INFORMATION TO USERS

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

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

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

Oversize materials (e.g., maps, drawings, charts) are reproduced by sectioning the original, beginning at the upper left-hand corner and continuing from left to right in equal sections with small overlaps. Each original is also photographed in one exposure and is included in reduced form at the back of the book.

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

Yalçınalp, L. Ümit, Ph.D.

Case Western Reserve University, 1991
META-PROGRAMMING FOR KNOWLEDGE BASED SYSTEMS IN PROLOG

by

L. ÜMIT YALÇINALP

Submitted in partial fulfillment of the requirements for the Degree of Doctor Of Philosophy

Thesis Advisor: Dr. Leon S. Sterling

Department of Computer Engineering and Science
CASE WESTERN RESERVE UNIVERSITY
August, 1991
CASE WESTERN RESERVE UNIVERSITY

GRADUATE STUDIES

We hereby approve the thesis of

L. Ümit Yalcinalp

candidate for the Ph. D. in COMPUTER SCIENCE
degree.*

Signed:  

Sterling  
(Chairman)

Date  April 10th, 1991

*We also certify that written approval has
been obtained for any proprietary material
contained therein.
I grant to Case Western Reserve University the right to use this work, irrespective of any copyright, for the University's own purposes without cost to the University or to its students, agents and employees. I further agree that the University may reproduce and provide single copies of the work, in any format other than in or from microforms, to the public for the cost of reproduction.

[Signature]
META-PROGRAMMING FOR
KNOWLEDGE BASED SYSTEMS IN PROLOG

Abstract

by

L. ÜMIT YALÇINALP

Meta-programming is programming by treating other programs as data. Meta-programming languages provide a high level abstraction for explicit representation of other languages and their control mechanisms. Using meta-programming techniques, it is easy to define, extend and alter a model of computation of a language.

Tools for representing special purpose domain languages and inference mechanisms are crucial for developing knowledge based systems. This thesis concerns the use of meta-programming for this purpose, for building knowledge based systems. It specifically illustrates the methods and tools for building expert system shells by identifying and extending relevant meta-programming techniques in Prolog.

Prolog's representation, namely Horn clauses, can be used directly as the domain language to represent the rules for building knowledge based systems. However, this approach is very limited because of a hardwired representation of a domain language and its interpretation. In Prolog, we can represent and access the language constructs of the language and also represent its interpretation within itself by meta-programming. A meta-interpreter provides an explicit representation based on an abstraction of a computational model. Using meta-interpreters as a basis for writing special purpose shells is a well known
method. The selected model of computation is then extended by techniques to write special purpose interpreters for revealing, altering and extending Prolog's representation. This thesis studies known meta-interpreters and demonstrates techniques to enhance them as expert systems shells. The extensions in the literature are based on the well known four clause meta-interpreter called vanilla for modelling Prolog's computation in Prolog. The novel concept in this thesis is a layered abstraction where the computation is described in two layers. The layered interpreter provides a flexible framework as the computation can be tailored for a single line or multiple lines of reasoning, unlike extensions to vanilla interpreter. This abstraction allows us to model failure and negation, and also cuts explicitly in a single interpreter.

This thesis demonstrates different applications where the layered interpreter is used as a basis, namely expert system shells for explanation and uncertainty. We also illustrate the relevant techniques for writing such shells. As we have a proper basis, we obtain a shell which can explain successes, failures and negation and also illustrates the effects of cuts in the computation. For uncertainty, the shell can be tailored to accommodate different belief calculi for combining multiple evidence, such as point valued certainties or multi-valued Dempster-Shafer theory.

Using meta-programming is not limited to extending Prolog or writing shells for Prolog. We also demonstrate the development of a special purpose language for ordinal reasoning developed within Prolog and show how it can be integrated with Prolog's representation.

Prolog is not the only logic programming language which provides meta-programming. There are emerging approaches in other logic programming languages for meta-programming. We compare Prolog's capabilities and deficiencies with respect to these languages and show a naming technique which was useful for developing the shells.
To Aysel and Emin,
my parents

and

To İşık,
my sister
ACKNOWLEDGEMENTS

The life of a writer is tragic: the more we advance, the farther there is to go and the more there is to say, the less time there is to say it.

Gabriella Roy

The deadline for submitting this document is reached. As I tried to accomplish the task of writing in a language that I am still learning to master, I would like to thank Dr. Sterling, Dr. Özsoyoğlu, Dr. Ernst and Dr. Solow both for being in my committee and for their patience in reading and correcting my thesis.

Just like others before me, I am indebted to many people throughout this open-ended learning experience.

The first person I would like to thank is my friend and advisor Leon Sterling. His encouragement, optimism and support for research is a goal that I wish to achieve in my future work working with others. Leon has always inspired me with his optimistic approach to problems and his honesty with me about academics. His guidance as a friend and an advisor is irreplaceable. I also thank him for creating a creative environment, so called the COMPOSERS group, where we could share, exchange and develop our ideas in a collective framework. As one of his tenured students, I am also grateful for being able to go to many international conferences which expanded my academic horizons.

Consequently, thanks to the COMPOSERS Logic Programming group at Case Western Reserve University, for both creating a fruitful environment to work and providing me the first lessons regarding group interaction and participation in projects. I thank Arvind Bansal for being a very honest and straightforward colleague and Marc Kirschenbaum for critiquing a part of my thesis and some of my papers. Also thanks to the current COMPOSERS, Venkatesh Srinavasan, Leo Shulman and David Schwartz for making this lab a very enjoyable place to work. I hope they will not abandon drinking good coffee when I leave.
Thanks to one of the wonders of the universe, namely the sendmail program for enabling me to keep in touch with the other researchers in the field. I am grateful to my colleagues, Frank Van Harmelen for sending us his excellent thesis about meta-level reasoning, Stefania Costantini for various discussions regarding RP, and V.S. Subrahmanian and Raymond Ng for providing various technical reports and insights about uncertainty reasoning.

I thank Gil Yehuda for providing the simple expert system rules that we used in Chapter 6 and Afzal Tarar for his experiments with vanilla interpreter which prompted us to explore a better abstraction.

I owe a lot to my friends for their support. I will miss the discussions about life, universe and everything with Djamal Belhandouz and exploring Cleveland with him and Jon Cook. The postings to the funalpha mailing list whose members include Aydin Edger, the "boys", and Madhavi provided a colorful atmosphere in our department. My flatmate Bill Magner has been a good listener of my frustrations over the years.

Last, but not the least, I thank Austin Gurney for his endless support and for his solid trust in my abilities and intellect. Besides, this thesis would have never been written without his routine a-la-taxi service between home and office almost every night for the entire spring semester.
Contents

1. Introduction .................................................................................. 1
  1.1 Goals ......................................................... 1
  1.2 Basic Terminology and Assumptions .......................... 3
    1.2.1 Logic Programming ................................. 4
    1.2.2 Software Engineering ............................... 5
    1.2.3 Artificial Intelligence ................................. 7
  1.3 Results ................................................................................. 9
  1.4 Outline of the Thesis ......................................................... 10

2. Meta-Programming .................................................................... 11
  2.1 Defining Meta-Programming ............................................. 11
  2.2 Issues in Representation ............................................... 14
    2.2.1 Naming .......................................................... 15
    2.2.2 Control of Inference ..................................... 18
  2.3 Using Meta-Programming for Developing Knowledge Based Systems ............................................. 22

3. Approaches to Meta-Programming in Logic Programming .... 29
  3.1 Amalgamated Languages .................................................. 29
    3.1.1 Bowen/Kowalski ............................................. 29
    3.1.2 Prolog ............................................................. 32
      3.1.2.1 Prolog's Features ................................... 32
      3.1.2.2 Criticisms Regarding Prolog ..................... 36
  3.2 Bilingual Languages: Gödel .............................................. 41
    3.2.1 Non-Ground Representation ................................ 42
    3.2.2 Ground Representation .................................... 45
    3.2.3 Discussion ...................................................... 47
  3.3 Overlapping Languages: RP .............................................. 48
3.3.1 Reflective Prolog ........................................... 48
3.4 Altering the Object-Level Computation Model by Object Level
     Interpreters .................................................... 54
3.4.1 Gallaire/Lasserre ........................................... 55
3.4.2 Van Harmelen's Partial Reflection Interpreter .............. 57

4. Identifying Skeletons and Techniques for Meta-Programming 62
   4.1 Meta-Interpreters as Skeletons .............................. 62
   4.2 Techniques for Enhancing the Skeletons ..................... 65
       4.2.1 Collect ...................................................... 65
       4.2.2 Calculate ................................................... 66
       4.2.3 Extending the Object Language .......................... 67
       4.2.4 Altering the Object Level Interpretation .............. 68
   4.3 Different Abstractions of Prolog Computations .............. 69
       4.3.1 Single Line Reasoning ................................... 69
       4.3.2 Multiple Lines of Reasoning .............................. 71
           4.3.2.1 Layered Meta-Interpreters ......................... 74
       4.3.3 Extensions for Multiple Lines of Reasoning .......... 78
           4.3.3.1 Handling Cuts ....................................... 78
           4.3.3.2 Extensions for Additional Constructs ............ 82
       4.3.4 OLD Resolution and Prolog ............................... 84
   4.4 Discussion .................................................... 89

5. Extending Prolog by a Simple Embedded Language .......... 92
   5.1 A Simple System for Ordinal Reasoning in Prolog ........ 92
   5.2 Representation and Interpretation of an Embedded Language
       for Ordinal Reasoning ......................................... 93
   5.3 Embedded Languages and Prolog Interpreters ................ 95
   5.4 Discussion .................................................... 98

6. A Shell for Explaining Prolog Programs ..................... 100
   6.1 Explanation Facility for Expert Systems .................. 100
       6.1.1 The Purpose and Content of Explanations .......... 100
6.1.2 Types of Explanation .............................. 104
6.2 Developing Explanation Shells for Prolog Based Expert Systems 105
  6.2.1 Techniques for How Explanations .................. 106
  6.2.2 Techniques for Why Explanations .................. 106
  6.2.3 Techniques for Whynot Explanation ................. 112
  6.2.4 Other Types of Explanation ....................... 113
6.3 XPL Shell for Explanation ............................ 113
  6.3.1 Overview of XPL ................................. 113
  6.3.2 Extended Layered Interpreter for Constructing Deriva-
       tion Trees .................................. 116
  6.3.3 The Explainer .................................. 122
  6.3.4 Examples ..................................... 133
  6.3.5 Discussion ..................................... 141

7. A Shell for Uncertainty Reasoning in Prolog ................. 143
  7.1 Reasoning with Uncertainty and Prolog ................ 143
  7.2 Issues for Reasoning with Uncertainty in Prolog ........ 146
  7.3 Layered Interpreter for Uncertainty Reasoning ........ 148
  7.4 An Example for Adjusting Reasoning .................... 153
  7.5 Final Remarks ..................................... 154

8. Conclusions and Directions for Future Work ................. 158
  8.1 Achievements ...................................... 158
  8.2 Conclusions ....................................... 159
  8.3 Future Work ....................................... 160

Bibliography ............................................. 162
List of Figures

<table>
<thead>
<tr>
<th>Number</th>
<th>Figure Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.1</td>
<td>The Vanilla Interpreter</td>
<td>34</td>
</tr>
<tr>
<td>3.2</td>
<td>Representation of append/3 in Gödel</td>
<td>42</td>
</tr>
<tr>
<td>3.3</td>
<td>Meta-Level Representation of append/3 in Gödel</td>
<td>43</td>
</tr>
<tr>
<td>3.4</td>
<td>Vanilla Interpreter in Gödel</td>
<td>44</td>
</tr>
<tr>
<td>3.5</td>
<td>Reconstruction of an RP program in Prolog</td>
<td>53</td>
</tr>
<tr>
<td>3.6</td>
<td>Van Harmelen's Partial Reflection Interpreter</td>
<td>59</td>
</tr>
<tr>
<td>4.1</td>
<td>Constructing a Proof by Collect Technique</td>
<td>66</td>
</tr>
<tr>
<td>4.2</td>
<td>Calculating the Maximum Depth in the Proof Tree</td>
<td>67</td>
</tr>
<tr>
<td>4.3</td>
<td>Adding Disjunction</td>
<td>67</td>
</tr>
<tr>
<td>4.4</td>
<td>Result Collecting Interpreter</td>
<td>73</td>
</tr>
<tr>
<td>4.5</td>
<td>Basic Layered Interpreter</td>
<td>77</td>
</tr>
<tr>
<td>4.6</td>
<td>Enhancements to the Layered Interpreter by Adding Constructs</td>
<td>83</td>
</tr>
<tr>
<td>4.7</td>
<td>Warren's Skeleton for Developing an OLD Interpreter</td>
<td>86</td>
</tr>
<tr>
<td>4.8</td>
<td>Warren's OLDT Interpreter</td>
<td>87</td>
</tr>
<tr>
<td>5.1</td>
<td>Rules for Evaluating a Student</td>
<td>93</td>
</tr>
<tr>
<td>5.2</td>
<td>Re-formulating the Rules for Evaluating a Student</td>
<td>94</td>
</tr>
<tr>
<td>5.3</td>
<td>A Simple Embedded Language for Ordinal Reasoning</td>
<td>94</td>
</tr>
<tr>
<td>5.4</td>
<td>Extended Embedded Language</td>
<td>97</td>
</tr>
<tr>
<td>5.5</td>
<td>Meta-Level Interpreter for the Extended Embedded Language</td>
<td>97</td>
</tr>
<tr>
<td>5.6</td>
<td>Skeleton for the Embedded Language and Prolog</td>
<td>98</td>
</tr>
<tr>
<td>6.1</td>
<td>Technique for Collecting the Current Path</td>
<td>110</td>
</tr>
<tr>
<td>6.2</td>
<td>The Architecture of XPL Shell</td>
<td>115</td>
</tr>
<tr>
<td>6.3</td>
<td>Example derivation tree of a failure due to a cut</td>
<td>120</td>
</tr>
<tr>
<td>6.4</td>
<td>Example derivation tree for failure due to non existing axioms</td>
<td>122</td>
</tr>
<tr>
<td>6.5</td>
<td>Code for the Explainer: Top Level</td>
<td>125</td>
</tr>
<tr>
<td>6.6</td>
<td>Code for the Explainer: One Level Explanations</td>
<td>126</td>
</tr>
<tr>
<td>Section</td>
<td>Page</td>
<td></td>
</tr>
<tr>
<td>------------------------------------------------------------------------</td>
<td>------</td>
<td></td>
</tr>
<tr>
<td>6.7 Code for the Explainer: Identifying Nodes</td>
<td>127</td>
<td></td>
</tr>
<tr>
<td>6.8 Failure Handler</td>
<td>128</td>
<td></td>
</tr>
<tr>
<td>6.9 Small Expert System for Selecting Graduate Schools</td>
<td>134</td>
<td></td>
</tr>
<tr>
<td>6.10 Data for Graduate Schools and Students</td>
<td>135</td>
<td></td>
</tr>
<tr>
<td>7.1 Layered Meta-Interpreter for Uncertainty Reasoning</td>
<td>150</td>
<td></td>
</tr>
<tr>
<td>7.2 Hooks to the Top Layer for Specification of Belief Calculus</td>
<td>152</td>
<td></td>
</tr>
<tr>
<td>7.3 Specification of Belief Calculus</td>
<td>154</td>
<td></td>
</tr>
<tr>
<td>7.4 Specification of D-S Calculus</td>
<td>155</td>
<td></td>
</tr>
</tbody>
</table>
Chapter 1

Introduction

1.1 Goals

We are interested in the engineering of knowledge based systems and expert systems shells in this thesis. Our approach is to use a particular programming approach, which simplifies the development of such systems by utilizing the easy representation of knowledge and inference, called meta-programming.

The word "meta" refers to aboutness [Per88]. The study of meta-level reasoning and meta-level architectures has been quite popular in the Artificial Intelligence community. The interest has emerged in areas ranging from reasoning about knowledge, reflection and introspection in intelligent agents, reasoning about beliefs, reasoning with uncertainty, to develop new languages and tools for knowledge representation, knowledge based systems and knowledge engineering environments.

Although we will discuss its definition in detail later, meta-programming, can be considered as programming about programming. Its use is by providing expressive power for representation of languages and their control behaviour for programming. It is clear that tools for representing languages, knowledge and inference are crucial for developing knowledge based systems. Meta-programming is a means for explicit specification of control and interpretation of languages. Therefore, it can be used in building architectures for knowledge based systems based on meta-level reasoning.

The purpose of this thesis is to identify the approaches for meta-programming for constructing knowledge base systems in Prolog. In doing so, we will look at various aspects of using meta-programming in logic programming languages and some of the aspects of meta-level reasoning using Prolog. Particularly, we are interested in using Prolog for building systems where Prolog
is used as the domain language or is extended as a specific language. We also investigate the techniques to develop expert systems shells based on Prolog and its extensions.

Prolog is widely recognized as a good language for writing simple rule-based expert systems. There are two major factors. First, the primary language constructs, Horn clauses, are essentially rules. Second, Prolog's built-in backward chaining interpreter can be used as the inference engine. However, writing rule-based systems directly in Prolog has limitations due to the 'hardwiring' of both a particular type of representation and a particular inference strategy. Further, the language does not provide certain functionalities of an expert system shell, such as tools for interacting with the user, explanation, knowledge acquisition, debugging, reasoning with uncertainty.

Three different approaches have been suggested to overcome Prolog's limitations for building expert systems [CFL+88]. The first method is to build the desired extensions within the underlying Prolog engine at the implementation level of a Prolog compiler or interpreter. The second method is to translate a knowledge-based system written in a special language to Prolog by using a special compiler. The third method is to exploit the meta-programming facilities of Prolog. Clearly the first method is the most efficient yet the most inflexible. Developing the desired extensions within Prolog is the easiest method and the most flexible one, since it is not necessary to change the underlying Prolog system.

This thesis advocates this third method, by viewing meta-programming as a paradigm for writing knowledge based systems in Prolog. Our approach is not to define yet another logic programming language for meta-programming. Our primary focus is developing and using meta-programming techniques for building knowledge based systems for Prolog, by showing the relevant techniques that should be utilized for developing expert system shells, or tools for knowledge engineering environments within the language. We discuss the constructs that are already available in the language, or proposed as extensions to the language for meta-programming with a software engineering approach, by utilizing meta-programming for building such systems. In doing so, we will
also study meta-programming within logic programming, other logic programming languages for meta-programming and compare the Prolog's strengths and weaknesses with other approaches.

Our goals in this thesis can be summarized as follows.

- defining meta-programming.
- discussing the issues about representation in meta-programming.
- illustrating the use of meta-programming in Prolog, its purpose and its limitations.
- clarifying the other approaches to meta-programming in logic programming and comparing them with the capabilities of Prolog.
- showing how to utilize meta-programming in Prolog for developing expert systems shells by identifying specific techniques in building such systems.
- providing different applications of meta-programming in Prolog for developing and extending expert systems shells.

1.2 Basic Terminology and Assumptions

Our subject of this thesis is the development of knowledge based systems by using meta-programming techniques in Prolog. Our interest is primarily the software engineering of knowledge based systems by using meta-programming in logic programming. Prolog language and meta-programming are our connection to logic programming. Building knowledge based systems and expert system shells are our focus in Artificial Intelligence. The contents, therefore, bridge between these three fields. The chapters will describe the relevant terminology in detail. However, let us shortly introduce some of the basic terminology which will be used throughout the thesis in these fields.
1.2.1 Logic Programming

We will discuss logic programming languages and meta-programming in logic programming in this thesis. Although meta-programming concepts will be introduced later, the reader is assumed to be familiar with the basic principles of logic programming, namely the Horn clause syntax, the principles of SLDNF resolution, unification and the computational model of Prolog. The unfamiliar reader is referred to [Llo87, SS86] for the basics.

We will discuss expert systems shells where Prolog is used or extended as the domain language for the system. Therefore, the representation of Prolog’s computational model and search is central in this thesis. The following definitions regarding search will be used in this document.

Given a logic program $P$ and a goal as a query $G$, a search space for $G$ is defined as all possible ways of applying the inference rules to the set of clauses in $G$ and to the clauses derived from them [Kow79]. A search tree is a representation of the search space which is traversed in order to find a solution for $G$. Although there is only one search space, there might be different search trees depending on the goal selection from the resolvents in the refutation\(^1\). A search strategy is used to search this search space and determines the shape of the tree as it describes the selection of the goal from the set of resolvents and the order of the clauses from $P$ that will be used to resolve the goal that is selected from the resolvents. For example, two different methods for searching the search tree is using depth-first or breath-first search.

The search tree can be represented in two different ways, either by an AND-OR representation or an OR representation.

In an AND-OR representation, the root of the tree is labelled by $G$. Conjunctions are represented explicitly. Each node in the tree represents a single goal. The children of a node in the tree are the bodies of the clauses whose heads unify with the node. The conjunctions in the body of each clause are AND-ded together as children of this node. The arcs are labelled with respect to the clauses which are the disjunctive descendants of a node and the corresponding most general unifiers. The leaf nodes are either empty or goals.

