ger, int ler, int ger, term* ter;
output:
      SUCCESS/FAIL;
algorithm:
      begin
         ter = ter→def(&ler,&ger);
         if (this == ter)
            return SUCCESS;
```

```
        else if (ter→tag == VAR)
            begin
                ter→binding(lee, gee, 1);
                return SUCCESS;
            end
        else
            return FAIL;
    end.
```

The last parameter in function *binding* indicates that the *ter*'s value should be trailed. *Trail* is used to undo variable bindings on backtracking [War83].

In the case that a variable object invokes the unification, the operation is accomplished as follows:

- first, dereference the variable object; and deference the callee term object;
- if the variable object has no binding value, assign the callee term object to it; otherwise, activate its binding object to invoke further unification.

4.4 AND/OR Network Search Strategy

Since the definition of the Horn class directly mirrors that of the Horn clause, the control aspect is designed according to the operational semantics of Horn clauses and does not rely on any specific Prolog program. The skeleton of some typical objects are described as follows:

horn object
data:
/* *none* */
member functions:
virtual horn* try(...);

clause object (derived from horn)
data:
char* name;
horn* next;
argl* arg;
member functions:
constructor();
destructor();
virtual horn* try(...);
...

rule object (derived from clause)
data:
goal* first;
goal* last;
member functions:
constructor();
destructor();
horn* try(...);
...

goal object
data:
char* name;
horn* my_horn;
goal* next;
goal* prev;
argl* arg;
member functions:
constructor();
destructor();
goal* prove(...);
...

The major part of the inference engine consists of two member functions, *try(...)* and *prove(...)*, declared in the Horn class and the goal class, respectively. The core code of these functions is shown below. Note that function *try()* is a virtual function which can be invoked through the Horn pointer and is bound dynamically at runtime.

```
enum {FORWARD, BACKWARD}

/* to try if a Horn object can satisfy the goal */

Part I:

input:
        goal* cler, ...
output:
        HSUCC if succeed; return next choice or null if fails
algorithm:
        begin
           if (call_mode == FORWARD)
              begin
                 if (arg && !arg→unify(cler→arg, ...)
                    return next;
                 g = first;
              end
           else
              g = last;

           /* exhaust and chain */
           while (g != GSUCC && g != GFAIL) g = g→prove(...);

           if (g == GSUCC)
              begin
                 /* push a choice into choice_stack if */
                 /* necessary, possible a NULL */
                 return HSUCC;
              end
           else
              begin
```

```
            if (!next) return 0;
            call_mode = FORWARD;
            return next;
        end
    end. { Part I }



/* to prove that a goal object succeeds */

Part II:

input:
    ...;
output
    next goal if current goal succeeds; otherwise previous goal;
algorithm:
    begin
      if (call_mode == FORWARD)
        h = my_horn;
      else
        h = /* popped choice */

      /* exhaust or chain */
      while (h && h != HSUCC) h = h→try(this, ...);