---

\(^1\)This is called the computation rule [Llo87]
unable to be resolved further. When there are multiple solutions for a particular goal, it is hard to represent the overall picture of the computation, as for each substitution, the conjunctions need to be re-represented.

OR representation of the search tree is widely used in [SS86, Llo87]. The root of the tree is the query \( G \). Each node in the OR tree is a set of resolvents which is obtained by selecting one of the goals in the parent node and a clause whose head unifies with the selected goal and replacing the selected goal with the body of the clause. Each arc originating from a node is labelled by the clause that is used to generate the child node and the mgu resulting from unifying the head of the clause and the selected goal. The leaf nodes are either empty clauses which correspond to successful computations, or they contain resolvents which can no longer be resolved. This representation makes the resolvents in the computation explicit, but conjunctions are not represented. It is a collapsed AND-OR tree [PB87]. Both representations have advantages and disadvantages depending on what we want to illustrate about a computation for visualization. They are covered in [PB87].

A proof tree is a search tree which represents a successful computation of a \( G \) given a \( P \). For an AND-OR representation of the search tree, it is an AND tree\(^2\), where all the leaves are empty or a branch in the OR tree which has the empty goal at the leaf node.

Search strategies and constructing search trees are covered in detail in Section 2.2, Chapters 3 and 4.

### 1.2.2 Software Engineering

This thesis is concerned with meta-programming with an engineering point of view. One of our objectives is to illustrate the impact of software engineering methods in development of tools by meta-programming. We focus on using the stepwise enhancement methodology which was also proposed for logic programming development in [Lak89]. Recently this methodology is further studied in this department for developing Prolog programs and libraries based on the specification of skeletons and techniques [SKW⁺].

\(^2\)provided that there are no set predicates
In [LS90], Lakhotia defines a skeleton program as a program which encodes a flow of control.

“A skeleton program aims at capturing the factor influencing the primary control flow of a program. The factor might be data structures, algorithm, program cliches, language syntax or language semantics.”

By first identifying a skeleton program, we can extend the functionality of this program by using techniques which help us represent a particular aspect about the skeleton program explicitly and help us write more complicated programs based on this skeleton. The important point is first the identification of a skeleton as a basic control structure and enhancing it to accomplish a specific function based on this structure. Starting from a skeleton program, it is easier to write a complicated program by incrementally developing the skeleton and by using a set of techniques which reshape or increase the functionality of this skeleton. This method is called stepwise enhancement. Techniques add either new arguments or goals to the skeleton.

For example, consider the following tree traversal program as a skeleton.

\[
\text{tree(Tree) } \leftarrow \text{ Tree } = \text{ nil.} \\
\text{tree(t(Node, Left, Right)) } \leftarrow \text{ tree(Left), tree(Right).}
\]

A program for counting the nodes of a tree can be written easily based on this skeleton.

\[
\text{tree(Tree, Num) } \leftarrow \text{ Tree } = \text{ nil, Num } = 0. \\
\text{tree(t(Node, Left, Right), Num) } \leftarrow \\
\text{ tree(Left, LeftNum),} \\
\text{ tree(Right, RightNum),} \\
\text{ Num is LeftNum + RightNum + 1.}
\]

Also, a new program can be obtained by changing its base case, for example to count the leaf nodes of a tree as Tree = t(Node, nil, nil), Num = 1.
The technique which adds a new argument as the count of the leaves is categorized as an example of the calculate technique, which by adding new goals and arguments, makes a calculation about the structure which the control is based on. In this example, the structure is a binary tree. The collect technique is used to collect or construct another structure by abstracting from the control flow. For our example, the collect technique will collect the internal nodes of the tree, or leaves of the tree as a list as an example.3

This methodology also brings the issue of using different enhancements of the same skeleton and composing them to obtain the composite functionality of the separate enhancements [SL88]. For developing logic programs, Kirschenbaum and Sterling [KS90] define various skeletons and techniques based on a previously decided specific skeleton.

A shell is described as a specification of a logic/Prolog skeleton where not all parts of the skeleton are necessarily fully specified or developed as a program. A shell is refined by including the descriptions of the unspecified goals in the body or by adding or deleting new goals in the body of a clause. For the above example, an uninstanitated base case will provide a shell as we instantiate it later to either leaves or the empty tree. In addition, sometimes the enhancements of a skeleton provides a different skeleton. This usually occurs when the skeleton's control flow is altered based on an extension which is originally used to enhance the skeleton. Different abstractions of the same control flow also describe new shells. If a shell is specified fully and is runnable, then it is a final program. However, the programs can still be used as skeletons to describe new programs or shells by further refinement.

1.2.3 Artificial Intelligence

In the chapter on meta-linguistic abstractions in their well-known textbook, Abelson and Sussman [AS85] stress the importance of establishing new descriptive languages for programming. They stress the need for tools and tech-

3I believe that this technique should be called as construct technique. This name defines its function more adequately. In this thesis, however, I will refer to this technique as collect technique in order to be consistent with the rest of the publications of the COMPOSERS group [SKW+].
niques both to formulate new languages and to implement these languages by constructing evaluators as follows.

"Thus, given a language that a computer knows how to evaluate, if we can implement in that language an evaluator for the second language, then our computer will also be able to evaluate expressions of the latter language. This method of implementing a language is known as constructing an embedded language."

Abelson and Sussman describe here how one language can describe the design and interpretation about another language. This method is only practical if it is easy to design and implement the language. This is where the role of meta-programming becomes obvious. Meta-programming enables building tools for extending or revealing the representational and semantical aspects of languages at a high level. From an engineering point of view, it is also much simpler to build new languages and tools within a high level language by delegating some of the work, i.e. matching, unification, or backward chaining to the underlying system.

Above, we gave a definition for a shell within the context of software engineering for logic programming. For our purposes, it has a different meaning. An expert system shell is described as the inference engine and the user interface of an expert system without a knowledge base [Bra90]. Our intent in this thesis is to illustrate the techniques for developing shells of the second kind, although we will use stepwise enhancement based on runnable skeletons.

The interpretation or control flow of a language can be defined by a meta-program as a skeleton. This skeleton serves as the basic interpreter (or evaluator) of the language and further specialized by using the enhancement techniques, like the ones discussed above, in order to build special purpose inference engines based on this language which can further alter, reveal or enhance the interpretation and the representation of the language. The changes and enhancements are made at the meta-level. In this respect, meta-programming

4interpreters
has close ties with meta-level inference and meta-level architectures. Two recent studies which discuss meta-level architectures for knowledge based systems in detail are [van89b, Jrv89]. We discuss this topic in Chapter 2.

In developing knowledge based systems, we will build systems that are based on meta-level inference by using meta-programming techniques in Prolog. We use meta-programming in Prolog as a tool for representing and extending its inference which we use as a basis for developing special purpose shells. We identify the meta-programs which serve as the skeletons for describing the control flow or interpretation of a domain language\textsuperscript{5} from which we specify expert systems shells. Skeletons have varying capabilities in representing all the computational aspects of a language. Identification of particular skeletons which can serve as a basis for modelling designated aspects of computation, such as negation, is necessary. We identify different skeletons to describe single or multiple lines of reasoning, negation and also discuss the techniques for specializing the inference based on these skeletons in Chapter 4.

1.3 Results

The results achieved in this thesis are as follows.

- classification of the approaches identified as meta-programming in logic programming.

- identification of the existing techniques for meta-programming in Prolog, and their extension for knowledge representation.

- identification of new techniques and skeletons for dealing with multiple lines of reasoning, and negation-as-failure.

- extension of these skeletons for applications to develop expert system shells in Prolog.

- specification and development of an explanation system and a shell for uncertainty reasoning in Prolog.

\textsuperscript{5}either Prolog, extended Prolog or a special purpose language such as ordinal reasoning
1.4 Outline of the Thesis

This thesis has five basic sections. In the next chapter, we begin with the discussion of the title of this thesis, what meta-programming is all about. Namely we will discuss its definition, its purpose and the main issue that regards meta-programming: the representation of languages and control for meta-programming. This chapter can be considered as our justifications of why we use meta-programming.

We discuss how meta-programming is approached by other researchers in the field, its purpose and its uses in the third chapter. We will look at the role of representation in Prolog and other proposed logic based systems in the literature.

In the fourth chapter, we will look at different skeletons that describe computational behaviour of Prolog programs. We differentiate skeletons on the basis of their expressive power for describing computational behaviour, specifically with respect to their capabilities of representing single versus multiple line of reasoning. Existing and new techniques for enhancing skeletons as shells are also discussed with respect to the functionality of the skeletons we want to achieve.

The fifth, sixth and seventh chapters deal with three aspects of using meta-programming in Prolog, namely building an embedded language, a shell for explanation and a shell for uncertainty reasoning based on Prolog and within Prolog. The shells presented in these sections take the skeletons discussed in the fourth chapter as a basis.

In the last section, we present our conclusions and discuss possible points for future research.
Chapter 2

Meta-Programming

We begin this chapter with definitions of meta-programming. We identify the role of meta-programming for constructing knowledge based systems in Section 2.3. In doing so, we also discuss how meta-level inference and metaprogramming are related, by illustrating the similarities and the differences. Specifics of different approaches in representing logic based languages which support meta-programming will be covered in Chapter 3.

2.1 Defining Meta-Programming

We begin our discussion with describing the main topic of this thesis, by trying to answer the question what is meta-programming? One common definition [AR89, SS86] states

Meta-programming is writing programs that treat other programs as data.

By this definition, familiar programs, such as assemblers, compilers and program transformers are meta-programs.

Another definition of meta-programming can be obtained by following Hayes-Roth’s: Meta-X is a macro definition for ‘X about X’ [HRWL83]. Expanding this macro as in [AL88], yields another definition of meta-programming.

Meta-programming is programming about programming.

Both of the above definitions are descriptive. In our opinion, however, both of these descriptions miss the essence of meta-programming by ignoring both
the purpose of writing meta-programs and the approach taken for developing them. Below, we try to bring to the definitions of another perspective.

In his Turing Award lecture, Floyd [Flo87] describes a paradigm as an approach to writing programs. According to Floyd, a paradigm is a collection of methods and/or techniques to facilitate the construction of certain classes of programs. Examples of paradigms are structured programming, dynamic programming, divide-and-conquer and object-oriented programming. We regard meta-programming as a paradigm. In order to support this claim, we have to answer the question of what meta-programming facilitates.

It is clear that meta-programming is concerned with writing programs about other programs, using them as data by looking at the descriptions above. We talk about two elements in these descriptions, the programs that operate on (other) programs, and the programs that serve as data. As described earlier, metaness refers to aboutness. Van Harmelen discusses the meaning of this word in the context of meta-level reasoning in [van89b]. In our context, the metaness in meta-programming is due to facilitation of programming about programs. Informally, we will use this intuitive notion in giving a description of programs, and calling the ones that operate on other programs as meta programs and the ones that serve as data as object programs. Similarly the languages that used to define such programs will be called meta-language and object language respective of their roles. The computation model of the object lanugage is (partially) determined by representation of its interpretation in the meta-language.

Meta-programming facilitates the means of

- representation of the object language,
- defining the relation between the meta-language and the object language
- an interpretation of the object language.

We next ask the question, What is the purpose of meta-programming? Meta-programming provides building a language which abstracts and interprets other languages. Hence, the purpose of meta-programming is to make
the manipulation of object programs easier. Two different uses of meta-
programming can be identified, based on how and when the interpretation
of the object language is used.

1. Run-time Interpretation of Object Programs. Object programs are con-
trolled by meta-programs at run time. Since the interpretation is speci-
fied and controlled by the meta-level, different aspects of the interpreta-
tion is achievable by meta-programming.

   • Providing different types of interpretation which change the be-
   haviour of the program without changing its representation.

   • Enhancing and extending the model of interpretation of object pro-
   grams by modifying the meta-level specification of the interpreta-
   tion of the object program.

   • Extracting certain aspects of the interpretation, for example a trace,
   of the object level program.

2. Syntactic Manipulation of Object Programs. Based on the implicit or
   explicit interpretation of the object language and meta-programming is
   used to transform, optimize or debug the object level program.

Based on these observations, we provide yet another definition of meta-
programming below.

Meta-programming is building an abstraction of an object language
and providing an interpretation for the abstraction.

This definition does not contradict the previous definitions. However, it
focuses on the purpose behind meta-programming activity. What we are con-
cerned about is defining the abstraction hence the representation of the object
language and the defining of its interpretation.

In this thesis, however, we will use a modified version of this definition of
meta-programming.
Meta-programming is building an abstraction of an object language and developing an interpreter for the abstraction.

Building an abstraction of an object language is based on the extent of available representation of linguistic constructs of the object language. In addition, an explicit interpreter representing the control flow of the object language is not always necessary [van89b, CL90] as shown in Chapter 3. However, our focus in this thesis will be writing interpreters that describe a computation model explicitly.

2.2 Issues in Representation

Van Harmelen states that a computation model of the object level program can be described in three aspects, by representation of the object language, by describing the computational behaviour of the object language and the state of computation of the program in the object language [van89b]. Here we discuss the first two aspects of the computational model. The third one will be addressed in Chapter 4.

For meta-programming, the object language should be available by

- explicit representation of the language constructs of the object language in the meta-language.

- representation of the computation model of the object language by suitable axioms or procedures in the meta-language.

Here we use the term language constructs to indicate the linguistic entities of the object language, such as constants, variables, predicate symbols, function symbols, terms, formulas, sentences, or rules, sets of sentences, objects, sets of objects, etc. This is again a more general view of representation. For sake of simplicity, we will refer to an object language as $O$ and its meta-language as $M$. 
In meta-programming for logic programming, both the object language and
the meta-language are based on Horn clauses. A proper convention for repres-
senting the object level constructs is achieved by a naming relation between
the constructs of $O$ and the constructs of $M$.

We are particularly concerned about

- naming of the object level linguistic entities of $O$ within $M$.
- representation of the provability of $O$ with appropriate axioms in $M$.

The decisions regarding these two issues determine specific approaches to
meta-programming in logic programming.

2.2.1 Naming

For representing object languages by a meta-language, the constructs of an
object language should be identifiable in the meta-language. We are interested
in representing the following linguistic entities of the object language in a meta-
language.

- the alphabet of the object language, including constant symbols, variable
  symbols, function symbols, predicate symbols and delimiters.
- terms
- formulae
- clauses
- procedures and programs.
- proofs

A meta-language has also its own constants, variables, function and pred-
icate symbols, terms, clauses, etc. However, a meta-language should be rich
enough to be able to represent both its own linguistic entities and the ones
in the object language. A meta-level expression in the meta-language is an
expression which contains a reference to syntactic constructs of the object language. In order to be able to talk about the constructs of an object language in a meta-language, the object level expressions are mapped to their names in the meta-language.

Naming first order entities is a well known concept in mathematical logic. Representability of the first order formulae within the same language is achieved by using Gödel numbering for naming. It is utilized in representation of self reference and proving the incompleteness of first order theories and discussed in many mathematical logic texts, such as [Gra77]. Although Gödel numbering is a known convention in theory, it is obviously very impractical for real life systems.

In addressing the truth representation of first order theories within a meta-language, Tarski mentions two forms of naming for representing object-level linguistic entities. The idea is to map the object level expression to a variable free name in $M$.

- Representing an object level construct by a single constant in $M$. This is achieved by quoting the object level expression. For example, we represent the predicate $p(x, f(y))$ by using the constant $'p(x, f(y))'$.

- Representing an object level construct with a variable free term in $M$, where each variable is assigned a number. For example, $p(x, f(y))$ will be represented as $p'(\text{var}(0), f'(\text{var}(1)))$.

Tarski [Tar56] calls these two forms of names quotation-mark names and structural-descriptive names respectively. Structural descriptive names provide a better means of investigating the object level construct explicitly.

Subrahmanian [Sub89] identifies three different cases to illustrate how two logic languages $O$ and $M$ are related, based on the distinction of representations proposed by Belnap and Grover in [BG73]. The choice of languages, $O$ and $M$, and the type of naming convention between them, determine the following categorization for meta-programming.

1. $O$ and $M$ are two different languages. The difference of the languages are due to the capabilities of the naming convention used. This is called
bilingual representation by [van89a].

- The constructs of $O$ are represented by variable free terms in $M$. This is known as the ground representation [HL89]. The ground representation can be achieved by employing either quotation mark names or structural descriptive names.
- Name of names\(^1\) are not allowed. i.e. if $N_1$ names the entity $N_0$, it is not possible to have name $N_2$ that names $N_1$. An interesting consequence of this approach is that self-reference is not allowed in the language.

2. $O$ is contained in $M$.

- amalgamated ground representation. In this representation, $O$ is fully extended to contain the names of entities which can be again names in $O$. The idea is to use a quotation mechanism which is unambiguous again by using ground representation. Representation of Liar's paradox [Sub89] which is an example of self reference is possible with amalgamation since it is possible to quote a quoted name.

- non-ground representation The name of an object level construct is not mapped to variable-free terms in $M$. The object level constructs are represented as is, they are mapped to themselves. Therefore, object and meta-level variables are not distinguished. Prolog falls into this category. Consequently, non-ground representation is also referred to as the mono-lingual representation [van89a].

3. $O$ and $M$ have some common part. Unfortunately, no specific account of this case is in the literature to account for the properties of naming. I believe that it is a restricted case of amalgamation when it is possible to name only some of the names of $O$ and partially axiomatize the provability of $O$.

\(^1\)or double quotation
Adopting a proper naming is important for several reasons. Given an expression for a construct of $O$ in $M$, the most important one is obviously the ability to identify the corresponding object level expressions unambiguously [Esg86]. Resolving ambiguity is an important issue when $O$ and $M$ are amalgamated because, it is necessary to distinguish whether a name is referring to another name or an object level construct. This issue is also important in monolingual systems when multiple levels of interpretation are involved. Given an expression in the language, it is implicit in the expression where a term represents a name or an object level construct. Usually, implicit typing is used to distinguish between the two uses.

Another potential requirement of a naming mechanism is the capability to be able to identify each object level statement (or clause) uniquely. Sometimes we might need to be able to refer to a set of statements of $O^2$. This concept becomes particularly important for representing modules, or when we need to associate different theories/statements with a particular type of inference.

### 2.2.2 Control of Inference

Describing the computational behaviour of a language is by representation of the provability axioms of $O$ by appropriate axioms of $M$ [BK82, SBOS84]. We know that $M$ can not only represent the constructs of $O$ adequately, but should be able to represent the deduction in $O$ by appropriate axioms in $M$. Assume that these axioms in $M$ are called $Pr$ and they are represented by the demo predicate at the meta-level. Given a set of statements in $O$, such as $S$, and a goal $G$, if we can infer $G$ from $S$, as $S \vdash_O G$, then we can infer the same goal at the meta-level by using the provability axioms of the object level theory, as $Pr \vdash_M \text{demo}(G', S')$. Here $G'$ and $S'$ represent the names of $G$ and $S$ respectively.

When the model of the computation of $O$ is represented by an interpreter at the meta-level written in $M$, this type of inference is called meta-level inference. The procedural semantics is described by the inference rules of $O$, which describe the search space, and a search strategy, $\Sigma$, which specifies "how-to"
use these inference rules at the meta-level [van89b]. The search strategy is
described by an interpreter at the meta-level written in $M$ to describe the in-
ference of the language $O$. For the example above, how we write the provability
axioms $Pr$ describes the computational behaviour of $O$.

Representing the control of the computation at the meta-level brings the
issue of appropriateness of the meta-level interpreter for solving the object
level queries at the meta-level. Consider the language $O$, and a query $G$ for a
program in $O$ and the set of answers that are consequences of this program for
this query. The inference is \textit{complete} if all the answers that are consequences
of the program are derivable. The inference is \textit{sound} if the set of computed
answers are the only consequences of the program$^3$. Similar to defining the
soundness and completeness of $O$ and its inference, we can talk about the com-
binatorial soundness and combinatorial completeness of inference of $O$ at the
meta-level. \textit{Combinatorial completeness} indicates whether we can compute all
the results that are derivable from the object level program. \textit{Combinatorial
soundness} indicates whether we can derive only the results derivable from the
object level program at the meta-level. We call this the degree of \textit{faithful-
ness} of the meta-level interpreter's model of the object level computation. By
adjusting the interpretation at the meta-level, we can hinder some of the com-
putation of the object level, thereby affecting the combinatorial completeness
of the interpreter such as avoiding infinite search space, or we can derive more
answers from an object program and affect the combinatorial soundness of the
interpreter. For example, the search strategy $\Sigma$ affects the order of the answers
generated and which one is going to be computed, therefore the combinatorial
completeness of the inference at the meta-level. Van Harmelen further de-
scribes the three components of $\Sigma$, \textit{generative component} $\Sigma_g$, which describes
how to generate the search space, such as the number of nodes that will be
expanded in a search tree, \textit{directional component} $\Sigma_d$ which describes how the
search space should be traversed in the search tree, and \textit{termination compo-
nent} $\Sigma_t$, which describes when the search should be terminated. By changing
these three components, different strategies for the inference can be described

$^3$For a formal treatment of soundness and completeness of SLDNF resolution, see [Lio87]
at the meta-level. Further, specific strategies, such as are obtained by fixing the decisions made for these components. For example a depth first strategy is created by the directional component, and iterative deepening strategy is determined by the directional and termination components.

These aspects have to be utilized by meta-programming systems, especially for writing specific basis shells or to avoid the infinite search space, etc. Changing the interpreter provides what we expect to compute at the object level. Besides, the main purpose for the programming language might be to restrict combinatorial completeness or utilize derivation of additional results and not provide combinatorial soundness [CL89].

Two interesting aspects of the specification of computational behaviour by an interpreter at the meta-level are as follows.

1. One important point to keep in mind is the presence of a base level interpretation. For example, a meta-interpreter written in language $M$ has to rely on the underlying fixed computational model of $M$. Therefore, when the control knowledge of another language such as $O$ is specified by $M$, it should not be forgotten that the inference at the meta-level has to obey a fixed set of rules. Therefore, in order to reveal the characteristics of the computation at the meta-level, a meta-meta-level might be necessary. How many levels of meta-interpretation is necessary is defined by how interesting the meta-level computation is [Smi84]. We can just rely on the base level implementation if no aspect of the computation has to be revealed or altered at some meta-level.

2. The issue of reflection for meta-programming. Maes describes three essential aspects for an introspective system [Mae86], and thus for a meta-level architecture [van89b].

1. the model of the object level computation.

2. a causal connection between the meta-level actions and the object level behaviour.

3. an architecture of inspection which allows the switching between these two levels, namely meta-level and object level computations.
The cases above, 1-2, are specifically interesting in amalgamated systems, because we have the same language both at the base level and at the object level. Van Harmelen claims that the last two aspects are not necessary for meta-level inference systems since the model of the object level computation is fully specified at the meta-level. However, this is not necessarily true for meta-programming, where we might be willing to defer the computation to the object level implementation. Communicating the problems and solutions between the two levels, we can mix the level of interpretations at the same language. Recall that these last two properties of an introspective system are essential for amalgamated systems [BK82]. This is true for Prolog systems which are discussed in Chapter 3 and our focus in this thesis is building systems in Prolog.

Representation of object level failure is another example in favor of an introspective architecture that support the necessity of 2nd and 3rd cases. This aspect might be hidden in the control and can be reflected as the meta-level failure. When the implicit features of the object level computation have to be revealed and have to be monitored by a meta-level interpreter, failure might be needed to be represented explicitly at the meta-level. This requirement might change the basic specification of control that is used at the meta-level and require the state of the object level program O to be explicitly described. Such a formulation is shown in Chapter 3.

Meta-circularity is another interesting property of amalgamated systems. It is the capability of a meta-level interpreter to adequately represent its own computation. Such interpreters are referred to as meta-interpreters in the literature. It is worth mentioning here that this name is quite misleading. As we argued earlier, the metaness is due to the aboutness of one concept with respect to the other. We could have chosen a different representation for an object language, for example an object oriented domain language can be provided based on using a logic based language as the meta-language. This would still be a meta interpreter. Therefore, referring to an interpreter as a meta-interpreter is very misleading, for it indicates the metaness of the interpreter not the circularity of the interpreter. Not all interpreters can adequately represent their
computation models, but might be used at the meta-level. Therefore, meta-circular interpreters will be explicitly referred as meta-interpreters in order to prevent confusion. We will sometime refer to meta-interpreters which are not meta-circular as meta-level interpreters.

3. The third issue is how complete is the specification of the object level computation by the meta-level interpreter. Some of the object level computation might be implicit in the computation at the meta-level. The implicit aspects of the computation might be either due to representation of $O$, or the representation of the computation of $O$. For example, the unification is implicit in monolingual systems, there is no difference between meta-level and object level variables. In addition, the base level implementation which governs the computation at the meta-level which we discussed in 1., has an impact of the completeness of the specification of the object level behaviour implicitly at the meta-level.

2.3 Using Meta-Programming for Developing Knowledge Based Systems

In this section, we will look at the role of knowledge based systems and expert systems shells, discuss the relationships between meta-programming and meta-level reasoning.

For building knowledge based systems, it is important to provide users an appropriate language and suitable techniques to build their applications with. The language used to express the specific knowledge about the domain and the application is called the domain language. Knowledge about a specific application is called the domain knowledge. The domain knowledge is encoded by using the domain language to build knowledge bases.

Let us take a closer look at the user. Merrit [Mer89] describes four types of individuals who interact with a knowledge based system.

- Domain Expert: someone who provides the knowledge in constructing a particular application.
End User: someone who uses the system to get advice or to solve problems.

Knowledge Engineer: someone who encodes the domain knowledge in the domain language.

System Engineer: someone who designs and builds the architecture of the system, designs the domain language constructs, and implements the inference techniques used by the knowledge bases.

We next identify the following roles of expert systems shells.

- providing a language for representing the domain knowledge.

- providing an inference model for the domain, i.e. backward chaining, forward chaining, search strategies for inference.

- illustrating the conceptual or operational aspects of the expert systems decisions or computation, i.e. explanation, debugging, teaching.

- providing changes to the expert system operations, such as altering the standard search strategy or extending the expert systems operations.

An expert system shell does not necessarily fulfill all these roles to its users. Its functionality is usually limited to providing a predefined language representation, inference or search strategy.

Revealing and modifiability of a systems’ actions are highly dependent on the extent of explicitness about the design provided to the user. This is also dependent upon where we define our reference point. The end users’ interaction with a particular system is limited to posing queries, answering the queries of the system or by requesting explanations or other highlights of the operation of the system. This is revealing the systems’ actions by abstraction. In contrast, the requirements of a knowledge engineer are different where a detailed interaction with the systems operations which allows modifiability, without being restricted to inspection of the operation. It is the responsibility
of the knowledge engineer to construct and to modify the system. Therefore, the availability of modification in the shell's design is crucial.

Providing explicitness in a design by separating the domain language and the specification of its inference is a well known method. Its significance is well understood when we look at the role of expert system shells. The inference or control of the domain is independently described based on the representation model of the domain and the domain language. The inference is no longer a "hard wired" language but provided by a meta-level which represents the control or the inference of the domain explicitly. By separating control and domain language, meta-level inference provides a flexible approach to developing systems. Not only can a particular domain be used for multiple tasks, such as simple problem solving or teaching about the domain knowledge itself, but a particular control regime can also be used for reasoning with different knowledge bases represented in the same language.

Further, a collection of inference methods can be specified for different tasks. A particular type of language, or inference is usually suitable for certain classes of problems. For example an object oriented language might be preferable in simulation problems. Besides, it might be desirable to be able to extend domain language, or to be able to provide a collection of languages which could be used or knowledge base can be partitioned for different tasks as modules.

An example of a system which uses logic based knowledge representation and meta-level inference is Socrates [JRv89]. This system provides a logic based domain language for knowledge representation to the user. The domain knowledge can be defined at three different levels.

- Declaration of the logical representation of the language.
- Declaration of the proof theory.
- Declaration of the proof strategy.

The intent is to provide the knowledge engineers great flexibility to design a knowledge based system by using these declarations. The explicit definition
of the logical formulae provides the knowledge engineer flexibility in defining the logical connectives, such as conjunction, disjunction, and how they will be used in retrieval of formulae, such as in matching the object level formulae and the current goals in the inference. The declaration of the proof theory specify the inference rules for the system. The declaration of the proof strategy provides an explicit model of the inference based on the proof theory and the logical constructs of the domain knowledge. The proof strategy is defined as a meta-level interpreter. It performs the object level computation using definable proof strategies, depth first search, best first search, etc. By utilizing explicit representation of knowledge at the levels of defining logical formulae specification of inference rules and defining the control strategy of the inference explicitly, the system is aimed at providing high flexibility where all three aspects are provided by the requirements of the system engineer. For example, forward or backward chaining, or search strategies can be defined depending on the application. Yet a separate fourth level for describing scheduling knowledge is described in [JRv89]. Given a problem, we can define the problem by dividing it into different subproblems. These subproblems are also called tasks. Identification of appropriate tasks and languages has been addressed by Chandrasekaran who identifies the different kind of inference systems and the different possible architectures depending on the task that the system carries for problem solving in [Cha86]. For a knowledge base, different individual subtasks might require particular types of inference. Therefore, a fourth level in knowledge representation might be necessary to associate the task, or subproblem with the appropriate inference model. Sterling refers to the same concept as the planning level in [Ste84] which is proposed to be implemented by using logic based representation. In contrast, van Harmelen et al. propose a procedural implementation of this level and this level is not implemented in [JRv89].

The brief discussion on meta-level inference above is intended to illustrate the relationships between meta-programming and meta-level reasoning. As discussed in Section 2.2, meta-programming uses an explicit representation of the constructs of the object language. The provability or the inference of
the object language is defined by the meta-programs in the meta-language. The representation of the object language is (partially) available for meta-programming. This is very similar to the separation of the domain language representation and its control as discussed with the above example.

Let us consider where meta-programming fits in for building expert systems. A flexible architecture can be only obtained by a high level language and a collection of techniques to aid in the stages of designing and building the architecture of the system. We consider two cases for selecting a language for building the system.

1. For an object language and a meta-language which can represent the constructs of the domain language and its computation model, a subset of the object language can be provided as the domain language. If a language which supports meta-programming is selected for this purpose, then the extensions to the language and the development of other tools for the system, such as explanation, knowledge acquisition, etc, are easier to develop by meta-programming.

2. If a language supports declaration of new operators and allows definition of new constructs directly and permits their reference directly, then a language can be developed using the new operators that are declared and added, and new constructs and their interpretation can be defined within the original language since we have the power to represent the language explicitly. The resulting language is an embedded language within the original language which is then provided as the domain language.

For both cases, the idea is to have an explicit representation of the object language which is used as the domain language. However they differ in the following points. For the first case, a language which supports meta-programming (or a subset of it) is used to explicitly represent the domain language based on the object language. This representation serves as a basis for providing tools, changing the semantics or altering the behaviour of the object language by using the existing representation in the meta-language. In the second case, however, the design and representation of constructs which are not originally
in the language is possible by using the meta-programming capabilities. Since the language is represented on top of the original language and its interpretation is explicitly represented, the embedded language is an object language for the extended original language and can be used as the domain language of the system. A hybrid language can be developed by using both of the aspects of meta-programming, namely using a meta-language to represent two different languages, embedding one in the other and providing the resulting representations as the domain language. An example is given in Chapter 5.

The key idea here is the selection of an appropriate language for the task in hand, the design and implementation of domain languages by providing appropriate models of interpretation or inference. Here, we will use a similar example that [van89b] provides for meta-level architectures. For example, consider a C program which is interpreting 3-Lisp code. From the system engineers point of view, C can be considered as the meta-language for 3-Lisp. On the other hand, 3-Lisp is the object language and also the domain language for the knowledge engineer. 3-Lisp interpreter can be represented in some form which is not accessible to the knowledge engineer. The interpreter written in C is the object level interpreter for 3-Lisp for a 3-Lisp user. However, the system engineer or a knowledge engineer can also use 3-Lisp as the meta-language to represent 3-Lisp and develop a system based on 3-Lisp as the domain language, by using the self-referencing capabilities of this language. The important point here is to identify the respective roles of the languages and the expressive power of the language that we provide to the system and knowledge engineers for building systems. It is the level of abstraction and ease of development in a high level which allows the language to be extended and tools to be developed easily.

The above discussion indicates that using meta-programming should be considered as a tool for building knowledge based systems based on meta-level inference. Recall that in Chapter 1, we gave a description for an expert system shell. A language which enables the explicit representation of the language and its computation provides a tool for representing inference engines and also user interfaces based on this explicit representation. Thus, the contribution of
meta-programming is as a paradigm for knowledge engineering, by facilitating the construction of expert system shells or easy development of knowledge based systems.
Chapter 3

Approaches to Meta-Programming in Logic Programming

In this chapter, we will examine approaches to meta-programming within the logic programming paradigm. This means that we will be concerned about defining a $M$ for representing an $O$ where $O$ and $M$ both have Horn clause representation.

First, we discuss the programming languages available for meta-programming. We will extend the discussion of the previous chapter regarding the classification of languages with respect to the relation between $O$ and $M$, where languages can be amalgamated, separate or overlapping.

Second, we discuss systems, where the intention is not to provide a programming language, but tools to the user to guide the underlying object level interpreter. From a user's perspective, an explicit interpreter is not written in the language to change the underlying computation of the language. Instead, the user can control the computational behaviour of the underlying system by specifying directives to the object level interpreter. The systems in this category focus on controlling the behaviour. They are extensions of the system proposed by Gallaire and Lasserre [GL82] and will be covered in Section 3.4.

3.1 Amalgamated Languages

3.1.1 Bowen/Kowalski

The idea of amalgamation for meta-programming is proposed in the landmark paper of Bowen and Kowalski [BK82]. The intent is to extend $O$ with names to refer to the language constructs of $O$ and provide the provability relation of $O$ within the resulting extended language, which can act as its meta-language
The focus of [BK82] is to define the computational behaviour of the object language by giving provability axioms in the extended language. These axioms describe an interpreter for $O$ in $M$. The provability axioms of $O$ can be represented by the demo predicate in $M$. For a goal, $G$, and a theory $T$, if $T \vdash_O G$, then $Pr \vdash_M \text{demo}(G', T')$. $O$ is extended to represent its names and the amalgamated language can represent its own provability relation.

The following interpreter, demo/2 represents the provability of logic programs as a logic program from [BK82].

\begin{verbatim}
% Demo Interpreter
demo(Prog, Goals) ← empty(Goals).

demo(Prog, Goals) ←
    select(Goal, Goals, RestofGoals),
    member(Clause, Prog),
    rename(Clause, Goals, VariantClause),
    parts(VariantClause, Head, Body),
    unify(Head, Goal, Unifier),
    combine(Body, RestofGoals, GoalSet),
    apply(Unifier, GoalSet, NewGoals),
    demo(Prog, NewGoals).
\end{verbatim}

In demo, a theory or a program is explicitly represented by Prog, the set of goals to be proven by Goals. The behaviour of a logic program is reflected with the demo interpreter. The selection of one of the goals by select and selection of the appropriate clause by member affect the shape of the search tree.

The choice in combining the body of a clause with the existing goals by combine to get the resolvent describes the kind of search strategy. For example,

\begin{verbatim}
combine(Body, RestofGoals, GoalSet) ←
    append(Body, RestofGoals, GoalSet).
\end{verbatim}
describes a depth first search\footnote{The combination axioms are the directional component of search strategy which is discussed in Section 3.4.2.}.

\begin{verbatim}
combine(Body, Restofgoals, GoalSet) ←
    append(RestofGoals, Body, GoalSet).
\end{verbatim}

describes a breath first search.

However, describing different strategies is not the focus in [BK82]. The focus in this paper is the extension of $O$ to include its names as a meta-language for it can refer to its own expressions and refer to its computational model by the provability axioms of demo. The problems of $O$ can be translated and solved by $M$ with the amalgamation.

\[
Pr \vdash_M \text{demo}(G', T')
\]

\[
T \vdash_O G
\]

Also, the results obtained in $M$ can be communicated to $O$ with the following relation.

\[
Pr \vdash_O G
\]

\[
Fr \vdash_M \text{demo}(G', T')
\]

Namely, it is the capability to solve problems by using the object language more efficiently by transferring the results from $M$ to $O$. We can do all the computations at the meta-level but some problems can be solved more efficiently at the object level. These two axioms are known as the \textit{linking rules} between $O$ and $M$, or \textit{reflection principles} [Wey80].

These principles provide the expressive power when $O$ and $M$ are the same, where $O$ has the capability to represent its own entities and computational model adequately. \textit{The meta and object levels can be mixed and used interchangeably within the same language and their results can be used within the same language.} This provides the amalgamated approach with introspective capabilities since self reference is provided.

For example, a pessimist's approach to life can be expressed as follows.

“All men are guilty unless otherwise proven innocent.”

This is represented as

\[
guilty(X) \leftarrow \text{man}(X),
\]

\[
\text{not demo(facts, \text{‘innocent(X)’})},
\]
relevant(facts).

The point worth mentioning here is the representation of provability and normal programs. It is assumed that \texttt{demo(Theory, Goal)} can be proved from provability axioms \textit{if and only if} the \texttt{Goal} can derived from the \texttt{Theory} at the object level. This restriction is necessary to represent normal programs at the meta-level. If the computation of a \texttt{Goal} infinitely fails in \texttt{O}, we then can expect the meta-level to represent non-provability by this restriction. Floundering computations at the object level will cause floundering at the meta-level.

As pointed out by Esghi [Esg86], this paper never illustrates the hard task, choosing an appropriate naming scheme with self referential capability for extending the object language. The language is never developed. For example, \texttt{facts} apparently refers to and thereby “names” a theory, a set of axioms. When a given expression allows mixing the object and meta-levels as given by the example above, it is hard to distinguish whether a particular term is the name of another term (or a name) or simply a term. Esghi suggests an implicit context scoping for this purpose.

MetaProlog extended the ideas about representing theories separately and explicitly [Bow85]. However, the theories are not named by a single term, but given as a list(set) of axioms in a Meta-Prolog program. Meta-Prolog concentrated more on representing different contexts or theories with separate modules for knowledge structuring. The computational model for these modules are the same as Prolog’s [Cic89].

### 3.1.2 Prolog

#### 3.1.2.1 Prolog’s Features

The best known and most widely used logic programming language is undoubtedly Prolog. Prolog has meta-programming capabilities which allows a user both to write an explicit interpreter representing different control and to both represent and access its constructs in a \textit{limited} fashion.

Prolog provides meta-programming by utilizing two different aspects of the language.
1. Prolog provides meta-level built-in predicates to represent and access its linguistic entities, such as clause/2, var/1. The first predicate is used to access the clauses in a program and the second predicate is used to identify whether its argument is a variable. In addition, by using functor/3, arg/3, =../2, a Prolog programmer has the capability to inspect a term or an argument or construct a term. Further, call/1 or implicitly using a variable in a body of a clause (meta-variable facility) provides the ability to pass the deduction to the predicate which is instantiated at the time of the call.

2. Prolog allows the user to define new operators.

Therefore, meta-logical capabilities of Prolog are actually not limited to the ability of referring to Prolog’s constructs only. Combining this feature with the capability to construct and access new terms, predicates, and other constructs allows a user to use Prolog as a basis for extending itself for specific needs, or to design and implement special purpose languages and their interpreters, such as production rules, dcgs, objects, plans, etc. Examples for using these features in AI applications are given in recent texts such as [Mer89] and [Bra90]. In Chapter 5, an example for such an extension will be given.

There is no difference in representation of linguistic entities of a meta-language and an object language in Prolog. More specifically, Prolog has non-ground representation for representing and naming the object-level terms, which are again Prolog terms. That is, there is no explicit variable-free name that is assigned to a linguistic entity in Prolog, every construct represents itself. The built-in predicates provide the manipulation of the linguistic entities. Therefore, meta-programming in Prolog is characterized as mono-lingual [van89b]. The reader might wonder why we chose to discuss Prolog with amalgamated languages. Prolog is not discussed in a separate category by its own, because there are language extensions that provide implicit naming by non-ground representation.

The computation model of Prolog is fixed. It is depth first traversal of the search tree, with a fixed order for selection of the goals and selection of the
solve(true).
solve(A,B) ←
  solve(A),
  solve(B).
solve(A) ←
  clause(A,B),
  solve(B).

Figure 3.1: The Vanilla Interpreter

clauses for the computation.

The computation model of pure Prolog can be represented in Prolog by a meta-circular Prolog interpreter. The interpreter below is called the Vanilla interpreter and it represents the computation model of Prolog programs. Most likely, this name is chosen to represent the basic flavor for defining classes of interpreters².

The clauses of the vanilla interpreter are mutually exclusive. Each clause represent the meta-level axiom to describe the computation of the specific type goal represented as its argument, such as conjunctions, clauses and empty goals. It is assumed that the empty goal is represented by true. The intended declarative reading of this interpreter is as follows. The solution of a conjunctive goal is the solution of its conjuncts, solution of an empty goal is empty and the solution of a clause is the solution of its body.

When we compare vanilla with the Bowen-Kowalski interpreter demo, we see that the search is fixed. For example, the procedural reading suggests the order of selecting the goals implicitly; in order to solve a conjunctive goal, first the first conjunct and then the second conjunct should be evaluated. The predicate clause/2 implicitly replaces the selection of the appropriate clause

²Actually, a simple meta-interpreter can be written within Prolog by using its meta-variable facility for reflection as solve(Goal) ← Goal. The computation is delegated to the base level language, which is Prolog. By using the unification and meta-variable facility, the results obtained at the base level is communicated to the meta-level. However, using this form of reflection and meta-interpreter is not interesting since neither the computation model is represented nor revealed for our purposes.
from the program, renaming and unification of the variables with the current
goal. This aspect is entirely implicit since renaming and unification is handled
by the underlying unification mechanism. In addition, the depth first search
strategy is obtained by solving the body of the selected clause. The underlying
implementation of clause selects the clauses in the order they are defined in
the program.