      if (!h)
        begin
          call_mode == BACKWARD;
          return prev;
        end
      else
        return next;
    end. { Part II }
```

The search of an AND/OR graph is performed by the inference engine. A Horn object succeeds if all its goal objects are proved. A goal object succeeds if one of

its matching Horn objects is successful. A global variable *call_mode* indicates the execution state. It can be either FORWARD (computation) or BACKWARD (backtracking). A special *Horn* pointer, HSUCC, is returned when a call to a Horn object succeeds. In forward execution, a *choice_stack* must be maintained in case of failure. A choice is a data object that saves information for possible backtracking. Different from WAM and other conventional models, where a choice is pushed upon a procedure call if the procedure has untried alternatives, a Horn object constructs a choice based upon its successful completion as well as other factors collected in its execution. Typical factors influencing the construction of a choice are the involvement of *cut*, nondeterministic goals, and global variables in the course of execution. Discussion of implementation details is beyond the scope of this thesis.

Chapter 5

OPTIMIZATION AND BUILT-IN PREDICATES

The efficiency of a Prolog system can be greatly improved by applying different optimization schemes. For example, in order to reduce the cost of backtracking, more sophisticated strategies, such as "intelligent backtracking" [Bru78], "selective backtracking" [PP82], have been proposed. The intelligent backtracking scheme attempts to retain partial computations to avoid recomputing them in repetition. The selective backtracking approach tries to analyze the cause of the failure and backtracks directly to the nearest point where the computation would take a new evaluation path. Sometimes it is possible to improve the performance when goals are executed in generator-tester manner, such as the *naive sort problem*. A coroutining optimization can be used in this situation: switching control between several active objects. Another commonly used optimization is to remove tail recursion. It is a technique of replacing the last recursive call with iteration. The purpose of this chapter, however, is not intended to cover all of them, but to illustrate the potential extensibility of our model to implement these schemes. One typical optimization, the first parameter hashing method, will be discussed in detail.

Another subject concerned in this chapter is the method of implementing built-in predicates. To achieve the expressive power of Prolog, some predicates, such as meta-logical predicate, extra-logical predicates and system predicates [SS86] are usually

implemented in Prolog systems, even though they are out of the scope of first order logic. Rather than performing unification, the meta-logical predicates check the states of terms. One example of meta-logical predicates is *var(Term)* which tests if *Term* is a free variable. The extra- logical predicates, such as *read(...)*, *write(...)*, *etc.*, involve in I/O operations whose side effects can not be recovered by backtracking. The system predicate is usually referred to *cut* which is used to reduce the searching space in order to improve the efficiency of a program. Our model provides a uniform interface to allow most of these built-in predicates to be implemented.

5.1 First Parameter Hashing Optimization

The proposed system provides the facilities to perform fundamental computation of Horn clauses in terms of depth first search without screening unnecessary choices, which tend to result in inefficiency. One way to improve the efficiency of the our system is to eliminate those alternatives which can be checked out to be ununifiable before runtime to form optimized OR-chains. This method is called the *first parameter hashing optimization.* The first-parameter hashing optimization is achieved by looking at the type or the value of the first parameter of the current *goal object* and hashing its call to a (partially) deterministic *Horn object* so that useless backtracking is reduced.

A hash table is created for each optimized predicate, in which each slot maintains one entry to a shortened OR chain. Some slots may be empty, and any goal which is hashed into these slots for its OR chain knows that it has no choice to try and it returns a failure immediately. Figure 5.1 illustrates one possible hash table for the

predicate *position/2.*

It is possible that several different first parameters are hashed onto the same value, the situation known as hash collision. If a hash collision occurs, the Horn object involved is appended into the corresponding OR chain, that is, collision resolution by chaining. Sometimes, the first parameter of a Horn clause may be a variable. It implies that the corresponding Horn object should be fit into each optimized OR chain involved. Even though the optimized OR chains are created and can be accessed through the hash tables, the original OR chain is still necessary when the first parameter of the current goal is a variable.

The mechanism described above is implemented by introducing some new objects in the object base and by defining the behaviours of these objects in the inference engine accordingly, as shown in the Part I and Part II:

```
Part I:

/* introducing new objects in the object base */

stub objects: {
      sb_obj_1:    horn_1, next_1;
      sb_obj_2:    horn_2, next_2;
      ...}

hash table 1: {
      entry_1:     sb_obj_1;
      entry_2:     sb_obj_2;
      ...}