The interpreter is meta-circular if and only if we could assume a representa-
tion for clause/2 to represent itself, i.e.

\[ \text{clause}(\text{clause}(A,B), \text{true}) \leftarrow \text{clause}(A,B). \]

However, the actual implementations of Prolog do not allow this. There is
yet another problem with the implementation of clause/2. The empty goal
and conjunctions should not be within the meaning of the clause/2 predicate.
Despite the intended semantics, current Prologs create runtime errors instead
of failing for these cases. This forces explicit cuts to be placed in the program\(^3\).
For example, a cut is placed at the first and second clauses of the interpreter
by committing the object level Prolog interpreter to these clauses when we
have an empty or a conjunctive goal respectively. This obviously prevents the
meta-interpreter to be meta-circular.

The vanilla interpreter is usually extended to cover built-in predicates, like
clause/2, or for handling arithmetic, etc. The interpreter is extended with
the following clause with a simple form of reflection.

\[ \text{solve}(A) \leftarrow \text{built-in}(A), A. \]
\[ \text{built-in}(A \text{ is } B). \]

Notice that unification and failure is implicit in the whole interpreter. Uni-
ification is implicit for the whole computation because of the nonground repre-
sentation. The object level unification can be taken for granted to simplify the
selection of clauses and instantiating the variables in a computation. Failure
is implicit in the interpreter because the interpreter is faithful to the object

\(^3\)There are oppositions to this claim as stated in [O’K90]. However, we will stick with
this definition of clause/2 in this thesis.
level computation model. The failure at the object level provides a failure of the meta-level interpreter. These two aspects will be covered in detail below and in Chapter 4.

Vanilla interpreter is faithful to the object level interpretation. It neither computes more solutions than the underlying Prolog implementations, nor restricts the computations. There are many other Prolog meta-interpreters used for revealing, altering or extending the actual model of computation defined at the meta-level, by changing the combinatorial completeness and combinatorial soundness respectively.

We did not discuss handling extra-logical predicates here. In addition, different interpreters are necessary as skeletons for representing a specific control model. These aspects are discussed in Chapter 4.

3.1.2.2 Criticisms Regarding Prolog

Prolog has been criticized widely for its meta-programming capabilities, both for its non ground representation of the object language and object level theory and for its fixed model of computation.

Several aspects of nonground representation are considered inadequate by Esghi[Esg86], Hill and Lloyd [HL89], [NTU84], [Bar89]. These concerns can be classified as theoretic and pragmatic. The solutions can be found in two different categories, developing a different programming language to overcome the theoretical problems, or extending Prolog to provide more flexibility.

A. First, employing the nonground representation is criticized for obscuring the declarative meaning of the program [HL89].

There is no distinction between representing object level constructs and meta-level constructs in Prolog. The variables, the function and predicate symbols, the terms are mapped to themselves. The confusion in trying to understand whether a term is a meta-level term or an object level term is best illuminated by the representation of variables.

A variable in Prolog has a dual role for representation. It can refer to either meta-level or object level constructs depending on the purpose it is used. There is no difference in representation of variables between the object
terms and meta level expressions. This duality is exemplified above by the implicit functionality of the \texttt{clause/2} predicate in \textit{vanilla}. For each clause $P_i \leftarrow B_1, \ldots, B_n$ in a program, we can think that there is a corresponding clause \texttt{clause}(P_i, B_1, \ldots, B_n) in the program. If we would like to think meta-variables ranging over meta-level terms, and object level variables ranging over object level terms, there is no distinction in Prolog. We would expect meta level terms to represent object level constructs such as meta-variables to range over terms, predicates, proofs, theories/programs. On the other hand, object level variables might range over object level terms only. The variables in \texttt{clause}(A, B) range over meta-variables, where the variables in an individual clause range over object level terms. The duality is absorbed by unification which also provides the clause selection based on Prolog's clause order, renaming and unification of the selected goal which is explicitly represented in \texttt{demo/2} interpreter.

The duality occurring as a consequence of the nonground representation is actually first discussed in Nakashima et. al [NTU84], for \texttt{clause/2} and other predicates with side-effects such as \texttt{assert/2}, \texttt{retract/2}, \texttt{read/1}, \texttt{write/1}, and later in [HL89]. Their concern is to allow more flexibility for manipulating terms by extending Prolog. However, Lloyd et. al [HL89] are concerned about the declarative semantics of meta-programs with nonground representation. The problem is overcome by introducing types to the language and by distinguishing between the meta and object level variables. Every object level linguistic construct is mapped inductively to its quoted form at the meta-level. For example, a variable $X$ is mapped to $X'$, a term $f(X, a)$ is mapped to $f'(X', a')$, etc. Conjunction, implication and negation are mapped to distinctive symbols as \&, if and not. By explicit quoting and typing, the vanilla interpreter in Figure 3.1 is reformulated. For example, \texttt{solve(Goal),} the variable \texttt{Goal} has the type meta-level variable as well as the variables $A$ and $B$ at the fourth clause of the interpreter in \texttt{clause}(A, B). On the other hand, a clause such as $p(X) \leftarrow q(Y)$ is considered to be represented as \texttt{clause}(p'(X'), if q'(Y')). By typing, they could illustrate soundness and completeness of the vanilla interpreter [HL89]. However, problems in semantics still exist with
the use of the built-in predicate var/1. For example, consider that there is only one fact in a program, p(a). For the query, var(X), solve(p(X)) the answer will be yes and X will be instantiated to a. However, if we change the order of the goals in the query, it fails because X is no longer a variable.

The clauses or a theory that constitute a program is also implicit in Prolog in contrast to [BK82, Bow85]. Since theories are implicit in Prolog, assert/retract provides a dynamical updates of the theory which create side effects. Namely, the effects of assert/retract can not be "undone" upon back-tracking. The semantics of a program can not be defined when program/theory is implicit. This is true for the side effect family of predicates including read/1, write/1. It is, of course, possible to adopt an interpreter for referring a program explicitly in a way similar to Meta-Prolog where object programs as a list are represented by a meta-level variable and addition and deletion of the clauses are explicit.

B. On a pragmatic account, the meta-programming practice depends on the non-ground representation because of its simplicity for exploiting the underlying Prolog implementation by reflection and also exploiting unification. However, even if we abide by the procedural semantics of Prolog, the non-ground representation creates yet another problem. This is due to "automatic" unification where instantiating meta-variables might accidentally instantiate the object level terms. To overcome this problem, usually a variant of the current goal is used to avoid instantiating the variables in a computation.

The need for a ground representation within Prolog is undeniable. Therefore, there are different proposals to extend Prolog's capabilities or already extensions to Prologs for overcoming some of the existing problems with naming to provide ground goals.

- quotation names
  This representation exists only for clauses in certain Prologs, such as Sicstus and Quintus Prolog. For example Ref in clause(Head, Body, Ref).

- structural descriptive names
numbervars(Term, M, N) predicate converts the term given as an argument to a ground representation, where the variables in Term are numbered ranging from M to N-1.

The numbervars predicate does not provide an adequate representation for naming the terms and clauses since it changes the terms to a ground form by destroying the original one. Even if a variant of a term is used, the representation is not unique. For example, p(X) and p(Y) are different terms but they can be considered equivalent as they are variants. They are represented by the same meta-level term f(var(1)) if the same M and N are chosen as 1 and 2 respectively by using numbervars.

To have both of the representations in the language [SS86], Nakashima et. al. [NTU84] and Barklund [Bar89] suggest having a predicate which freezes a term by a unique ground name as freeze(Term,Name) and a predicate which melts the frozen form to its original representation as melt(Name,Term). Hence given a term, Term, the following axiom is true which indicates a one-to-one mapping between an object level term and its meta-level name.

freeze(Term, Name), melt(Name,AnotherTerm), Term == AnotherTerm

Nakashima et. al also suggest a melt_new/2 predicate which provides a fresh variant of a term given its frozen form. This can easily replace the copy_term predicate in the current Prolog's [Swe88].

The main difference between the proposals of [NTU84] and [Bar89] is whether to provide the same name for all the terms which are variants. Barklund advocates this choice. This is easily implemented with the current technology, by using the fresh variant goal to generate the name for freezing a term, if it is sufficient to map the variants of a term to the same name. However, mapping the variants of a term to the same name, does not preserve the variables uniquely. Therefore, a term after being melted might be a variant of its original representation.

Using frozen representations of terms has also been rediscussed in [SL90] and the same ideas in [NTU84] are proposed in the context of extending constraint logic programming for meta-programming. We have used a restricted naming which is discussed regarding developing an explanation shell later in
Chapter 6 by exploiting the `numbervars` predicate.

In [NTU84], it is also suggested that Prolog should be extended to have
the frozen counterparts of those predicates which not cause side-effects but
also have the duality in instantiating the variables by unification as discussed
above. They are `assert/2`, `retract/2`, `read/1`, `write/1`, and `clause/2`.
Their counterparts are donated by `*` in their name, they not do automatic
renaming of the variables and existing Prolog predicates are defined by them
as follows [NTU84].

\[
\text{assert(Term) } \leftarrow \text{ freeze(Term, Name), assert*(Name)}.
\]
\[
\text{retract(Term) } \leftarrow
\]
\[
\text{rename(Term, NewTerm), retract*(NewTerm),}
\]
\[
\text{melt_new(NewTerm, Term)}.
\]
\[
\text{clause(Head, Body) } \leftarrow
\]
\[
\text{rename(H, NewHead),}
\]
\[
\text{clause*(NewHead, NewBody),}
\]
\[
\text{melt_new((NewHead \leftarrow NewBody), (Head \leftarrow Body)).}
\]
\[
\text{read(Term) } \leftarrow \text{ read*(NewTerm), melt_new(NewTerm, Term)}.
\]
\[
\text{write(Term) } \leftarrow \text{ freeze(Term, Name), write*(Name)}.
\]

Note that this solution is not necessarily declarative.

The problem with this proposal is the definition of `rename/2`. If the predi-
cate `melt_new/2` is assumed to have a ground frozen term as the first argument,
then `rename` must be defined to have two frozen terms as its argument, which
is not specified as such in [NTU84].

It is not accidental that the applications that are suggested in [NTU84]
such as the debugger, global logical variables and meta-level inference are ap-
plications of meta-programming. Although the necessity of having such vari-
ables [NTU84, Bar89, SL90] and their implementation [Bar89] are proposed,
currently they are not part of known Prologs\footnote{Indeed the `freeze/2` predicate has
an entirely different definition [Swe88]}.  

\footnote{Indeed the `freeze/2` predicate has an entirely different definition [Swe88]}
3.2 Bilingual Languages : Gödel

The example in this category is the language, Gödel, which is currently being developed in University of Bristol by the research team directed by J. W. Lloyd. It is constantly changing and developing, but the most recent version of this language is given in the report [BHL90] when this thesis is written.

Gödel is currently being developed as a new language for logic programming to overcome the problems with Prolog, not only its meta-programming capabilities, but to redefine cuts, provide modules and types for programming. We will focus on meta-programming aspects of Gödel in this section after a brief introduction.

Gödel provides programming with modules, where each module contains module and language declarations, and statements within the module. Language declarations define a typed language based on many sorted logic with parametric polymorphism. The declarations declare the symbols of the language, base types, constructors, constants, functions, and predicates. The base types and the constructors which are functions that are defined on base or parametric types define the set of all types which can be used in declarations for the language, namely the types of constants, functions and predicates. Syntactically Gödel employs a representation which is just the opposite of Prolog: the variables are represented by lower case letters, all other names are represented by capitalizing the first letter of the constant. Conjunctions are represented by \&, implications by \texttt{\textless{}-}, disjunctions by \texttt{\textbar{}} and negation by \texttt{\textbar{}}. The universal quantifier is represented by \texttt{\textforall{}} and the existential quantifier is represented by \texttt{\textexists{}}. An example program for defining appending two lists of the type integers, alphabet or list of alphabet or integers, etc. is given in Figure 3.2.

The meta-logical capabilities of Gödel naturally reflect the result of the study of Prolog's meta-programming capabilities in [HL89] where the semantics of meta-programs written in Prolog is investigated. By representing the object level program by using a typed meta-level language, it is possible to illustrate the soundness and completeness of the meta-level interpreters. Lloyd
and his group studied both ground and nonground representations of object languages by a typed meta-language for meta-programming. Consequently, the resulting language Gödel allows meta-programming by using one of the two available representations.

### 3.2.1 Non-Ground Representation

Recall that Prolog has a monolingual representation. The language does not distinguish between object level expressions and meta-level expressions. The expressions are treated "as is". Gödel uses the notation ' to quote the object level expressions at the meta level. The object language 0 is represented as 0' and has the base type ObjectForm. The connectives are represented in their quoted form at the meta-level. The bases are mapped to quoted bases, (i.e. Alphabet to Alphabet') constructors are mapped to quoted constructors (i.e. List to List'), etc., except predicates and connectives which are mapped to functions in quoted form. In addition the variables are mapped
MODULE  UNITAPPEND'.

BASE  ObjectForm, Integer', Alphabet'.
CONSTRUCTOR  List'/1.

CONSTANT  Empty : ObjectForm;
           Nil' : List'(alpha);
           U', M', I', T' : Alphabet'.

FUNCTION  <-': xfx(700) :
           ObjectForm*ObjectForm->ObjectForm;
           Cons': alpha * List'(alpha) -> List'(alpha);
           Append': List'(alpha)*List'(alpha)*List'(alpha).

PREDICATE  Statement : ObjectForm.

Statement(Append'(Nil', ys', ys') <-' Empty).
Statement(Append'(Cons'(x',xs'),ys',Cons'(x',zs')) <-'
           Append'(xs',ys',zs')).

Figure 3.3: Meta-Level Representation of append/3 in Gödel

to quoted variables. The append program is represented at the meta-level as shown in Figure 3.3. The Empty constant is used to represent the empty body of a statement similar to true in Prolog. Gödel has a facility to import and export predicates from modules. At the meta-level a flattened meta-level representation of a module is used by using these declarations to incorporate all the relevant predicates and declarations that constitute this particular module. We selected a simple example for clarity. The details can be found in [BHL90].

Gödel version of the vanilla interpreter given for Prolog in Figure 3.1 is given in Figure 3.4. Call is implemented by a direct call to its argument. The Closed predicates are those which are declared in modules that are declared by CLOSE name [BHL90]. Although we did not discuss here, Gödel provides
MODULE VANILLA.

IMPORT UMITAPPEND'.

PREDICATE Solve, Closed, Call : ObjectForm.

Solve(Empty).
Solve(x &’ y) <- Solve(x) & Solve(y).
Solve(’ x) <- ’ Solve(x).
Solve(x) <- Statement(x <-’ y) & Solve(y).
Solve(x) <- Closed(x) & Call(x).

Figure 3.4: Vanilla Interpreter in Gödel

utilities for Integers, Strings and Lists.

By using the non-ground representation, the vanilla interpreter, similar to Prolog's vanilla, can be the object program to itself. This is a meta-meta evaluation level. The declarations are renewed by quoting the quoted names, such as

BASE ObjectForm, ObjectForm', Integer'', Alphabet''.
FUNCTION Append'': List''(alpha)*List''(alpha)*
                            List''(alpha) -> ObjectForm.

The statements are represented at the meta-meta level as

Statement(Statement'(Append'(Nil'', x'', x'') <-'' Empty'))).

An example of meta-meta evaluation level is given below\(^5\). The convention in meta-meta evaluation levels requires all the arguments of a quoted function to have quotation levels which are equal or more than the quotation level of the function. For example, Append is quoted twice, Solve is quoted once, etc.

\(^5\)The [ ] as syntactic sugar for Cons.
\(-\text{Solve(Solve'}'(x'', y'', [A'', B''])')\).

The non-ground representation utilizes Prolog's unification. Therefore, the semantic problem with Prolog's var still exists here.

### 3.2.2 Ground Representation

There are two major goals in offering a ground representation in Gödel: to be able to overcome the problems of var and to be able to represent an object level program explicitly as a term in the meta-language, instead of a set of declarations and statements. The intent for doing so is to offer retractable counterparts of Prolog's assert/1 and retract/1. By using the ground representation Gödel maps the object level variables to constants at the meta-level.

Gödel incorporates a naming scheme to make all the symbols in an object program unique for the entire program after flattening the modules that the object program is defined by. The intent is to avoid symbol name clashes\(^6\). Gödel offers a module Ground which has all the declarations that are necessary for a meta-program to access and manipulate the object level terms.

All possible linguistic constructs of a parametric many sorted object language are explicitly declared as types. The BASE types of the module Ground are OLan\(-\text{guage}, \text{OConstructor, OFunction, OPredicate, OType, OTerm, OFormula, OTermSubst, OTypeSubst, OModulePart, and OProgram.}\) The predicate declarations in this module are quite lengthy for a concise coverage in this thesis. However, we will illustrate only some of the base level declarations and predicates which can be used for meta-programming.

The predicate declarations include several predicates to access and manipulate the representations of programs as strings and to obtain their counterpart as object level terms. These predicates are useful in making a correspondence between the source level representation of an object program and its internal representation. For example, by using \text{RepTerm/4} predicate as \text{RepTerm}(P, 'UMITAPPEND', 'x', x) we can bind the actual representation of the

\(^6\)This occurs frequently when predicates with the same name are defined in different modules.
variable ‘x’ to the variable x which is in the module UNITAPPEND with respect to program P. Gödel also provides the representations of substitutions for terms and types at the source level to be obtained by the actual representations of the object level program, by RepTermSubst/4, RepTypeSubst.

There are predicates such as And, Or, Not, Implies, Forall, ForSome which provide the representation of connectives and quantifiers for the object language. For example, And/4 is true when in an object language, there are two object level formulae and the fourth argument is the conjunction of this formulae in the object language.

The predicate Variable is similar to Prolog’s var with OType as its type. However, recall that the variables in this representation are mapped to ground representations. Gödel further provides predicates such as Constant, Type, Term, Literal, Statement, NormalBody, DefiniteStatement, etc. to determine the form of an object level formula. Note that Statement is a counterpart of Prolog’s clause and most of such predicates can be easily defined for Prolog. Gödel also provides predicates for identifying types of the object level program and predicates which allow the programmer to inspect a module to determine whether statements, constants, functions, etc. are declared for a given module.

In contrast to Prolog’s implicit treatment of assert/1 and retract/1, Gödel provides InsDel predicates to insert/delete bases, constructors, functions, statements, imported predicates given an object program and a module. It is not specified in the document [BHL90] whether the order of insertions to a program can be specified. Note that the object program is explicit in Gödel, therefore insertions and deletions can be specified without side effects.

Gödel provides predicates similar to Prolog’s meta predicates like functor/3, arg/3, such as FunctionArgs, PredicateArgs, as well as ConstructorArgs which can be used to inspect object level constructor types. Gödel also provides predicates to manipulate unification of terms, and types. Since a ground representation is used, the underlying unification can not be exploited. All the unifications are handled at the meta-level by explicit axioms of unification. Given two object level terms, the predicate UnifyTerms finds the
mgue of these terms. A possible formulation of meta-level unification is given in [BHL90]. In order to be able to define unification at the meta-level, there are other predicates for finding bindings for terms or types in substitutions, applying substitutions to terms, types or formulae, for composing substitutions, for renaming types, terms or formulae. As unification is taken for granted for the monolingual case, it has to be defined fully at the meta-language since ground representation is used.

Gödel has the predicates Success and Fail\(^7\). Success is intended as the Gödel equivalent of the demo predicate given in [BK82]. Fail is intended to be true when a Goal has a finitely failed SLDNF tree for a given flattened program.

### 3.2.3 Discussion

Gödel is intended as a new programming language and it is still being developed. The major problem in Gödel for meta-programming is the inefficiency of the unification. Van Harmelen [van89b] illustrates that the meta-level overhead is larger for bilingual languages in his thesis. Given that meta-programming is already expensive, Lloyd suggests a remedy for unification, reflecting the unification to the implementation level and returning the results of unification, i.e. the substitutions, to the meta-level. Lloyd also suggests that Succeed and Fail should be reflected to the implementation level. This naturally, like designing the object level interpreter which is discussed in Section 3.4.2, prevents a meta-meta evaluation level which we obtain by using the non-ground representation. As Subrahmanian points out [Sub89], this approach prevents the system to reason about its own reasoning, as providing names for names and defining a meta-meta level of computation is not allowed. Further, as we discuss in Chapter 4 for Prolog interpreters, this method does not permit us to use such interpreters as skeletons and their further extensions for obtaining proofs or as basis of shells.

\(^7\)They are not defined in the last report for Gödel. An intended formulation is given in [HL89]
The other problems with Gödel as stated by Lloyd in [BHL90] is the representation of object program and language explicitly. While representing the program has benefits for declarative representation of asserting or deleting object level constructs, it is very impractical to represent large meta-programs in this manner. The problem in representing the language results from checking whether the formulae are well formed in this typed language. Although Prolog has problems, the implementations of non ground representation has benefits by using the language carefully.