hash table 2: {
      ...}

...
```

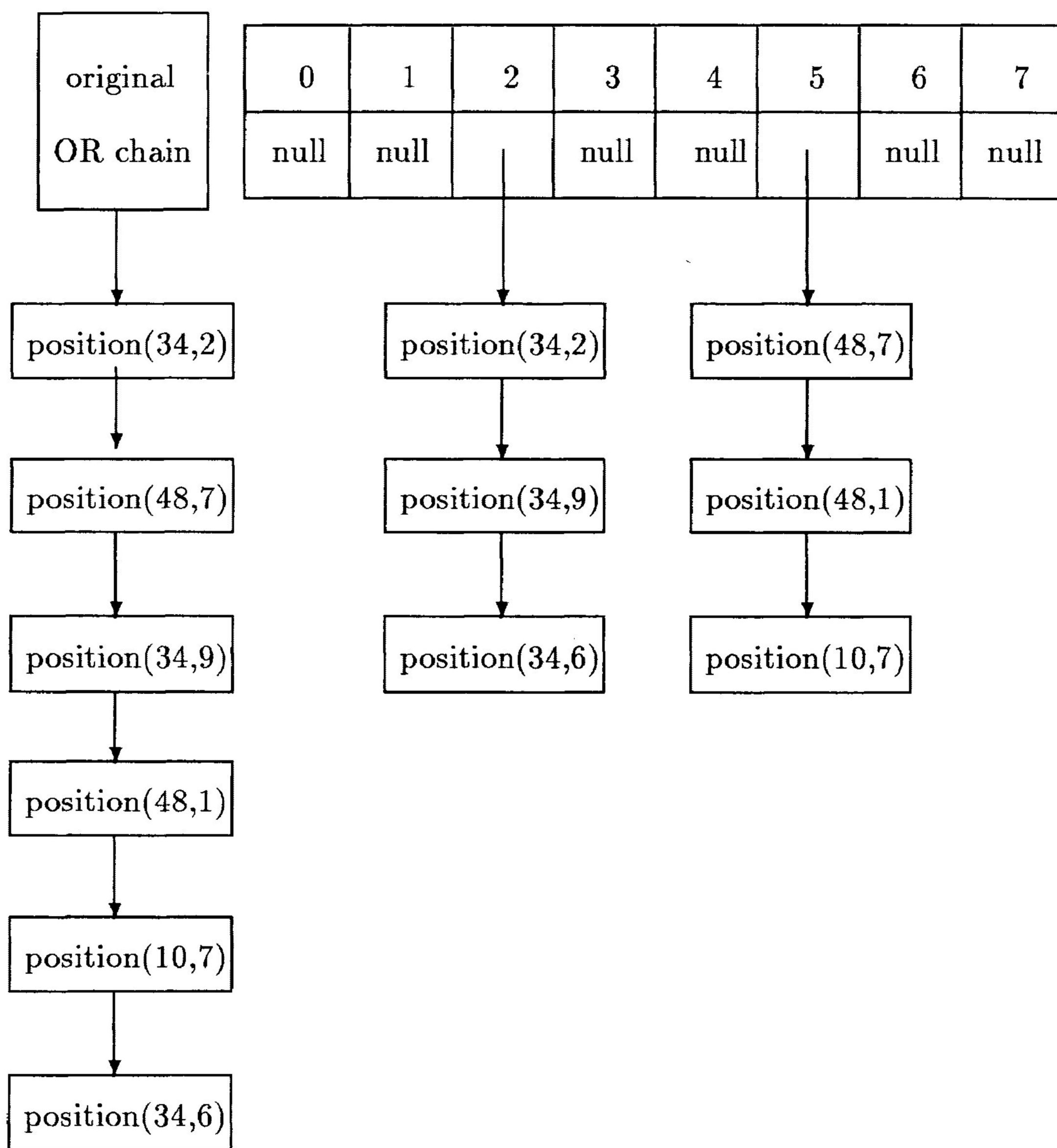


Figure 5.1: An original OR chain and the hash table

root objects: {
 rt_obj_1: hash_table_1, horn_1;
 rt_obj_2: hash_table_2, horn_2;
 ...}

Part II:

/* the definition of root object and stub object */

Root Object: (derived from horn)
date:
horn** hash_table;
horn* var_branch;
int hash_size;
member functions:
horn* try(goal*, ...);
...

Stub Object (derived from horn)
data:
horn* my_clause;
horn* next;
member functions:
horn* try(goal*, ...);
...