3.3 Overlapping Languages: RP

In this section, we will examine Reflective Prolog, or simply referred to as RP, which we believe has an object language and meta-language overlapping.

3.3.1 Reflective Prolog

Reflective Prolog (RP) has been developed at University of Milano by S. Costantini and A. Lanzorone. The related papers describing RP can be found in [CL89, CL90]. The last reference covers this language extensive with the applications of RP which is not published elsewhere.

As its name suggests, Reflective Prolog is a language which is designed to provide automatic reflection based on metaknowledge representation about the object language for meta-programming. It is based on a Horn clause representation.

One of the aims of reflective Prolog is to provide a suitable naming convention for representing metaknowledge which distinguishes and allows the meta-level terms to refer to object level terms within the same language. The intention in RP is to represent the properties of the object level predicates and functions by meta-level names. Based on the meta-level quotation mechanism, resolution is extended to incorporate reflection within the language. The meta-level names and their object level counterparts are used by reflection to switch between object and meta-levels within the same language. This is done to extend the meaning of the object-level program, or in other words to extend
what can be inferred from the object level program. The reflection mechanism is implicit in contrast to [BK82] as it is based on an extended resolution and extended unification instead of an explicit representation of provability. As far as the computational soundness is concerned, the intention in RP is to extend the results that will be otherwise directly obtained from an object level theory. We will give examples later.

RP is a typed language which distinguishes the language constructs by an explicit naming syntactically. It distinguishes meta and object level variables and name constants which are utilized for meta-programming to distinguish the terms. Meta-level variables are three different kinds, which are function, predicate and general meta variables. They are represented by the preceding characters, %, #, and $ respectively. Name constants are divided into categories similarly, to refer to quoted constants in quotation marks ("" ) which refer to object level constants, predicate constants in angled brackets (< >), which refer to the predicate symbols, and the function constants that refer to function symbols by curly brackets ({} ). All the name constants in the language are formed by applying the quotation over the symbols in this set. In addition to object level variables and constants, the terms in RP incorporate name terms that are formed inductively by using name constants, meta-variables and the distinguished predicate and function symbols function, predication, functor, arity, args [CL90] to construct function and relation name terms. Note that this scheme allows building names of names. However, a short notation is allowable by using the brackets, quotation symbols directly, such as <p>("a"). Any term which contains a name term is a meta-level term in RP. A term which can not contain its name in RP, i.e pred(..., <pred>) is forbidden.

An RP program is represented by three different kinds of clauses: object level, meta-level, and meta-evaluation clauses. Meta-level clauses are the ones which contain at least one meta-level term. The meta-evaluation level consist of meta-level clauses which contain at least one distinguished predicate symbol solve. The argument of solve has to be a goal, therefore a relation name term. In addition, the clauses in a RP program are considered in two different
levels. The meta-evaluation clauses comprise the meta-evaluation level. All the remaining clauses comprise the base-level. The name and its referant are forbidden to be used within the same clause in RP which prohibits a meta-meta evaluation level. There is also theory clause, predicate which can be used to represent to clauses and facts in an object program.

The naming relation which relates the terms to their corresponding names is used to dereference the terms to their names or conversely to reference the names to their terms in RP. The naming relation is used in an extended unification that is called the g-unification. Unification is generalized to unify meta-level terms as well as object-level terms by using referentiation and dereferentiation, which therefore switches between meta and object levels. Recall that RP is a typed language. Unification of name terms and object level terms is based on types. For example, predicate meta variables unify with predicate name constants, function meta variables with function name constants. Naturally, general meta variables extend over meta-level terms and unify with meta-level terms. The unification algorithm extends the unification algorithm for SLD resolution and incorporates reflection implicitly by using reference/dereferencing. For example, a predicate meta variable and a predicate symbol unify and dereferencing the predicate symbol to a predicate constant is included in the substitution. Conversely, a predicate meta variable unifies with a term if the reference of the term is a predicate name constant. For example, the following terms $P($X(\texttt{"c"})$)$ and $pr(f(Y))$ unify by where the $gmgu =$ $P/<pr>$, $X/{f}$, $Y/c^8$.

The resolution in RP is called RSLD and it extends SLD resolution. Metaevaluation clauses which contain the distinguished predicate solve and the generalized unification algorithm are utilized to infer answers to queries when normally an answer can not be found by using SLD resolution. We will summarize the RSLD resolution shortly below.

Given a goal $G_i$ in the resolvent $G_1, ... G_i, ... G_m$ and a clause $C \leftarrow A_1, ..., A_n,$

- first tries to resolve them as in SLD resolution.

$^8$There is a typo here in the original paper. "$f$" used instead of \{f\}
• if $G_i = \text{solve}(Goal)$, $Goal$ and $C$ g-unify by $\theta$, then the new resolvent is $(G_1, ..., G_{i-1}, A_1, ...A_n, G_{i+1}, ..., G_m)\theta$.

• if $C = \text{solve}(Goal)$, $Goal$ and $G_i$ g-unify by $\theta$, then the new resolvent is $(G_1, ..., G_{i-1}, A_1, ...A_n, G_{i+1}, ..., G_m)\theta$.

We have not discussed how theory clause are similarly used in the resolution. For details of generalized unification, see [CL90]. RSLD resolution also prevents infinite loops by preventing re-evaluation of variants of the goals at the meta-evaluation level.

The expressive power of RP is observed in describing the properties of relations and functions. The information about functions, relations or terms are represented distinctively by meta-level clauses. Naturally, the properties of terms are meta-level concepts. They are syntactically differentiated by naming in RP programs. Let us give an example to illustrate how RP is used. Our example is an adapted version from [CL90],

\[
\text{solve}(\#P(X,Y)) \leftarrow \text{symmetric}(\#P), \text{solve}(\#P(Y,X)). \\
\text{solve}(\#P(X,Y)) \leftarrow \text{equivalent}(\#P, \#Q), \text{solve}(\#Q(X,Y)).
\]

The relations symmetric and equivalent represent the properties of symmetry and equivalence between certain predicates. This program simply states that we can use the properties of symmetry or equivalence between predicates in order to find a solution to a Goal. These relations will be used if the goal can not be proven at the object level to provide alternative ways of solution.

This is by using the implicit reflection mechanism used in the resolution.

Consider the following relations defined at the base level:

symmetric(<friend>).
symmetric(<equivalent>).
equivalent(<arkadas>, <friend>).

friend(georges, marc).
arkadas(umit, venky).
happy(X) ← friend(X, venky).
In SLD resolution, the query \((\text{happy}(X))\) simply fails. However, RP provides a solution by using the symmetry property of \text{friend} and equivalence of \text{arkadas} and \text{friend} by implicit reflection via generalized unification as described above. RP first tries to find a solution the base level. If base level computation fails, the computation reflects one level up to try to find a solution via the meta-evaluation predicates. At each step the generated resolvent first will be tried at the base level, which is tried automatically by reflecting down in the unification.

In contrast to RP, Prolog has a limited reflective capability. Reflection in Prolog is obtained \textit{explicitly}. Consider the following goal which provides downward reflection to the base level implementation of Prolog. Upward reflection is only achieved when the computation at the base level, namely Prolog's, fails or succeeds respectively. The base level failure result in meta-level failure.

\begin{verbatim}
solve(A) ← A.
\end{verbatim}

Conversely the clause below is an explicit upward reflection.

\begin{verbatim}
A ← solve(A).
\end{verbatim}

The power of RP is by employing an explicit naming mechanism and an expressive language which utilizes automatic reflection effectively.

Can we represent the functionality of RP by meta-interpreters in Prolog? To a limited degree, it is possible to achieve this. We give a Prolog meta-interpreter as the Prolog version of the solution of the program presented above in Figure 3.5. This solution is different than the one given by personal communication in [CL90] and does not suffer the limitations criticized there.

Note that we try to solve the goal at the base level as well, which is given by the first three lines of the vanilla interpreter. However, this has to be handled by explicit reflection since we can not reflect the solution to Prolog's base level for control. If the solution can not be obtained at the base level, then the relations about the predicates are used. In order to seek a solution for the relations, symmetry and equivalence \textit{first} at the base level otherwise at the meta-level, the solution is sought at a meta-level above, i. e. by call of \text{solve(equivalent...}). This interpreter does not have an explicit naming. Besides, the infinite loops are explicitly checked. The interpreter also has to
demo(\text{Goal}) \leftarrow \text{demo(\text{Goal}, \text{[ ]})}.

demo(\text{true}, \_S) \leftarrow !.
demo((\text{A}, \text{B}), \_S) \leftarrow !, \text{demo(\text{A}, \_S)}, \text{demo(\text{B}, \_S)}.
demo(\text{Goal}, \_S) \leftarrow \text{clause(\text{Goal}, \text{Body})},
\quad \text{demo(\text{Body}, \_S)}.

demo(\text{Goal}, \_S) \leftarrow 
\quad \text{not clause(\text{Goal}, \text{Body})},
\quad \text{Goal}=..[\text{Name}, \text{Arg1}, \text{Arg2}],
\quad \text{not invalid(\text{symmetric}, \text{Name})},
\quad \text{not in_stack(\text{Goal}, \_S)},
\quad \text{demo(\text{symmetric(\text{Name}), \_S})},
\quad \text{NewGoal}=..[\text{Name}, \text{Arg2}, \text{Arg1}],
\quad \text{demo(\text{NewGoal}, \text{[Goal|S]})}.

demo(\text{Goal}, \_S) \leftarrow 
\quad \text{not clause(\text{Goal}, \text{Body})},
\quad \text{Goal} =..[\text{Name}\text{|Args}],
\quad \text{not in_stack(\text{Goal}, \_S)},
\quad \text{not invalid(\text{equivalent}, \text{Name})},
\quad \text{demo(\text{equivalent(\text{Name, Name2}), \_S})},
\quad \text{NewGoal}=..[\text{Name2}\text{|Args}],
\quad \text{demo(\text{NewGoal}, \text{[Goal|S]})}.

\text{in_stack(\text{Goal}, \text{[GoalB|S]})} \leftarrow 
\quad \text{variant(\text{Goal, GoalB}), !}.
\text{in_stack(\text{Goal}, \text{[|S]})} \leftarrow \text{in_stack(\text{Goal, \_S})}.

\text{invalid(\_X, \_X)}.

\text{Figure 3.5: Reconstruction of an RP program in Prolog}
check whether the terms that are constructed are valid. Recall that a predicate
can not refer to itself. This is checked by invalid predicate.

We conclude that RP is a useful language which utilizes reflection implicitly. It frees the user by providing a language which is more suitable to express
properties about the contracts of an object language which are then used in
resolution to find solutions which are otherwise not obtainable by SLD resolution
directly. There is a lot of similarity between providing a special purpose
languages, or embedded languages, which are tailored for an application and
RP. RP similarly hides the unnecessary details of reflection from the user.
However, RP also utilizes the reflection by unification at an implementation
level which also increases the efficiency as explicit reflection is not used.

We have classified RP in overlapping languages, because RP provides
names of names of the terms in the language. However, it prevents the name
of a predicate to appear within the scope of the predicate itself. This prevents
RP to have a meta-meta level computation by using the distinguished predi-
cate solve. Namely, RP can not axiomatize the reasoning of the meta-level
with a meta-meta level, although a meta-meta level naming of a predicate is
possible.

3.4 Altering the Object-Level Computation
Model by Object Level Interpreters

In this section, we discuss the systems or proposals which control the object
language interpreter by meta-level specifications without providing a meta-
level interpreter. The intention is not to provide a programming language,
but tools to the user to guide the underlying object level interpreter. From
a user's perspective of such a system, an explicit interpreter is not written in
the language to change the underlying computation of the language. Instead,
the user can control the computational behaviour of the underlying system by
specifying directives to the object level interpreter.

These systems are based on the common observation that the control of
the object level interpreter can be conceptualized as an iterative loop.
1. select a literal

2. select a clause

3. find the unifier of the clause and the literal

4. generate a new goal

5. goto 1.

The idea is to provide hooks to the object level interpreter which enable changing its own behaviour according to the specifications that are provided by the user. These hooks are specified at critical points of this loop which change the control, for example where selection of the goals or clauses occurs (Points 1 and 2 above). By recognizing the process that the object level interpreter undertakes, the necessary hooks can be provided which can manipulate its control and provide the user a language rich enough to specify the control.

Note that at the abstract level, this loop is not very different from the loop defined by the meta-level interpreters given by Bowen and Kowalski [BK82] or a Prolog meta-interpreter 3.1. In meta-programming languages, however, the user has the capability of writing a specific meta-level interpreter by having the access of a given meta-language (whether it is monolingual, bilingual or amalgamated). Here, the idea is to find the suitable hooks and meta-level specification language to guide the object level interpreter. The advantage of using an object level interpreter is obviously avoiding the meta-level overhead. This, of course, depends on how efficiently the meta-level directives are implemented. Meta-level overhead is discussed in detail in [van89b] and will not be covered here.

3.4.1 Gallaire/Lasserre

The first example in this category is the Prolog system described by Gallaire and Lasserre in [GL82]. Their approach is to specify the control of the object level interpreter by meta rules written as Horn clauses without specifying the
entire computation model. These rules have the form

$$\text{Action} \leftarrow \text{Condition}_1, ..., \text{Condition}_n$$

and specify an Action that will be applied in the derivation depending on the set of Conditions. There is a set of meta-predicates which enable the user express the rules to shape the derivation both for clause and goal selection. In the meta-rules a literal or a clause is specified by using Prolog's unification. There is no specific naming mechanism.

The user can specify the order of clauses in the computation two different ways. The order can be specified based on the order of clauses with respect to each other in the program (position directed invocation) or based on the goals that are in their body (content directed invocation). The former is used by using the predicate oporder. The latter utilizes a predefined predicate clause(L,Name,Body), where L gives the subset of clauses that are named by Name and have the goals Body as a list in their body (or alternatively the number of goals in Body). For example, a content directed invocation of clauses can be defined by using the opbefore and clause predicates by the following rule. It states that the shorter clauses (by the number of goals in their body) should be given priority in the computation (which unifies with C).

$$\text{opbefore}(C, \text{Cl}_1, \text{Cl}_2) \leftarrow$$

$$\text{clause}(\text{Cl}_1, \text{Name}_1, \text{Body}_1),$$

$$\text{clause}(\text{Cl}_2, \text{Name}_2, \text{Body}_2),$$

$$\text{less}(	ext{Body}_1, \text{Body}_2).$$

The system has a set of predefined predicates for selecting literals in the computation indirectly. The predicate literal(L,Name,PropertyList) gives a literal L named by Name which satisfies the PropertyList. A set of predefined symbols, such as ancestor, father, depth, solved, can be used to specify the properties of literals. They are given as a pair, i.e. ancestor:A in the PropertyList. These symbols are useful in specifying a particular type of control strategy which can be used to override the default control strategy of
the underlying interpreter, which is the same as Prolog's. There are a set of predicates which affect the process of goal selection, such as before/2 which prefers one literal over another depending on the current resolvent, need/1, which specify the degree of instantiation necessary before a literal can be considered further in the derivation, needby/2, which specifies the resources that a goal in its first argument requires given in the body of the meta-rule and which will otherwise be given by the selection of the goal in the second argument, finish/1, which restricts the attention of the interpreter to the goal in its argument, ready/1 which indicates that a goal must be selected as soon as possible when certain resources are available, etc. Note that predicates like need, ready are available to enable coroutining of the goals. The intent is to be able to write the underlying object level interpreter by a parallel programming language to exploit multi processing capabilities.

Other missing points in this loop are inhibiting backtracking (as introducing a cut), or selective backtracking, providing the termination conditions for the iterative object level loop.

3.4.2 Van Harmelen's Partial Reflection Interpreter

In his thesis [van89b], Van Harmelen presents an interpreter as an extension of Esghi's thesis[Esg86]. The objectives in his extensions are similar to the objectives in [GL82]: to present a hard wired interpreter and a set of hooks which is appropriate to design most of the possible search strategies by exploiting the efficiency of an object level interpreter. Instead of specification of interpretation by a meta-level interpreter, a powerful set of meta-level directives for the object-level interpreter is provided. He calls the hooks to the interpreter which are changeable by meta-level directives as the programmable steps of the object-level inference.

Van Harmelen's approach is based on the observation that the programmable steps of the object level interpreter is based on an explicit representation of the object-level search tree. This abstract data structure search tree is generated by the iterative loop of the computation we have provided above. Its representation is gradually built as the computation progresses by the partial
reflection interpreter. The shape of the tree and its traversal is based on the meta-level directives that are supplied by the user. These are based on the programmable steps in the specification of the interpreter which direct the gradually expanding search tree. Simply, the search tree is built explicitly based on the hooks in the interpreter.

As in [Esg86], Harmelen's approach provides the user to specify clause and goal selection. However, the computation is modified with respect to the current status of the tree. An explicit representation of the tree provides the user the ability to specify the termination of the computation based on this representation and also selective backtracking. From a software engineering point of view, the partial reflection interpreter is a meta-meta specification of the actual interpretation process which is programmed by the user. In contrast [GL82] neither provide an explicit interpreter underlying the computation cycle, nor the search tree that is traversed in the computation.

Since the tree is built gradually and it is an incomplete data structure, we will call it a partial search tree\(^9\). As the tree is fully available in the computation, it is fully inspectable. Van Harmelen represents the search tree as an OR tree and uses a ground representation, where each node in the tree has the triple of the form \((\text{Set of Open Goals, Children of Node, Substitutions for Variables for Computed Goal})\).

The nodes in the tree are divided into types, which are leaves or non leaves. The leaves are non-terminal nodes, nodes which have open goals waiting for expansion, or terminal nodes which are either success or failure nodes. As the tree is constructed dynamically, the type of a node can change. For example, the leaves which are expanded in the interpretation become non-terminal nodes. Depending on the definition of the node type, the programmable steps determine the termination of the interpreter.

We give the interpreter as it is given in [van89b] in Figure 3.6\(^{10}\). The

\(^9\)This is not Van Harmelen's description.

\(^{10}\)Note that this interpreter is not a meta-level interpreter. It is written in Prolog format just for readability as it is intended to be implemented in a medium which will provide efficiency as an object level interpreter.
prove(Goal, Theory, Status) ←
   initialize(Goal, Tree),
   prove_loop(Tree, Theory, Status).

prove_loop(Tree, Theory, succeeded) ←
   * succeeded(Tree).
prove_loop(Tree, Theory, failed) ←
   * failed(Tree).
prove_loop(Tree, Theory, Status) ←
   * select_node(Tree, Theory, Leaf),
   * select_goal(Leaf, Theory, Goal),
   expand_goal(Goal, Leaf, Theory, NewLeaves),
   * filter_leaves(NewLeaves, Theory, FilteredLeaves),
   combine_nodes(FilteredLeaves, Leaf, Goal, FinalLeaves),
   hook_up_nodes(FinalLeaves, Leaf),
   prove_loop(Tree, Theory, Status).

Figure 3.6: Van Harmelen's Partial Reflection Interpreter

programmable steps are represented by * in the interpreter. The main computation loop of the interpreter is represented by the predicate prove_loop where the program (the theory) and the search tree are represented explicitly as its arguments.

Some steps in the interpreter are hard wired, such as the main computation loop(proof_loop) which represents the computation loop that was described before. They are discussed in detail in [van89b]. As seen in Figure 3.6, the predicate expand_goal expands the Leaf based on the selected Goal in all possible ways, by applying all the possible inference rules. This predicate gives NewLeaves where either the goal selected is reduced to subgoals or to a success node. The predicate filter_leaves can be used to eliminate the inference steps are non-promising. The hard wired combine_nodes determines the substitutions that can be applied to the filtered leaves in the tree. As a result of propagating some of the substitutions from the parent node to the new generated leaves, not all leaves are applicable in a resolution step. The predicate hook_up actually reconstructs the tree by attaching the new leaves
to the search tree.

A depth first, non exhaustive search can be programmed to this interpreter as follows. The computation terminates the first time a successful node is found.

\[
\text{succeeded(Node) } \leftarrow \\
\text{successful(Node).}
\]

\[
\text{succeeded(Tree) } \leftarrow \\
\text{nonleaf(Tree),}
\]

\[
\text{children(Tree, Children),}
\]

\[
\text{thereis(Child, Children, succeeded(Child)).}
\]

\[
\text{failed(Tree) } \leftarrow \text{failurenode(Tree).}
\]

\[
\text{failed(Tree) } \leftarrow \\
\text{nonleaf(Tree),}
\]

\[
\text{children(Tree, Children),}
\]

\[
\text{forall(Child, Children, failed(Child)).}
\]

\[
\text{select_node(Node, Theory, Node) } \leftarrow \\
\text{nonterminal(Node).}
\]

\[
\text{select_node(Node, Theory, Node) } \leftarrow \\
\text{children(Node, Children),}
\]

\[
\text{member(Child, Children),}
\]

\[
\text{select_node(Child, Theory, Node).}
\]

As we can see, the computation is based on the current status of the tree by using the predicates describing the nodes of the tree. Recall that a triple is used to denote the nodes in the tree as described above. A successful node will be one which has the empty clause, no pending goals and empty substitutions; a nonleaf node is a node which can not be expanded any further with pending goals, etc. Note also that these definitions are given with the OR tree representation in mind. An AND-OR tree representation would yield a different set of axioms. A breath first aproach is also described in [van89b].
Van Harmelen also correctly observes that this approach requires re-traversal of the entire tree. Naturally, this is a factor which works against having an object level interpreter for the purpose of increasing efficiency. He observes that having a built-in predicate \texttt{leaves} which gives the current leaves of the tree will provide more efficiency in describing the computation. For example, the success and failure in the computation is described below.

\begin{verbatim}
 succeeded(Tree) ←
    leaves(Leaves),
    member(Leaf, Leaves),
    successnode(Leaf).
 failed(Tree) ←
    leaves(Leaves),
    forall(Leaf, Leaves, failurenode(Leaf)).
\end{verbatim}

Goal selection is easily described in terms of this predicate [van89b].

Van Harmelen claims that by using the programmable steps, the termination component of the interpreter can be modelled explicitly as well as the generation and directive component in the search in contrast to [Esg86]. He also claims that other search strategies are easily programmable with this approach.

The reader might wonder why we put emphasis in presenting this object level interpreter and the programmable steps which are written like a logic program. We share a similar interest in terms of moulding the computation with the approach presented here. Our approach will utilize the computational results of the object level at the meta-level. We defer a detailed discussion to Chapter 4.
Chapter 4

IdentifyingSkeletonsandTechniquesfor
Meta-Programming

Our focus in this chapter is representation of control for developing expert
system shells in Prolog. We will discuss the implications of using stepwise
enhancement methodology for developing meta-interpreters which serve as a
basis for shells. We emphasize the identification of the abstraction of control
that a meta-interpreter represents and the techniques for developing the meta-
interpreters.

4.1 Meta-Interpreters as Skeletons

A meta-level interpreter might be written for many purposes, such as for rep-
resenting a particular type of deduction, as a basis of expert system shell,
for extending Prolog or developing a language on top of Prolog. Depending
on the intent that a meta-interpreter is developed, the key issue is represent-
ing the control behaviour of the object language. Depending on whether a
monolingual, bilingual or amalgamated approach is taken, the keys are ex-
plicitness of the behaviour represented by the meta-interpreter which depends
on the abstraction and the support in writing the interpreter provided by the
meta-language.

The idea of using a meta-interpreter as a skeleton is very useful because it
serves as a basis for describing the control flow. Consequently, various
techniques can be identified or developed to enhance the use of this meta-
interpreter to reveal, increase or alter the functionality offered by the under-
lying control flow. Since we are talking about a control flow of a program, it
is the control flow of the object program that is modelled by the meta-level

62
interpreter.

O'Keefe [O'K90] describes an interpreter as "just another program that walks over a data structure". If the definition of skeleton in Chapter 1 is restricted to the context of meta-programming, a meta-program is a skeleton which describes the traversal or process based on a data structure. The data structure for meta-programming purposes is a search tree traversed to represent the control of the object language. A meta-interpreter concretely represents a control structure based on a particular abstraction of this search tree. Nevertheless, the explicitness of this data structure allows or conversely restricts how we represent, control or reveal the computation.

This definition also indicates the importance of selecting meta-interpreters as a basis to build knowledge based systems. Selecting the appropriate meta-interpreter for describing a specific control structure is important as well as both recognizing different skeletons with their capabilities and their limits for enhancement. Therefore, skeletons can be considered in two major categories.

- Skeletons to represent different abstractions of a particular search strategy.

- Skeletons capable of representing different search strategies.

Vanilla interpreter in Figure 3.1 is a well known example of the first category.

The meta-interpreter abstracts the depth first, left to right search of the search tree and it represents the proof strategy. Since the meta-interpreter is written in Prolog\(^1\), it obeys the same rules that it represents.

Skeletons in the second category might be used to represent a particular search strategy and can be used to obtain a skeleton that belongs to the first category. For example, the demo interpreter in Chapter 3 is an interpreter which can be used as a shell\(^2\) to specify a meta-interpreter that belongs to the second category. It is a skeleton which can represent different search strategies to prove a given query.

\(^1\)The base language is Prolog.

\(^2\)from the software engineering point of view as discussed in Chapter 1.
The natural question to ask is whether it is possible or plausible to identify a general skeleton to cover most of the possible purposes that we can envision. As observed in [TS86],

"The ideal logic programming system which we envision will consist of a variety of approximating methods, tools to determine which approximation is exact or sufficient, and a powerful set of optimization techniques rather than a complete and efficient universal interpreter."

We favor and discuss some of the different abstractions provided by meta-interpreters by developing them separately. The advantage of writing separate meta-interpreters for representing different abstractions is the ability to use a collection of meta-programs as skeletons for different partitions of a knowledge base.

Our goal is to be able to develop specializing shells for expert systems such as shells for explanation, uncertainty reasoning, etc. Based on the slogan

**Expert System = Knowledge Base + Inference Engine**

a reformulation of an expert system based on meta-level inference is

**Expert System = Meta-Level + User Shell Inference Interface**

Obviously, a representation of object (domain) language should be available for meta-level inference. Meta-level inference is handled by a meta-level interpreter used as a skeleton which describes a search strategy. Once a certain search strategy is identified and represented as a skeleton explicitly, the skeleton can be used as a basis for developing such customized shells. Customizing shells requires representing the aspect of the object level computation explicitly which needs to be revealed or altered by using stepwise enhancement techniques on the skeleton. In order to achieve our goal, such techniques should also be identified within the context of meta-programming.

Therefore, building specialized expert system shells can be summarized as follows,
• The skeleton with the appropriate abstraction should be obtained for the given task. This task depends on what we need to make explicit about the object level computation so that it can be modelled by the meta-interpreter, i.e. successful computations, failed computations, suspending loops, etc.

• The techniques which are crucial in developing a customized shell should be identified. Once the particular skeleton is chosen, it is easy to build upon this skeleton by obtaining different aspects of this particular abstraction by using the available techniques.

Specializing interpreters can then be defined by the formula

Specializing Interpreters  Meta-Interpreter for Techniques for
for                =  Control + Enhancement
Expert System Shells

In the next section, we review some well known techniques for enhancement.

4.2 Techniques for Enhancing the Skeletons

There are several useful techniques which are used to enhance a meta-interpreter to develop shells. The most common techniques are collect and calculate which are briefly introduced in Chapter 1. Some of the enhanced skeletons which are obtained after applying the techniques are discussed as flavors in [SB89].

4.2.1 Collect

This technique is used to enhance the skeleton to collect/construct some aspect of the object level computation as a meta-level term. For example, the collect technique is used to construct a proof tree of the object level interpretation based on the vanilla interpreter as shown in Figure 4.1.
Collect Technique – solve(Goal, Proof)

solve(true, fact).
solve((GoalA,GoalB), (PA, PB)) ←
solve(GoalA, PA),
solve(GoalB, PB).
solve(Goal, cl(Goal, Proof)) ←
clause(Goal, Body),
solve(Body, Proof).

Figure 4.1: Constructing a Proof by Collect Technique

Collect technique is used to reveal an aspect of the object level interpretation explicitly. Therefore, it is utilized by the shells which have to reveal computation, such as explanation which constructs a proof tree.

There other examples of the collect technique which are critical for explanation as well. For example, an additional History argument is used to collect the line of reasoning as a stack of goals, an additional Tree argument is used to construct a partial proof/search tree. These are discussed in detail in Chapter 6.

4.2.2 Calculate

The calculate technique is used to compute a property of the object level computation. An example of the calculate technique is calculating the maximum depth of the proof tree as illustrated by Figure 4.2. Note that this interpreter does not calculate the depth of the search tree. In addition, proof tree is not explicitly constructed since successful computation of the underlying skeleton represents the provability at the object level.

When the object language is extended as described in Chapter 7 where a certainty factor is assigned to each clause, calculate technique is utilized for uncertainty reasoning.
% Calculate Technique – solve(Goal, MaxDepth)
solve(true, 0).
solve((GoalA,GoalB), Depth) ←
   solve(A, DepthA),
   solve(B, DepthB),
   max(Depth, DepthA, DepthB).
solve(Goal, Depth) ←
   clause(Goal, Body),
   solve(Body, DepthBody),
   Depth is DepthBody + 1.

Figure 4.2: Calculating the Maximum Depth in the Proof Tree

solve(true).
solve((A;B)) ←
   solve(A).
solve((A;B)) ←
   solve(B).
solve((GoalA,GoalB)) ←
   solve(GoalA), solve(GoalB).
solve(Goal) ←
   clause(Goal,Body),
   solve(Body).

Figure 4.3: Adding Disjunction

4.2.3 Extending the Object Language

The enhancements discussed above are easy to develop. However, if the object language is extended by new constructs, such as including disjunctions, new goals have to be added to the skeleton to represent the object level interpretation of the added goals at the meta-level explicitly. It is illustrated in Figure 4.3. This is a stepwise refinement of the interpreter to incorporate the desired extensions to the object language at the meta-level. The techniques, such as collect and calculate have to be built on the extended skeleton.
4.2.4 Altering the Object Level Interpretation

One of the objectives of writing meta-interpreters is to be able to overcome the "problematic aspects of the object level computation at the meta-level". A well known example is to avoid infinite computations of the object level interpretation by detecting loops at the meta-level formulation. In [SS86], a depth bound is used to terminate the computation when a preset depth is reached by the meta-level interpreter. This is represented by an additional argument Depth. In contrast to the role of the Depth argument for the calculate technique which is described above, the Depth is used both to represent the state of the computation at the object level and to change it. Since the the object level interpretation is defined by the meta-level interpreter, this interpreter is a skeleton for describing computations which would be terminated based on the depth bound on the search tree.

A similar technique is to collect the goals that are resolved so far in a list which is represented as an additional argument. This list is referred to check whether a variant of the currently interpreted goal has been previously encountered in the interpretation. This is an indication of infinite computation at the current branch of the search tree.

Depending on how the meta-interpreter is written to resolve this problem, the computation might be terminated at the meta-level or a different branch can be attempted to search for more solutions by suspending the infinite branch. The latter method is described in Section 4.3.4.

The choices affect the combinatorial soundness and completeness of the meta-interpreter. By premature termination, some of the solutions might not be found, thus making the meta-level interpreter combinatorially incomplete.

All the above examples that are presented here are based on monolingual representation.
4.3 Different Abstractions of Prolog Computations

Our focus in this section is to identify and highlight the issues regarding the selection of skeletons as a basis for control and inference. The choice of representation is very important in describing a flow of control based on a search strategy. Abstractions of the same search strategies might be based on several forms of representation. For representing resolution with a meta-interpreter, we might have an OR or an AND-OR representation for the search tree as discussed in Chapter 1. There are drawbacks and advantages of each representation. They are illustrated in [PB87] within the context of selecting the appropriate representation to teach resolution to new learners of Prolog. A similar phenomena occurs in choosing the representation which the meta-interpreter models as a skeleton. For example, in the OR tree representation, the goals in the resolvent which are carried at each step of the computation are explicit. This is reflected in the interpreter. In contrast, the vanilla skeleton based on the AND-OR representation, the resolvents are implicit as they are handled by the base level interpreter.

4.3.1 Single Line Reasoning

As we well know, Prolog's computation model is a depth first left-to-right traversal of the search tree. This computation can be represented by two different interpreters based on AND-OR or OR tree abstractions. For example, vanilla is based on AND-OR representation. The following basic interpreter taken from [O'K90] is also based on AND-OR representation\(^3\). The difference is the representation of goals as a list instead of conjunctions. Only the current goals that are introduced

\[
\text{solve}([ ]). \\
\text{solve}([\text{Goal}|\text{Goals}]) \leftarrow \\
\text{solve\_one}(\text{Goal}),
\]

\(^3\text{Assume every clause is represented as rule}(A, [B1,B2,\ldots,Bn]).\)
solve(Goals).

solve_one(Goal) ←
    rule(Goal, Body),
    solve(Body).

The following interpreter, on the other hand, is based on OR representation representing Prolog's computation. In contrast to the interpreter above, at each stage of the computation, the entire resolvent is explicit. By changing the append predicate, a breadth-first strategy is possible.⁴

% OR
solve([],).
solve([Goal|Goals]) ←
    rule(Goal, Body),
    append(Body, Goals, NewGoals),
    solve(NewGoals).

The difference in abstraction and selecting a basic skeleton do not only depend on the representation. Consider drawing a search tree on paper. Even if the form of the tree is decided, all aspects of the computation are explicit.⁵ That is, the leaves showing successful computations, finite failure and infinite branches, as well as the substitutions are made explicit. A meta-interpreter which traverses the same search tree implicitly abstracts from certain aspects of this tree, while making other aspects explicit. Different abstractions regarding how the interpreter represents and uses the information about the branches of the tree brings us a different skeleton.

The simple skeletons above represent

- Successful computations explicitly.

⁴O'Keefe gives a different version of this interpreter by eliminating the append via using difference lists and term_expansion predicate in order to transform clauses to the rule format used here [O'K90].
⁵As an example, refer to Page 61 in Lloyd [Llo87] where a depth first search tree is represented.
• Failed computations *implicitly*.

• Unification *implicitly*.

### 4.3.2 Multiple Lines of Reasoning

The Prolog meta-interpreters that are discussed above represent a single line of reasoning. They traverse and can handle only one branch of the search tree at a time. However, representing multiple lines of reasoning is also required for many different reasons, such as representing resolution. One of the reasons is for obtaining multiple solutions for a (set of) goal(s). Representing OLD resolution [TS86] requires representing multiple resolvents and solutions for a particular goal, with failed and infinite branches. For expert system shells and AI applications, representing multiple lines of reasoning is also important. For example, in the context of explanation, modelling failure and negation is a desired property. This is because an expert system is expected to provide the reasons for its failure to its users⁶. Reasoning with uncertainty which incorporates multiple evidence also requires representation of multiple lines of reasoning.

Obtaining multiple solutions is usually accomplished by using the set predicates in Prolog, such as `findall` and `setof`. In the context of modeling computations, modelling Prolog computations with failure and negation requires an adequate representation of multiple lines of reasoning. As we discussed earlier, a meta-interpreter serves as a skeleton to describe the control flow explicitly which then can be used as a building block for enhancement. In order to be able to do this, an adequate interpreter has to be written as a skeleton which can represent and then reason about the multiple paths taken in the solution of a goal in the search tree explicitly.

Negation is defined using the negation as failure rule in Prolog. A negated goal succeeds if its non negated form finitely fails in all the subtrees originating from this goal in the search tree. Conversely, a negated goal fails if there

---

⁶This form of explanation is known as the *whynot* explanation in literature and covered in the next chapter extensively.
exists a branch which succeeds among the children of the non negated goal in the search tree. In vanilla, as certain portions of the search tree are not explicitly represented, the extensions of this skeleton will not capture the computation regarding failure branches. For example, one naive extension to vanilla abstraction is to include a negated form of solve to represent negation as one of the mutually exclusive clauses is below.

\[ \text{solve}(\text{not Goal}) \leftarrow \text{not solve}(\text{Goal}). \]

Obviously, any extension on this refined skeleton will not capture the information obtained by typical extensions, such as collect or calculate. Procedurally, if the negated solve indeed succeeds, then this clause will fail. If it succeeds, the clause will not compute extra information via extensions. This is because negation does not instantiate any variables and return any results.

There were two attempts to overcome this problem. The first approach is to write a separate interpreter which will traverse the failure branches in the search tree. The search tree is re-searched when a goal fails by using the second meta-interpreter which explicitly represent the failure branches. This method has been taken by the researchers who attempted to provide a shell for explanation based on meta-interpreters as a basis in [SL86, BH88] and will also be discussed in Chapter 6. The second approach is by an enhancement of the solve interpreter with an additional meta variable, Result, which shows the result of the computation in the current branch of the search tree [LS88, SL88]. The Result is set to yes when the current branch succeeds, otherwise it is set to no. This approach is represented in Figure 4.4.

This new skeleton explicitly represents the failure branches in the search tree. Further, this program illustrates the order followed in solving conjunctions, as if the first conjunct fails, the second one is never attempted and the Result of the conjunction is no. However, this skeleton can not represent the computational behaviour of Prolog. Consider the following program.

---

7In [LS88], both the conjuncts are solved.
solve(true, yes).
solve((A,B), Result) ←
    solve(A, ResultA),
    solve_and(ResultA, B, Result).
solve(A, Result) ←
    built_in(A),
    (A → Result = yes; Result= no).
solve(A, Result) ←
    clause(A, Body),
    solve(Body, Result).

solve(A, no) ← % if the Goal is not covered above
    A ≠ true,
    A ≠ (B, C),
    not clause(A, B).

solve_and(yes, B, ResultB) ← solve(B, ResultB).
solve_and(no, B, no).

Figure 4.4: Result Collecting Interpreter
\texttt{class\_mates(Student1, Student2) ←}
\texttt{takens(Student1, Course),}
\texttt{takens(Student2, Course),}
\texttt{Student1 \neq Student2 .}

\texttt{takens(jane, cs454).}
\texttt{takens(jane, cs471).}
\texttt{takens(bill, cs471).}

This interpreter incorrectly assigns a no result to the queries \texttt{class\_mates(jane, bill)} and \texttt{class\_mates(X, Y)}. This is because the Prolog interpreter continues search until either a successful solution is achieved or all the branches subsequently fail. In this interpreter, the computation stops at the \textit{first} failure branch, instead of searching for a possible solution which might be in another branch of the same search tree. For example, for the first query the interpreter finds the solution for \texttt{takens(jane, Course)} by proving \texttt{takens(jane, cs454)}. The second goal \texttt{takens(bill, cs454)} fails. As the interpreter returns a no result instead of finding another solution for \texttt{takens(jane, Course)} the interpreter incorrectly indicates a failure by a no result. Searching all the possible branches for a solution is handled by automatic backtracking by the Prolog interpreter. This interpreter does not have this feature.

4.3.2.1 Layered Meta-Interpreters

We propose another skeleton to overcome these problems, by which we specify and design our termination condition in \textit{two layers}. Our first observation is to utilize the meta-variable Result similar from the previous interpreter. This variable can be used to monitor the computation in another layer which we will call the \textit{top layer}. Consider a similar interpreter to the previous result generating interpreter above used at the \textit{bottom layer}. The skeleton which we specify is as follows.
solve_top(Result, Goal) ←
    solve_bottom(Result, Goal),
    filter_result(Result, Goal).

solve_top(Result, Goal) ←
    determine_result(Result, Goal).

This skeleton is actually a shell\(^8\) which we can further specify and refine to adjust the computation behaviour by using the Result meta-variable. When the Result variable is used to return a yes or a no result, the following extension can be used. Imitating Prolog’s behaviour, the branches are generated one at a time and the failure of a goal is indicated when all the possible paths are generated and terminate in failure. This is handled by the second clause of solve_top.

filter_result(yes, Goal).
filter_result(no, Goal) ← fail.

determine_result(no, Goal).

We call this class of Prolog interpreters *Layered Interpreters*. The layered interpreter explicitly represents both success and failure branches in the search tree by utilizing Prolog’s backtracking feature. There are two termination conditions for this interpreter with the extensions above. The interpreter terminates at the top level when

- one successful derivation is made at the object level.

- all possible branches of the search tree are generated and each terminate in failure.

The second termination condition is obtained by the second clause of solve_top. Since the meta-interpreter utilizes Prolog’s built-in backtracking mechanism, the variables in a goal are never instantiated if the goal ultimately

\(^8\) in software engineering sense
fails. Note again that the layered interpreter does not fail but return a no result upon the second termination condition.

The full layered interpreter which uses the variable Result as an indicator of the result of the computation is given in Figure 4.5. The clauses of the layered interpreter are mutually exclusive. They handle the possible linguistic entities, such as conjunctions, built-ins, clauses, separately. We have chosen to use solve_top in the first conjunct for uniformity.

In contrast to other monolingual interpreters [SL86, BH88, Nib84], this is the first monolingual interpreter which handles both successes and failures without the necessity of utilizing a separate interpreter. This property enabled handling negations within the same interpreter as shown in Figure 4.5. First the unnegated goal is solved by the interpreter. As we expect, if there is a solution\(^9\), the interpreter returns a no result for the negated goal. Likewise, if the goal ultimately fails, we expect the negated goal to succeed by negation-as-failure. The predicate invert is used to toggle the Result that is obtained by computation of the unnegated goal.

There is one important point here that needs to be clarified. As we discussed in Chapter 3, the monolingual approach is based on the built-in unification of Prolog which converts the object level variables to meta-level variables. In order not to reflect the unification at the object level, a very standard approach is to use a copy of the current goal in the solution. If we wish to communicate the results between meta-level and object level directly, a copy of the goal is not necessary for the layered interpreter as the second clause of solve_top does not instantiate variables upon failure.