```
/* specification for try(...) in stub object */

Input:
        goal* cler, ... ;
Output:
        /* return either an entry of the hash_table or the var_branch with respect
        to the type or the value of the caller's first parameter; */
algorithm:
        begin
           /* hash table operations */
        end.


/* specification for try(...) in root object */

Input:
        goal* cler, ... ;
Output:
        HSUCC, or next choice;
algorithm:
        begin
           horn* h = my_clause→try(cler,...);
           if (h != HSUCC)
              return next;
```

```
    /* modify the top choice if necessary */
    return HSUCC;
end.
```

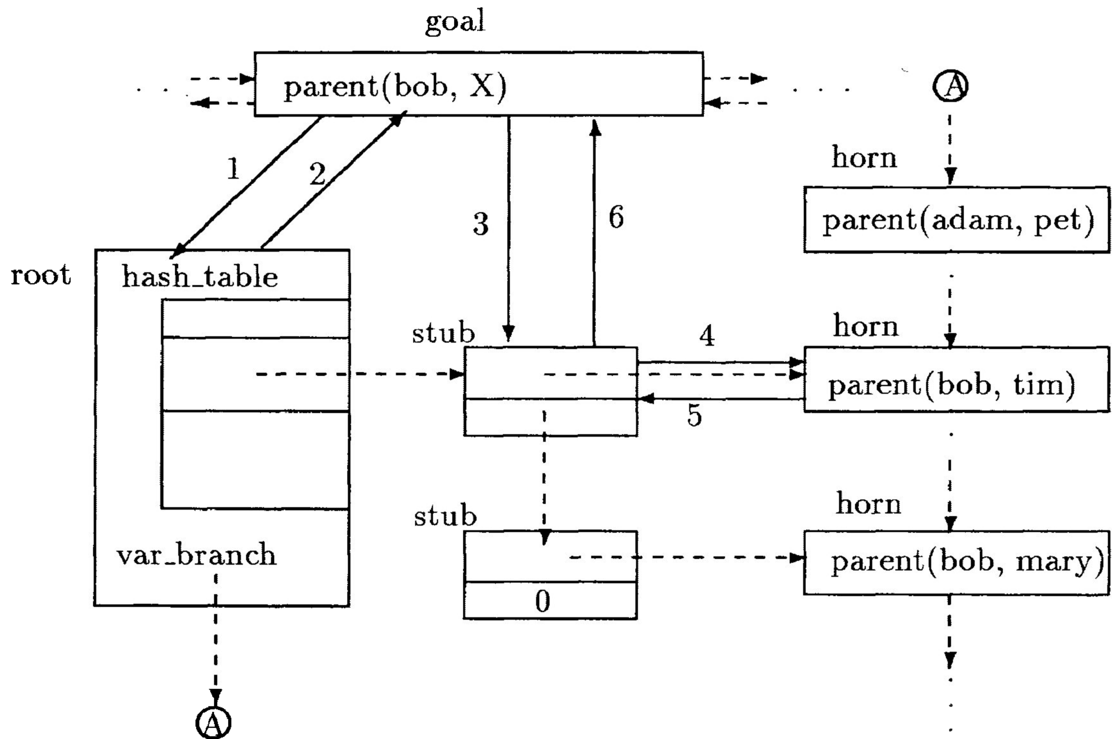
Two classes, *root* and *stub*, both derived from Horn, are used to form new objects in the object base. A *root* object will be created if the first-parameter hashing method is applied to a predicate - a set of *Horn* objects with the same head and arity. Entries to these objects are attached to a set of *stub* objects whose entries are subsequently attached to the *root* object as a hash table. The *root* object then serves as the invocation point of the predicate, that is, it branches a call according to the type or value of the first argument of the caller (goal). As a result, the OR chain pointer in a goal object is replaced by the pointer to its *root* in such an optimization. The *stub* class is used to form a shortened *or-chain* in which all attached *Horn* object entries are hashed into the same slot of the hash table. Obviously, if the first argument of the current *goal* is an unbound logic variable, then the *var_branch* will be used and the execution thread is the same as without the optimization.

For example, Figure 5.2 shows the various objects and their linkage with respect to the program:

```
parent(adam,pet).
parent(bob,tim).
parent(bob,mary).
...
```

The steps to execute the sample program with first-parameter hashing optimization are summarized as follows:

1. *goal* issues a call to its *root*;



legend:

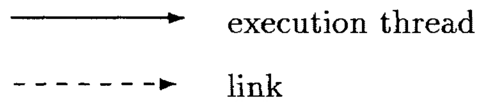


Figure 5.2: An example of first-parameter hashing optimization

2. *root* returns a *stub* pointer found by hashing the first argument of *goal*;

3. *goal* thus invokes a call by referencing the returned pointer;

4. *stub* carries out a real invocation on behalf of *goal*;

5. *horn* returns the execution result;

6. *stub* returns a link leading to another (partially) deterministic branch when the invocation fails or HSUCC if the invocation succeeds. In the latter case *stub* object will modify the top choice on the *choice_stack* for indicating which branch to try on backtracking.

5.2 Implementing Built-in Predicates

Most built-in predicates are treated as Horn objects in our model, even though they are beyond the scope of the Horn clause. Although built-in predicates differ in their functions, they are implemented through a uniform interface as follows:

- a data file, referred as a built-in predicate file, which provides information to specify each built-in predicate, such as its name, arity and its corresponding class name;

- the member function *try(...)*, which defines the behaviour of individual built-in predicate object.

Suppose that we are to implement meta-predicates, such as *var*, *nonvar*, *integer*, etc. These predicates can be grouped into a single class _meta. The implementation

is achieved by two steps. First, for each of built-in predicates to be implemented, it is necessary to put into the built-in predicate file a corresponding triple, *i.e.*, the name of built-in predicate, the number of its parameters and the class name in which it will be defined, so that it can be recognized and processed appropriately by the transformation program. For *var*, *nonvar*, *integer*,..., the triples to be inserted can be described in Table 5.1.

built-in predicate name	number of parameter	class name
var	1	_meta
nonvar	1	_meta
integer	1	_meta
⋮	⋮	⋮

Table 5.1: A Built-in Predicate File

The second step is to redefine the "*try*" function with respect to the functionality of predicates. For example:

_meta Object (derived from horn)
data: ...
member functions: horn* try(goal*, ...); ...

/* specification for *try*(...) in _meta object */

```
Input:
       goal* cler, ...
Output:
       HSUCC if succeeds; otherwise return 0;
Algorithm:
       begin
          switch (*(cler→name))
             begin
                case 'v': /* return HSUCC if cler→arg is an unbound var */
                case 'n': /* return HSUCC if cler→arg is a ground term */
                case 'i': /* return HSUCC if cler→arg is an integer */
                ...
                default: return 0;
             end
       end.
```

After all procedures described above are done, a Prolog program with new built-in predicates can now be executed under the Prolog system. As far as the transformation program is concerned, it read the built-in predicate file into a corresponding built-in predicate table so that the new built-in predicates can be processed. The built-in predicate table with above newly implemented predicates is shown in Table 5.2.

name	arity	index
is	2	0
!	0	1
\==	2	2
=\=	2	2
==	2	2
>=	2	2
<=	2	2
var	1	3
nonvar	1	3
integer	1	3
⋮	⋮	⋮

Table 5.2: The Built-in Predicate Table

With the class names of new built-in predicates, the transformation program also constitutes following code in the object base so that built-in objects can be referenced and linked by corresponding goals.

```
...