By changing the termination specification at the top layer, it is possible to utilize the same shell as an architecture to operate in a different way. For example, the scope of the Result variable can be extended or changed. For the former, we give two examples below, handling cuts and handling infinite computations. For the latter, we can utilize the interpreter to compute belief functions instead of computation results. We discuss this particular extension

\(^9\)Here we refer to the first available solution since the object level computation imitates Prolog's behaviour.
% TOP Layer

solve_top(Result,Goal) ←
    solve_bottom(Result,Goal),
    filter_result(Result, Goal).
solve_top(Result, Goal) ←
    determine_result(Result, Goal).

% BOTTOM Layer

solve_bottom(yes,true)←!.
solve_bottom(Result, not Goal) ←
    solve_top(NotResult, Goal), !,
    invert(NotResult,Result).
solve_bottom(Result,(A,B)) ← !,
    solve_top(RA, A),
    solve_bottom_and(RA,Result,B).
solve_bottom(Result,A) ←
    built_in(A), ( A → Result = yes ; Result = no), !.
solve_bottom(no,A) ←
    not clause(A,Body).
solve_bottom(Result,A) ←
    clause(A,Body),
    solve_top(Result,Body).

solve_bottom_and(yes,ResultB,B) ← solve_top(ResultB,B).
solve_bottom_and(no,no,B).

invert(yes,no).
invert(no,yes).

Figure 4.5: Basic Layered Interpreter
in Chapter 7.

The properties of the layered meta-level interpreter are given below.

1. The layered interpreter uses the *generate-and-test* paradigm to imitate Prolog's behaviour. It generates the branches at the bottom layer, and tests them at the top layer.

2. Unlike the interpreters based on single line of reasoning, the failure of at object level computation is *not* indicated by the failure of the meta-level interpreter\(^{10}\), but a no Result obtained from the meta-interpreter. The variables at the object level are *not* instantiated at the meta-level when the object level computation fails.

3. It is possible to generate alternate solutions by using the layered interpreter, since the interpreter terminates by obtaining the *first* solution but it is backtrackable.

4. It is the first interpreter which can model Prolog's computation behaviour with successes and failures explicitly based on non-ground representation\(^{11}\). Besides, the variables in a goal are *not* instantiated upon failure as we would expect in modelling computations with failure.

The layered interpreter is used as a basis for developing expert system shells. They are covered in Chapter 6 and Chapter 7.

### 4.3.3 Extensions for Multiple Lines of Reasoning

#### 4.3.3.1 Handling Cuts

Prolog's cut has been labeled as the "go-to" statement of Prolog [SS86] due to its problematic unsound semantics\(^{12}\). While giving it a better definition, proper semantics and treatment is underway[BHL90], we will try to model the procedural behaviour of cut by using meta-interpreters.

\(^{10}\)The interpreters are called piggy-back interpreters in [van89b].

\(^{11}\)Note that the clause order for the interpreter is important.

\(^{12}\)For a brief discussion of this topic, see [Kun89].
The procedural semantics of Prolog’s cut given initially in Dec10 Prolog manual is adapted as a standard by other Prologs [Swe88] as follows:

When first encountered as a goal, the cut succeeds immediately. If backtracking should later return to the cut, the effect is to fail the “parent goal”, i.e. the goal which matched the head of the clause containing the cut... In other words, the cut operation COMMITs the system to all choices made since parent goal was invoked...

It is a well known technique to handle cuts at the object level by using a meta variable, which returns a value such as Cut. When a cut is encountered at the object level, then a cut is issued at the meta-level interpreter to control the computation. O’Keefe in [O’K85] illustrated handling cuts by a meta-interpreter by extending the same technique. The goals are partitioned into two groups, as the goals occurring before cut and after cut in the body of a clause respectively. After the body of a goal are solved and if a cut exists in the body, a cut is issued at the meta-level. He calls conditionally issuing a cut at the meta-level as the conditional cut. Again in [O’K90], he uses term_expansion to translate the clauses to a rule structure which will represent this information.

The important question is where and how to reflect the procedural behaviour of commitment in the meta-interpreter. Commitment has a different scope, depending on the Prolog system used or the linguistic construct in question. For example, a negated goal in the body of a clause does not necessarily pose a commitment to the clause it is defined. However, the scope of cuts extends over nested goals, such as disjunctions within conjunctions. Another definitely redundant scope of cut is within if-then-else constructs where a cut within the conditional part of the goal creates a commitment for the clause it is defined.

We use a meta-level variable to handle cuts as well, but our treatment in enhancing the layered meta-interpreter is somewhat different than O’Keefe’s. We take control flow of the language as our basis. At the bottom layer, we handle the cuts only when a failure occurs to return to a cut upon backtracking
following the Dec10 description of cut. Recall that the behaviour of the meta-interpreter was adjusted depending on the value of the Result variable in 4.5. We indicate commitment upon failure by instantiating the Result variable to commit.

solve_bottom(yes, !).

solve_bottom(commit, !).

Note that the clause order is important in this program segment.

The meta-interpreter should reflect the behaviour of the cut appropriately also within the constructs. We include one more clause for solve_and clause to handle commitment within conjunctions which stops the solutions upon encountering commit.

solve_and(commit, commit, B).

The conjunctions are instead defined as follows.

solve_bottom(Result, (A, B)) ← !,

   solve_top(RA, A),
   (RA = yes → solve_top(RB, B),
    ((A = !, (RB = no;RB = commit)) → Result = commit;
     Result = RB);
   (RA = no → Result = no;
    RA = commit → !, Result = commit)).

A single extension to this predicate will not reflect the effect of cut upon failure. Since the scope of cut extends to the parent clause, a cut should be placed in the construct which it is defined at the meta-level. We used a version of solve_and which is partial evaluated and then extended within solve_bottom clause in order to represent the interaction of commitment with the goals that are solved before the cut. Our treatment is implicit compared to [O'K90] and based on commitment upon failure given by the control flow.

A conditional cut is enforced to prevent backtracking of the meta-interpreter itself, such as back to the first conjunct. This can be handled in two different
ways. One of the choices is to impose a conditional cut at the top layer when a
commitment is found. This prevents further backtracking to the bottom layer.
This approach is illustrated by adjusting the first clause of `solve_top` below.

```prolog
solve_top(Result, Goal) ←
copy(Goal, G),
solve_bottom(ResultA, G),
filter(ResultA, Result, Goal, G),
(ResultA = commit → ! ; true).
solve_top(no, Goal).
```

This extension has two exit points to indicate the failure of `Goal`. The
ultimate failure is indicated by the second clause\(^\text{13}\) and the conditional fail-
ure due to commitment is indicated by the conditional cut in the first clause.
However, a copy of the goal for the computation is now necessary because the
variables in the goal might be instantiated at the bottom layer although the
commitment is a type of failure. Since the commitment is handled by the top
layer, the `filter` predicate instantiates the `Result` meta-variable appropri-
ately depending on the type of goal. For example, the `Result` for a predicate
will be instantiated to `no` to indicate failure if the `Result` obtained by solv-
ing body of the goal is commitment. The `filter` predicate also instantiates
the variables by unifying the copy used in the solution and the original goal
when the computation succeeds. Since we used the monolingual approach, it
is natural in order not to instantiate variables upon failure.

The other choice is to handle the effect of commitment within the bottom
layer without altering the top layer. The scope of commitment is transparent
upto the parent clause which has a cut in its body.

```prolog
solve_bottom(Result, A) ←
clause(A, Body),
solve_top(Result, Body),
(Result = commit → !; true).
```

\(^{13}\text{The determine_result predicate is partially evaluated for clarity.}\)
4.3.3.2 Extensions for Additional Constructs

In the layered interpreter, we only presented the code for the conjunctions, built-ins and clauses. However, other constructs, such as disjunctions, if-then-else, etc, can be added by adding another clause to the layered interpreter in 4.5. We assume that the desired constructs are represented by the predicate is_extension_goal/1 as a table.

\[
\text{solve_bottom} \left( \text{Result}, \text{Ext\_Goal} \right) \leftarrow \\
\quad \text{is\_extension\_goal} \left( \text{Ext\_Goal} \right), !, \\
\quad \text{solve\_ext} \left( \text{Result}, \text{Ext\_Goal} \right).
\]

Extensions are also given depending on the values of the Result variable. Prolog’s if-then-else, if-then, disjunctions, and findall are given in Figure 4.6.

Handling cut makes writing the appropriate skeleton more difficult because of the procedural semantics. In this figure, we present how to handle the cuts by extending the scope of the meta-variable Result. This approach is appropriate if we assume that a conditional cut at the top level to handle disjunctions or cuts will be contained only within the body of clauses since a conditional cut will be placed by the parent clause appropriately. This is an interesting decision however. If we assume that a goal which contains cuts can be given as a query, i.e. solve_top((A, !, B), Result) at the meta-level, due to communicating the presence of commitment the commitment might not be placed at the meta-level. This is because of the need to consider the interaction of commitment within nested goals, i.e. \((A,!,B);C\), because the first disjunct has to communicate the commitment to the entire goal and a commitment is necessary at the level of the nested goal. By using a meta-interpreter, the effect of commitments would only be local. This problem can be solved two different ways.

- Checking the computation results at the top level and communicate the occurrence of commitment of a nested goal. This requires maintaining the proof of the computation and communicating the occurrence of commitment by examining the proof.
solve_ext(Result,(A → B; C)) ←
solve_top(ResultA,A),!,
solve_if.then_else(ResultA,Result,B,C).

solve_ext(Result,(A → B)) ←
solve_top(ResultA, A), !,
solve_if.then(ResultA, Result, B).

solve_ext(Result, (A;B)) ←
solve_branch_or(Result,(A;B)).

solve_ext(yes, findall(X,Pred,L)) ←
findall(R-X,solve_top(R,Pred),AList), !,
eliminate(AList,L,Proof).

solve_if.then_else(yes,Result,B,C) ←
solve_top(Result,B).
solve_if.then_else(no,Result,B,C) ←
solve_top(Result,C).
solve_if.then_else(commit,commit,B,C) ← !.

solve_if.then(yes,Result, B) ← !,
solve_top(Result, B).
solve_if.then(no, no, B) ← !.
solve_if.then(commit, commit, B) ← !.

solve_or(Result, (A;B)) ←
solve_top(ResultA, A),
branch_or(ResultA,Result,A,B).

branch_or(yes, yes, B) ← !.
branch_or(no, Result, B) ← !,
solve_top(Result, B).
branch_or(commit,commit,B) ← !.

eliminate([yes-X|Rest],[X|R],[P|Ri]) ←
eliminate(Rest,R,Ri).
eliminate([no-X|Rest],[ ], [P]).

Figure 4.6: Enhancements to the Layered Interpreter by Adding Constructs
• Creating a dummy rule/clause as the top level query which contains the goals with the cut as its body.

4.3.4 OLD Resolution and Prolog

In [War], D.S. Warren presents the development of an interpreter for OLDT resolution [TS86]. We include this example here for two reasons.

• An appropriate skeleton is required for representing multiple lines of reasoning as a basis for representing OLD resolution.

• Stepwise enhancement is also used to extend this interpreter for implementing the computational model for well founded semantics of logic programs[vRS88].

The skeleton has to represent multiple lines of reasoning since the OLDT algorithm computes a set of answers for a given goal. This is in contrast to the layered interpreter because our specification termination there is not always computing all the answers for a query.

The skeleton also has to represent the current branch in the computation and a table of answers obtained for a predicate explicitly [TS86]. The first is required in order to detect infinite computations by checking whether a variant of the current goal has been encountered and to suspend such goals for that particular branch. The answers that are earlier computed for a suspended goal are used to generate other branches to search for possible answers when infinite loops occur.

First modelling multiple lines of reasoning is required. It is given by the doubly recursive interpreter in Figure 4.7. Each goal in the computation cycle is represented by a frame \texttt{Ans <- ListofGoals} where the subgoals required for the computation of goal \texttt{Ans} are given as the second argument. \texttt{Ans} is an atom which is an instance of the head of the clause that is being solved. The list is obtained from the body of the clause that unifies with the predicate. As the \texttt{ListofGoals} are computed, this list dynamically gets shorter. The program is represented by a set of clauses by using a representation similar to [O'K90]
as discussed in Section 4.3.1. They are converted into the frames when the
interpreter identifies the set of all applicable rules for the current goal resolved.

The interpreter represented by the predicate old/2 represents the branch of
the current search tree (Nodes) in the computation explicitly as its argument.
The branch of the search tree is represented as a stack of frames.

The clauses for map-old apply all possible rules that can unify with the
current goal traversing the list of frames. For each such rule, the frame by
unifying this goal and the rule is pushed to the stack and this particular branch
of the tree is traversed by the interpreter. This corresponds to the generative
component of the search which generates all applicable rules one at a time
by utilizing the findall/3 predicate. At the bottom of the tree, indicated
by the second clause of old/2, a terminal node is represented which has no
subgoals. The computed answer of this current goal is returned (apply_subst
to its parent (calling goal) by popping up the stack and unifying the goal
with the leftmost subgoal of the calling goal. Note that the subgoals in frame
are solved from left to right and the environment list of a goal gets smaller as
each subgoal is completely solved. The computation in each branch terminates
when all the subgoal lists are solved as shown in the first clause of old/2.

This interpreter only models the multiple lines of reasoning. For imple-
menting the OLDT algorithm, an extension table is required to represent the
computed answers for a predicate. We also need to represent a list of sus-
spended goals. The solution is to represent all of them in the extension table,
as a triple to represent the predicate, the list of suspended goals (which is the
current branch in the tree) and the list of computed answers for this predicate.

An enhanced version of the previous interpreter\textsuperscript{14} is given in Figure 4.8.
This is an implementation of OLDT resolution described in [TS86].

For a goal, before all the possible rules are applied, the table is consulted
by get_ans. This is to check whether a variant of the goal is already in the
table which indicates an infinite computation. It indicates that this goal is
suspended earlier. If it exists in the table, the current branch is added to the

\textsuperscript{14}We do not give all the code for this interpreter here but explain the major points. All
the details are covered in [War].
\texttt{\textless \texttt{op}(700, xfx, \textless \texttt{)}}.

\texttt{old([\texttt{Ans} \leftarrow [ ]]) \leftarrow write(\texttt{Ans}), \texttt{nl}. \% display answer}

\texttt{old([\texttt{Ans} \leftarrow [ ], \texttt{[Frame|Nodes]}]) \leftarrow}

\hspace{1em} apply\_sub(\texttt{Ans}, \texttt{[Frame|Nodes]}).

\texttt{old(\texttt{Nodes}) \leftarrow}

\hspace{1em} Nodes = \texttt{[\. \leftarrow \texttt{[Goal|\.]}]},

\hspace{1em} findall(\texttt{Goal<\textless Body, rule(\texttt{Goal, Body), Frames),}}

\hspace{1em} map\_old(\texttt{Frames, Nodes}).

\texttt{map\_old([ ], \texttt{).}}

\texttt{map\_old([\texttt{Frame|Frames}], \texttt{Nodes}) \leftarrow}

\hspace{1em} old([\texttt{Frame|Nodes}],

\hspace{1em} map\_old(\texttt{Frames, Node}).

\texttt{apply\_subst(\texttt{Ans}, \texttt{[Frame|Nodes]}) \leftarrow}

\hspace{1em} copy\_term(\texttt{Frame, Goal \leftarrow \texttt{[Ans|Goals]}),

\hspace{1em} old([Goal \leftarrow Goals|Nodes], \texttt{Program}).}

Figure 4.7: Warren's Skeleton for Developing an OLD Interpreter

list of suspended branches and all the computed answers that are obtained so far are returned to the parent of the goal on the current branch traversed. This is handled by \texttt{map\_anss} predicate. These might generate new answers for the goal which in turn extends the table. Otherwise, computation proceeds by applying all the possible rules.

At the bottom of each branch, the table is also consulted (by \texttt{add\_ans}) to check whether the leftmost subgoal of the parent goal has been suspended and the current solution occurs in the table. If the solution is not in the table, it is added. If the goal is suspended first the current solution is returned to the parent goal and all the suspended branches are invoked by \texttt{map\_nodes}.

Warren further extends this particular shell to compute the well founded semantics of logic programs which terminate and does not flounder starting with handling negation. Since the algorithm is based on computation with multiple lines of reasoning, handling negation requires to use an additional table which is empty at the time of the call. By an additional clause, the
\% oldt/3 evaluates the Nodes given a Program and a Table
\% and constructs an output Table with all the solutions

oldt([Ans <- [ ]], Tab,Tab) ← write(Ans),nl.
oldt([Ans <- [ ]|Nodes],Tab0,Tab) ←
    Nodes = [.. < [Goal|.]|.],
    (add_ans(Tab0,Goal,Ans,Susps,Tab1)
    → apply subst(Ans,Nodes,Tab1,Tab2),
        map nodes(Susps,Ans,Tab2,Tab)
    ;Tab = Tab0
).

oldt(Nodes, Tab0, Tab) ←
    Nodes = [.. <- [Goal|.]|.],
    (get_ans(Tab0,Goal,Nodes,AnsS,Tab1)
    → map ans(AnsS,Nodes,Tab1,Tab)
        findall(Goal<-Body,rule(Goal,Body),Frames),
        map oldt(Frames,Nodes,
            [tab(Goal,[ ],[ ])|Tab0],Tab)
    ).

Figure 4.8: Warren's OLDT Interpreter
solutions for a non negated version of the negated goal are computed by using an empty table. If this sub interpretation computes any answers for the non negated goal, then the interpretation does not alter any of the tables to indicate failure. Note that failure of the object level is indicated by the computed answers in the table instead of success or failure of the object level interpreter. Warren successively extends and refines the generated shells to collect answers instead of printing them. The environment tuple is extended to handle a three valued semantics and returns a true or unsup value to indicate unsupported computed answers.

There are couple of interesting points in Warren's approach. From an engineering point of view and within the scope of our discussion regarding skeletons and techniques, this interpreter has the following properties.

- Similar to the partial reflection interpreter in Section 3.4.2 and the layered interpreter presented in this section, object level failure is not modelled by a meta-level failure of the interpreter. It is fail-safe.

- In contrast to those interpreters, object level failure is implicit in the interpreter. For example, the interpreters in Figure 4.7 and Figure 4.8 neither compute any answers nor indicate the status of the computation. Although it is not intended in the model, it is easy to extend the representation of the environment nodes to include a false value to indicate finite failure for each node.

- The interpreter is not written for practical purposes. The data structure to implement the search tree is a stack of environments which provides the alternating top-down/bottom-up approach to represent the computation. By making the branches of the search tree explicit, the computation does not rely on Prolog's computation model for representing the object level interpretation and suspended goals can be handled with the expense of retaining each branch in the search tree.

- The object level unification is handled by the copy predicate which retains the original calling goal without instantiating its variables. Therefore, computed answers of the object level interpretation can only be
obtained from the table, not from the meta-interpreter.

4.4 Discussion

Skelettons represent a particular object-level interpretation by making the necessary properties of the interpretation explicit. In this chapter, we identified the type of skeletons that can be written for a given set of objectives. We also present a novel meta-interpreter to describe Prolog's computation model in layers.

Our study shows that it is possible to write different skeletons to represent the same computation model. The representations we choose are based on the abstract data structure, the search tree. They hinder or enable the extensions that we would like to refine our skeletons further and the capability of our shells that are based on these skeletons.

The traversal of the tree can be handled partially or completely by Prolog's object level interpreter. When certain implicit aspects of Prolog's computation are required explicitly, those aspects need to be represented explicitly in the meta-interpreter as well. For example, the result of the computation, namely success and failure, single and multiple lines of reasoning can be represented explicitly by a layered skeleton at the meta-level without representing the full search tree in contrast to [van89b]. This is because we have the ability to reason about the leaves in the search tree by making the result of the object level computation available by utilizing a meta-variable Result. We can use the object level interpreter to generate and traverse the search tree without explicitly constructing it.

Further, the layered interpreter does not only compute a yes or no result. It can compute other results at the bottom layer which can be monitored at the top layer. Although not discussed in detail in Section 4.3.4, Warren uses three different truth values of the well-founded semantics. Warren comments that a range of values can be used as truth values between 1 and 0 by assigning 1 to true and 0 to false and redefining the value of and, or and negation. He proposes the OLDT interpreter as basis. Our approach to uncertainty
reasoning is already in that spirit and presented in Chapter 7 which is based on the layered skeleton.

It might not however be possible to represent all aspects of the object level interpretation at the meta-level without representing the search tree fully or partially explicitly and using it as a meta-argument in the interpreter. If we wish to represent the suspended goals and their reactivation as in Section 4.3.4, the branches of the search which are suspended need to be represented explicitly. As we discussed earlier, the branches in the search are made explicit similar to using a collect technique for keeping history lists which represent the current branch of the search/proof tree [SS86]\(^{15}\). The collect technique used in the vanilla skeleton is a simple enhancement. It is used for inspection of the computation modelled by the skeleton. On the other hand, the technique used in the OLDT interpreter is for modification of the computation. The computation is based on the data structure that represents the branches of the search tree. For this reason, the layered interpreter can not be easily extended for representing OLDT resolution. The search tree is implicit and based on the object level interpretation.