_meta _p_meta;
_bp[] = {
   &_p_is,
   &_p_cut,
   &_p_comp,
   &_p_meta,
   ...
};
```

A goal which is found to have a match in the built-in predicate table is processed in the same way as other goals, except that it links to a built-in Horn object. For example, in program:

```
relation(X,Y) :-
        friend(X,Y),
        var(X),
   ...
```

the goal object corresponding to *var(X)* has a linkage to *_bp[2]* which acts as its matching Horn object.

With the interface, even some other special built-in predicates can also be put into our proposed system in the same way. For example, the graphic-oriented debugging operations can be implemented as a set of built-in predicates to provide a useful tool for users.

Chapter 6

CONCLUSION

This thesis reveals a fairly original view of modelling Prolog which is suitable as the framework to design and implement a large variety of optimization schemes and other extensions. It differs from conventional models by simulating Horn clauses as collaborating objects instead of mapping them into passive procedures. In particular, the object-oriented model is integrated by the object base and inference engine. The object base represents the components and logic relationships of a Prolog program in terms of AND/OR graph. The transformation program has been developed to generate the object base. The transformation program decomposes a Prolog program into lexical elements such as atoms, functions, predicates, *etc.*, which are then represented by internal objects. The most important feature of this approach is that the object base is generated only according to the relationships specified by the Prolog program. The inference engine, which is embedded in the object base, provides the control details to make objects act in the same way as their corresponding components in the Prolog program. It enables objects to perform unification and to do backtracking as a computation encounters failure. One of the advantages is that the inference engine classify local/global variables dynamically to achieve optimal use of memory. The scheme of first parameter optimization is also discussed to show the extensibility of the proposed system. Furthermore, an interface is implemented to allow the proposed system to introduce new built-in predicates in a uniform manner.

An experimental object-oriented Prolog system, LU-Prolog, has been developed.

LU-Prolog consists of a transformation program and an executive system (inference engine). The transformation program (about 2,000 lines of C++ code) converts a given Prolog program into the object base, and the executive system (about 1,000 lines of C++ code) makes objects collaborate on a query and implements term unification.

Several typical programs, such as queen_8, nrev_30, qsort_50, *etc.*, were tested under LU-Prolog system and some measurements are shown in Table 6.1. Times was measured on a Sparcstation IPC without either counting system time or applying optimization, such as tail recursion optimization or the first parameter hashing. LU-Prolog runs twice as fast as C-Prolog for the same test programs.

Program	Time(in sec.)	variables	Global var's	values
qsort_50	0.05	23	8	2036
queens_8	0.43	26	7	114
hanoi_12	0.91	6	0	0
nrev_30	0.08	10	3	1861
nrev_30	0.1	10	3	900

Table 6.1: Some measurements for TOS

In Table 6.1, the third column indicates the number of variables in each program, the forth column shows the number of *global* variables recognized dynamically and the final column gives the total number of values remained in stacks when the first solution is found. The statistics show that the scheme has a significant effect on the optimized use of memory by collecting the memory of local variables promptly. For example, in Table 6.1, the difference between two nrev_30 programs is that the

second one has a "*cut*" in the program, therefore, more *local* variable values might be discarded in execution.

The term object sharing (TOS) in our model tends to combine the advantages of SS and NSS without any extra cost such as mode declaration or global flow analysis, and it can be used in either compiler or interpreter implementations. Another advantage of TOS (not discussed in this thesis) is that infinite structures [Col82] can be handled efficiently.

Although TOS reveals a promising term representation method, some aspects requiring further discussion. For example, TOS treats lists as normal nested functions. If a list involves N elements, then roughly 3N objects will be created (N element objects, N function objects and N argument list objects). Since list is a common and frequently used data structure in Prolog programs, a special design of list class might be necessary. Another problem deals with dynamic data - term objects created from I/O. The unification process should be expanded to handle term object arrays and dynamic term objects consistently. More programs need to be tested to compare the memory utilization issue between LU-Prolog and the conventional Prolog programs.

Obviously, under the current running environment, the future work of this project is to improve LU-Prolog with more optimization mechanisms, and to make further comparison of the performance between LU-Prolog and other Prolog systems. Since optimization schemes can usually have a dramatic impact on the efficiency of a Prolog system, it is expected that LU-Prolog will be more competent in terms of its running speed.

The object-oriented Prolog model broaches more topics for further investigation. First, the transformation program can be implemented by Prolog itself, and thus it

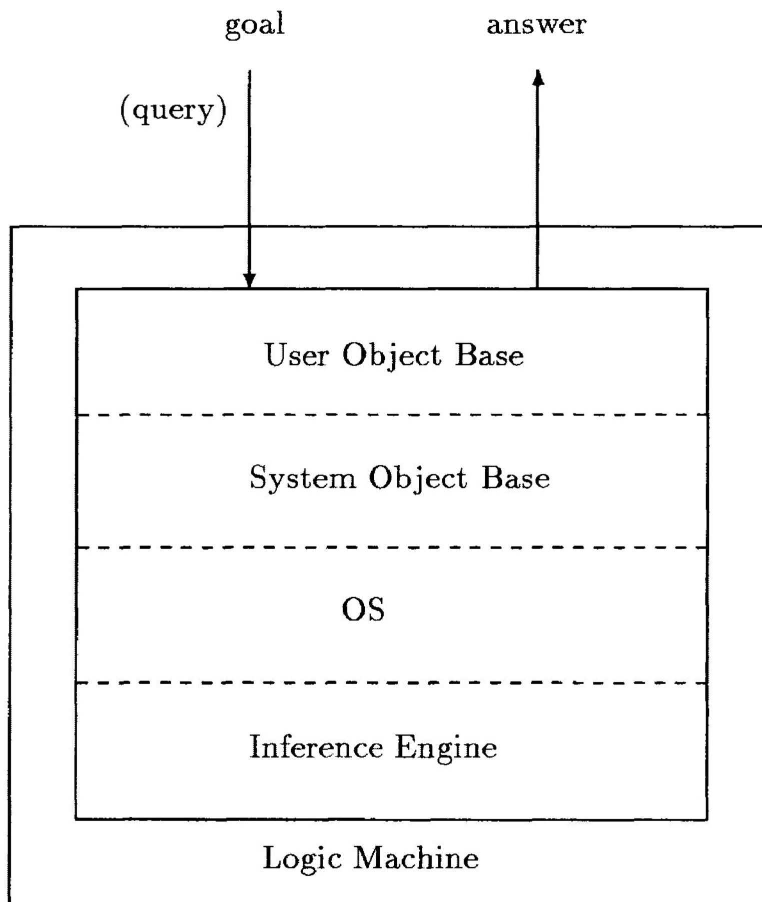


Figure 6.1: General architecture of object-oriented Prolog machine

can be integrated as the part of the Prolog system, known as meta-Prolog. Secondly, the executive system should be extended to cope with dynamic objects created during execution. Thirdly, as the inference engine constructs a choice upon completion instead of invocation of a Horn object, we need more tests to see how such strategy influences the execution performance. The next area of our research is to explore the possibility of implementing the inference engine in an object-oriented hardware framework. Figure 6.1 shows imaginary architecture of such a logic machine.

Another area worth further investigation is the development of a parallel Prolog system based on the model presented in this study. Since components of Prolog programs are all represented by self-contained objects, it tends to be much easier for LU-Prolog to explore parallelism than any conventional Prolog system.

The work described here is still in its infancy. At present, LU-Prolog is not efficient enough to compete with current commercial Prolog compilers, however, it does provide a new and comprehensive methodology which may lead to the design of a very efficient logic machine. The simplicity of the inference engine, as well as the object approach to represent the components of Prolog programs, forms a substantial basis to develop an object-oriented hardware which could finally dispel all the claim to the inefficiency of Prolog systems. Moreover, the object-oriented hardware allows the system to be driven solely by the logic relations with respect to the Prolog programs involved, rather than driven by the flow of conventional instructions. This may lead to a breakthrough away from reliance on a Von Neumann architecture.

Bibliography

[Apt92] K. R. Apt. Why the occur-check is not a problem. In *Proceedings of Programming Language Implementation and Logic Programming*, pages 69–86. Springer-Verlag, 1992.

[Bee88] J. Beer. The occur-check problem revisited. *The Journal of Logic Programming*, 5:243–261, 1988.

[BM72] R. S. Boyer and J. S. Moore. The sharing of structure in theorem proving programs. In *Machine Intelligence 7 (B. Meltzer and D. Miche, eds.)*, pages 101–116. Edinburgh University Press, 1972.

[Bru76] M. Bruynooghe. An interpreter for predicate programs: Part 1. Technique Report CW 16, Katholieke Universiteit Leuven, 1976.

[Bru78] M. Bruynooghe. Intelligent backtracking for an interpreter of horn clause logic programs. In *Mathematical Logic in Computer Science (Domolki, B. and Gergely, T. eds.)*, pages 215–258. North-Holland, Amsterdam, 1978.

[Bru82] M. Bruynooghe. The memory management of prolog implementation. In *Logic Programming (Clark K. L. and Tarnlund S-A. eds.)*, pages 83–98. Academic Press, 1982.

[Col82] A. Colmerauer. Prolog and infinite trees. In *Logic Programming (Clark K. L. and Tarnlund S-A. eds.)*, pages 45–66. Academic Press, 1982.

[Con87] J. S. Conery. *Parallel execution of logic programs.* KLUWER ACADEMIC PUBLISHERS, 1987.

[HS84] S. Haridi and D. Sahlin. Efficient implementation of unification of cyclic structures. In *Implementations of Prolog (Campbell, J. A. ed.)*, pages 234–249. Ellis Horwood Ltd., 1984.

[Kow79] R. Kowalski. *Logic for Problem Solving.* Elsevier North-Holland, Armsterdam, 1979.

[Mel81] C. S. Mellish. Automatica generation of mode declarations in prolog programs. In *Paper presented at Workshop on Logic Programming.* Long Beach, Los Angeles, 1981.

[Mel82] C. S. Mellish. An alternative to structure sharing in the implementation of a prolog interpreter. In *Logic Programming (Clark K. L. and Tarnlund S-A. eds.)*, pages 99–106. Academic Press, 1982.

[Mel85] C.S. Mellish. Some global optimizations for a prolog compiler. *Logic Programming*, 1:43–66, 1985.

[Nil71] N.J. Nilson. *Problem-solving methods in AI.* Standford Research Ins. Menlo Park, California, 1971.

[Pla84] A. D. Plaisted. The occur-check problem in prolog. *New Generation Computing*, 2:309–322, 1984.

[PP82] L. M. Pereira and A. Porto. Selective backtracking. In *Logic Programming (Clark K. L. and Tarnlund S-A. eds.)*, pages 107–114. Academic Press,

1982.

[RD92] P. V. Roy and A. M. Despain. High-performance logic programming with the aquarius prolog compiler. *IEEE Computer*, 25(1), January 1992.

[Rob79] J. A. Robinson. *Logic: Form and Function - the Mechanization of Deductive Reasoning.* North-Holland, Amsterdam, 1979.

[SS86] L. Sterling and E. Shapiro. *The Art of Prolog.* The MIT Press, 1986.

[TS82] J.P. Tremblay and P.G. Sorenson. *An implementation guide to compiler writing.* McGRAW-HILL BOOK COMPANY, 1982.

[TW84] E. Tick and D.H.D. Warren. Towards pipeline prolog processor. *New Generation Computer*, 1984.

[USL93] A. Keune U. Schreiweis and H. Langendorfer. A intergated prolog programming environment. *ACM SIGPLAN Notices*, 28, 1993.

[War77] D. H. D. Warren. Logic programming and compiler writing. Technique Report DAI 44, University of Edinburgh, 1977.

[War83] D. H. D. Warren. An abstract prolog instruction set. Technique report, SRI International, AI Center, 1983.