An explicit construction and representation of a search tree was also discussed in Section 3.4.2 in the context of design of an object level interpreter. Although the object level interpretation can be specified by meta-level directives (or programmable steps), obtaining a set of meta-level primitives which can help in formulating an interpreter efficiently is still crucial. This problem is similar to formulating specific skeletons to describe different interpretations. It is formulating the vocabulary of efficient meta-level directives to the object level interpreter. For example, Van Harmelen suggests [van89b] to extend the available predicates to improve efficiency. The predicate `leaves` gives the current leaves of the tree. This predicate is utilized to avoid re-traversing the entire search space. In order to represent the OLDT resolution by a partial reflection interpreter, we need additional meta-level predicates to represent the suspended nodes in the tree for an efficient implementation. Further, the bottom up propagation of answers can be easily defined if the parent node of

\(^{15}\text{This is discussed within the context of explanation in Chapter 6.}\)
each node was available. This topic requires further research.

Apart from identifying the steps to design a skeleton given a set of objectives, the other question we can ask is whether there exists a systematic way to describe the stepwise refinement procedures for meta-programming, such as showing the mutual exclusivity of the clauses, adding new clauses to the basic skeleton corresponding to extending the object language, or to specify how the clauses of the meta-interpreters actually interact, such as the askable predicates as we will discuss in Chapter 6. For mutually exclusive clauses of the interpreters which represent a particular type of goal, the goal order is not important. The operator precedence is handled by the base level interpreter which is again Prolog. We did not use any specification languages to generate the skeletons. At the moment there is no specification language which will generate the relevant interpreter by transformation to Prolog.
Chapter 5

Extending Prolog by a Simple Embedded Language

This chapter gives an example of designing an embedded language for an expert system application. We illustrate the usefulness of embedded languages by a working example.

5.1 A Simple System for Ordinal Reasoning in Prolog

Consider writing an expert system to evaluate graduate student applications to a department in a university. Such a system must weigh attributes of a student such as GRE scores, grades, recommendation letters, and senior project topic, and classify the student application into categories such as accept, reject, offer aid, consider further.

Our system will evaluate students by trying to write down heuristic rules expressed in these qualitative terms, such as

"The student should be accepted if her gre scores are excellent, the grades are very good (or better) and the recommendation letters are good (or better)."

A system expressing such rules can be directly written in Prolog as given in Figure 5.1.

We assume that the attributes can be described qualitatively, such as gre scores are excellent, recommendation letters are very good, and grades are poor. Further, the qualitative terms are assumed to lie on an ordinal scale, poor is worse than good is worse than very good, etc. A discussion on qualitative reasoning with attributes from ordinal scales is in [BDSP89]. We could
evaluate(Student, accept) ←
   gre(Student, Scores),
   excellent(gre, Scores),
   grades(Student, Grades),
   better_than(Grades, good),
   recommendations(Student, Recom),
   at_least(Recom, very_good).

evaluate(Student, accept) ←
   faculty_decision(Student, Decision),
   better(good, Decision).

Figure 5.1: Rules for Evaluating a Student

represent the ordinal concepts by a table look up predicate, hold, which compares the Student's values by an expected value of a parameter, such as gre scores. This approach is represented in 5.2.

5.2 Representation and Interpretation of an Embedded Language for Ordinal Reasoning

Instead of writing the Prolog program in the previous section, we could represent the ordinal reasoning within Prolog by an embedded language. To build such a system we need a syntax for expressing rules. Borrowing from Prolog syntax, and taking advantage of the ability to define operators, one is lead to rules of the form Conditions $\implies$ Action, where Conditions are a conjunction, denoted & or disjunction, denoted V, of a set of goals of the form (Attribute, RelOp, Value). Five sample rules for evaluating graduate students are given in Figure 5.3.

An interpreter is needed to evaluate whether a certain Student satisfies the requirements written in the rule language. Such an interpreter, is given below by the predicate evaluate(Student, Decision). It tries the rules in turn
evaluate(Student, accept) ←
  hold((gre, =, excellent), Student),
  hold((grades, >=, very_good), Student),
  hold((recommendation, >=, very_good), Student).
evaluate(Student, accept) ←
  hold((faculty_decision, >, good), Student).
evaluate(Student, consider) ←
  hold((gre, =, excellent), Student),
  hold((grades, >=, good), Student),
  hold((senior_project, >=, interesting), Student);
  hold((recommendation, >=, very_good), Student).
evaluate(Student, consider) ←
  hold((gre, >=, very_good), Student),
  hold((grades, =, excellent), Student),
  hold((senior_project, >=, interesting), Student),
  hold((recommendation, =, excellent), Student).
evaluate(Student, refuse) ←
  hold((gre, <, excellent), Student),
  hold((grades, <, good), Student).

Figure 5.2: Re-formulating the Rules for Evaluating a Student

(gre,=,excellent) & (grades,>=,very_good) &
  (recommendations,>=,very_good) → accept.
(faculty_decision,>,good) → accept.
(gre,=,excellent) & (grades,>=,good) &
  ((senior_project,>=,interesting) V
  (recommendation,>=,very_good)) → consider.
(gre,>=,very_good) & (grades,=,excellent) &
  (senior_project,>=,interesting) &
  (recommendation,=,excellent) → consider.
(gre,<,excellent) &
  (grades,<,good) → refuse.

Figure 5.3: A Simple Embedded Language for Ordinal Reasoning
until one is found which is applicable for the given student, essentially using a generate-and-test strategy. By having the representation built on Prolog, two of its features are exploited to evaluate the rules. Firstly, these rules are retrieved by using Prolog’s built-in backward chaining interpreter. Secondly, the instantiation of variables is achieved by using Prolog’s unification.

\[
\text{evaluate(Student, Decision) } \leftarrow \\
\text{Qualifications } \rightarrow \text{ Decision,} \\
\text{holds(Qualifications, Student).}
\]

\[
\text{holds((C1\&C2), Student) } \leftarrow \text{ holds(C1, Student).} \\
\text{holds((C1\&C2), Student) } \leftarrow \text{ holds(C2, Student).} \\
\text{holds((C1\&C2), Student) } \leftarrow \\
\text{ loads(C1, Student), holds(C2, Student).} \\
\text{holds((Concept, RelOp, ExpVal), Student) } \leftarrow \\
\text{ lookup(Concept, Student, Value),} \\
\text{values(Concept, Values),} \\
\text{compare(RelOp, ExpVal, Value, Values).}
\]

The \text{holds} predicate tests that all the \text{Qualifications} on the left-hand side of a rule hold with respect to the \text{Student} under consideration. It depends on \text{compare/4}, which compares the \text{Student’s Value} with the expected value, \text{ExpVal}, of the \text{Concept} based on the operator \text{RelOp} defined in the rule. Code for \text{compare} can be found in [SS86].

5.3 Embedded Languages and Prolog Interpreters

We can now pose two questions. Is the embedded language \text{adequate}? Adequacy is closely related to the purpose of creating the language. The language we presented is obviously a very simple language. It does not have chaining. However, it simplifies expressing and evaluating ordinal concepts for only one
person's qualifications by hiding a simple table look-up procedure for a pre-defined concept uniformly. We cannot use it if we want to compare different people and their qualifications. Besides, it is not obvious how it can be mixed with using Prolog or when and how to delegate the computation to the new interpreter. That brings forth the second question. What is the relation of the embedded language interpreter with the Prolog interpreters that we discussed as skeletons so far?

We can extend the language by using our formulation further. For example, consider that we would like to evaluate the rules based on a particular person, group the rules which are applicable to only a class of people. Further, we require the results of this computation to be used mixed with Prolog. Consider the following simple example. A department secretary has format letters which is sent to the people depending on their performance. However, the ordinal scales and evaluation rules are different for different classes of people, such as grad students, faculty, undergrads.

Let us change the format of our rules to accommodate a type for the rules which groups the rules together:

Type : Conditions⇒Action.

A program segment is given below to evaluate the Person based on appropriate rules defined for its type. The rules and the types for this example are presented in Figure 5.4.

We integrate the embedded language interpreter with the basic meta interpreter by a simple extension in 5.5. In this manner, the embedded language and Prolog can be mixed by enhancement of the appropriate meta-level interpreter to compute solutions for programs which integrate these two languages, like the program segment given above.

There is one issue we have to address here: Providing a shell for the embedded language. The skeleton in Figure 5.5 represents the computation of both Prolog and the embedded language. Thus, it can be used as the basis skeleton for developing the shell. In Chapter 4, we discussed techniques for enhancing the skeletons for developing special purpose shells. However, the standard techniques to extend this skeleton, i.e. using collect for explanation, will not
% example combination
secretarial(Person, Decision) ←
    Person:Conditions ⇒ Decision,
    letter(Person, Conditions, Decision, Letter),
    print(Letter).

type(umit, student).
type(leon, faculty).
type(paul, candidate).
candidate:(gre, <, excellent) &
    (grades, <, good) ⇒ refuse.
student:(math_qualifier, >, good) &
    (area_grades, =, excellent) ⇒ qualify.
faculty:(recommendations, >=, good) &
    (publication_rate, >=, very_good) &
    (publication_quality, =, excellent) ⇒ tenure.

Figure 5.4: Extended Embedded Language

solve(true) ← !.
solve((A, B)) ← !,
solve(A),
solve(B).
solve((Person:Conditions ⇒ Conc)) ← !,
solve(Person, Type),
Type:Conditions ⇒ Conc,
holds(Conditions, Person).
solve(A) ←
    clause(A, B),
solve(B).

Figure 5.5: Meta-Level Interpreter for the Extended Embedded Language
solve(true) ← !.
solve((A,B)) ← !,
    solve(A),
    solve(B).
solve((Person:Conditions⇒Conc)) ← !,
    type(Person,Type),
    Type:Conditions⇒Conc,
    solve(holds(Conditions,Person)).
solve(A) ←
    clause(A,B),
    solve(B).

Figure 5.6: Skeleton for the Embedded Language and Prolog

enhance the holds interpreter that is embedded within the meta-interpreter. We modify the skeleton to be able to represent the interpreter for the embedded language explicitly for enhancement. This is handled by reflecting up to Prolog's interpretation which is described by the meta-interpreter.

5.4 Discussion

We can summarize the experience we gained in this section as follows. An embedded language for an application is built by

- identifying the underlying abstraction necessary for the task.
- defining its representation in Prolog by creating language constructs, such as a new representation scheme for rules.
- building an appropriate interpreter for the language.

The embedded language is more convenient for building a specific function of an expert system, special purpose reasoning. The knowledge engineer has the possibility of communicating more directly with the expert and reducing the gap between expert knowledge and expert system knowledge. The user
does not need to know the full details of Prolog syntax and execution, but can focus on the embedded language. Comparing the system implemented directly in Prolog with building a new rule language illustrates the advantages of using an embedded language. This method provides a general framework for other applications which might use the same paradigm. For example, the generate-and-test paradigm for ordinal reasoning was used for two expert system applications, as described in [BDSP89]. In addition, flexibility and clarity is achieved by separation of the rules and how they are evaluated. Building on the power of Prolog, it is easier to include new rules or change the inference separately by using an embedded language. The embedded language can also be integrated with Prolog computations by extending an appropriate meta-level interpreter which we choose as a skeleton.
Chapter 6

A Shell for Explaining Prolog Programs

Providing explanations is an important function of an expert system's interaction with its users. Starting from MYCIN [BS84], explanation has been one of the studied aspects of an expert system design. In order to illustrate its reasoning, namely what the expert system is doing and why, how it has reached certain conclusions and why it could not, or to justify its actions, an expert system is expected to provide explanations about itself. This facility is regarded as one of the essential factors for establishing the user's acceptability and confidence of the system.

Writing explanation shells is another well known application of meta-programming in Prolog. Relevant meta-programming techniques have been first illustrated in the textbooks [SS86, Ham84] and today they are frequently discussed in other Prolog textbooks such as [Mer89, Bra90]. Our intention here is not to duplicate the efforts, but to illustrate the real issues and problems in developing explanation shells in Prolog. It is complementary to the previous approaches. In doing so, we use our knowledge of the skeletons and techniques discussed in Chapter 4 and illustrate some solutions to the non-trivial problems that occur in using the monolingual approach for developing an explanation shell.

6.1 Explanation Facility for Expert Systems

6.1.1 The Purpose and Content of Explanations

Explanation facilities serve the following basic purposes for expert systems.

- Illustrating the operation of the system.
• Justifying the system by illustrating the underlying domain-specific reasoning.

• Debugging or developing the system by helping the user identify the problems in the operations of the system.

• Teaching the problem solving methods provided by the system.

There are three important points regarding the content of the explanation. First, the representation of the expert systems architecture has the key role for explanation. The systems architecture is based on an abstraction of the domain knowledge used in building the system and its operation. Second, the explanation is given as part of the system’s interaction with its user, thus the identity of the expert system’s user, her knowledge of the systems methods or the problem domain is important, if the explanation is desired to be adaptive for user’s needs. Third, the form of communication or dialogue between the system and the user is an issue.

Traditionally, explanations present a detailed account of the systems operation. The operations of the system can only be revealed if the system is introspective, namely if the system has an account of its own operation. It is simple to observe that what can be explicitly represented about the architecture of the system could be used in providing the explanation. An introspective account of expert systems behaviour can be obtained by using a trace of the systems operation. Namely, the trace of an expert system constitutes the input for the explainer.

The explanation capability is based on the abstractions of the trace of the expert’s operations. The abstraction’s adequacy depends on two different things, whether the expert systems operations are adequately reflected to the trace and whether there are suitable techniques to abstract from the available trace.

The trace of the system also illustrates the adequacy of the knowledge representation chosen for the problem that the expert system is trying to solve. It also shows how explicitly or implicitly the problem solving can be modelled by using the available domain language and methods of inference in
the system. Providing adequate explanations is closely related to epistemology, how explicitly and adequately the domain knowledge can be expressed by using the available domain language. In [Cla83], Clancey points out some of the factors used in building the system are implicit in the representation such as reasoning with taxonomic information buried in MYCIN rules or the implicit control strategy inherent in the system. We have discussed earlier the importance of separating the control for building expert systems. This is not only important for the applicability of control strategies for different domains, but also for the explainability of the control strategies or methods of inference used in building the system. This is similar to the lesson learnt in separating the control knowledge for a backward chaining system explicitly for Guidon and EMYCIN [Cla86].

In order to obtain a trace of the computation, either we extend the domain language for collecting a trace or represent the operations of the expert system at the meta-level and augment this meta-level representation of operation to represent the trace of its own control. The following aspects of knowledge representation need to be explicit for the expert system to give an introspective account of the system.

- An abstraction of the domain language, such as rules, frames, objects, etc.
- An abstraction of how the domain language is used within the control strategy, the matching, unification, inheritance, etc.
- A model of the control or method(s) of inference that are employed in the system.

A justification of the systems design is usually given by illustrating the characteristics of the problem domain. The domain model is not necessarily reflected by the knowledge representation, therefore the justification of the system can not be obtained by the trace when it is not model based. One form of justification is to provide authoritative references to how the knowledge representation is obtained, i.e. by augmenting the rules with references to medical texts as discussed in [Cla86].
In order to display the underlying model, the domain specific model and reasoning is represented explicitly with the design of languages and reasoning specific for the problem in hand. This approach is taken in the XPLAIN system to integrate an explicit representation of the domain model and specific reasoning based on this model [Swa83]. Again the intention is to represent the domain specific reasoning at the language and control levels to provide explanations.

A similar approach is taken in [Cha86], where Chandrasekaran et. al consider the explanation problem at the task level. The claim is to provide the explanation based on identifying the generic tasks that the expert system is designed for, and the inference or control inherent in these tasks. Hierarchical classification, design by plan selection and refinement, state abstraction and knowledge directed information passing are four of the identified tasks. By having a fourth layer of representation, the strategies employed in problem solving can be explicitly represented at the language level and be explained at a better abstraction.

In [JRv89], this is identified as the scheduler, which determines the appropriate inference for the task. The idea is first to identify the necessary abstraction of the model(s) that the expert system (or its components) is designed for, provide an appropriate language for them, which consequently makes the operation of the system easier to explain and close to its purpose. In this manner, the two approaches for explanation, providing a trace and justification are combined. The language presented in Chapter 5 can be considered as an example for this approach where a specific language is given for ordinal reasoning.

The traditional trace based approach is lately criticized because of providing an operational view of explanation, being illustrative only to the knowledge engineer whom we would expect to require a detailed account of the system [WS89] or just simply serving as a debugging aid.
6.1.2 Types of Explanation

The type of explanations that are given to the user can be identified in the following categories.

- *how* explanations illustrate how the system has reached a particular conclusion or solution.

- *whynot* explanations illustrate why the system could not find solutions.

- *why* explanations illustrate why the expert system is currently pursuing a particular line of reasoning.

- *when* explanations provide a temporal account of the systems actions.

- *what* explanations illustrate detailed information about the assertions to the system, such as the possible values that could be assigned to a variable, its type or its value.

- *by whom* explanations illustrate how certain facts or conclusions are derived in the system, whether they are supplied by the user, inferred by the system or given in the knowledge base.

- *what if* explanations illustrate a simulation of inferences that will take place when certain facts, or inferred results are changed by displaying the changes.

In [WS89], Slagle and Wick present various queries and discuss a similar categorization.

Another issue is *when* explanations are provided. An explanation is provided both during the computation of the expert system or after the expert system has completed its operation, finding solutions, etc. The *how* and *whynot* explanations are given after the system has completed its operation. A *why* explanation is given during the systems operation, usually whenever the user needs affirmation about the systems actions. Except the *what-if* type of explanations, the remaining types can be given either after or during the computations. All these types except *what-if* explanation focus on revealing the
operations or decisions of the system. Their purpose is introspection. On the other hand, the purpose of the what-if type of explanation is simulation. It allows the user to change the actions of the system that has completed so far.

6.2 Developing Explanation Shells for Prolog Based Expert Systems

In this section, our goal is to illustrate different techniques which we use for enhancing an appropriate meta-interpreter that will allow us to provide an explanation of a certain type. In doing so, we will restate the issues in writing an explanation system in Prolog which depends on selecting the appropriate skeleton. Let us illustrate the concepts by discussing them along with the historical development of explanation facilities in Prolog for Prolog.

The explanation facilities for Prolog based expert systems are also based on the operational view. They too are based on a trace of the computation. For our purposes, this is based on the search or proof trees. They also develop parallelly to the development of EMYCIN from MYCIN where the computational model is described separately.

The first account of extending Prolog to provide explanations is in [CM82]. There, each clause in the program is augmented by an additional argument. These additional arguments are used to collect the proof tree in solving the goal. The extensions were at the object level program. The separation and representation of domain language and the control was achieved by using a meta-interpreter. A meta-interpreter is used as the skeleton which is enhanced by appropriate techniques which are necessary for an expert system shell. Using a meta-interpreter as a basis for representing the computation was used in The APES (Augmented Prolog for Expert Systems) expert system shell [Ham84]. In APES, how, why and whynot explanations were illustrated. Later [SL86, SS86] discuss these ideas used in providing these type of explanations based on the meta-interpreter technology in detail. We discuss and extend them with respect to skeletons and techniques and present their development in this section.
6.2.1 Techniques for How Explanations

With respect to skeletons and techniques, we have illustrated the utility of the collect technique to construct the proof tree of the deduction in Chapter 4. A proof of deduction collected as a proof tree is used as input for the explainer to provide a how explanation at the end of the computation. Today, recent books on using Prolog for applications such as [SS86, Mer89, Bra90, O'K90] show examples of using this method. Usually these systems give a simple translation of rules used in the computation to English and give a trace of the computation by displaying each rule used in the computation. As the rules are solved by using backward chaining, the rules that are used in solving the goals in the body of the current clause are displayed if the user requests them after examining the current clause. The references [SL86], [SS86] give a trace of the rules based on this method. However advanced tree browsing capabilities are not explored. The explanation advances as the user browses to the deeper levels of the tree. The methods for displaying previous computations is not given. Further, the techniques for explaining full Prolog is not discussed.

6.2.2 Techniques for Why Explanations

A why explanation is given when the user requires a detailed account of the systems current reasoning. Usually, a why explanation is requested when the user is asked to communicate additional information by the system when the system can not infer results from the knowledge base and needs assistance. As a reply to the why explanation, the user is supplied the current line of reasoning during the computation. Previously, however, the skeletons we have discussed did not allow user interference. This is achieved by a well known enhancement which is called the query-the-user facility [Ser83]. The idea is to add another clause to the skeleton in order to provide user inquiry when the interpreter fails to solve a goal. This facility is useful as part of the meta-interpreter because information about certain predicates can be obtained from the user when there is no information about them in the knowledge base. It is used in [Nib84, SL86, SS86]. Here we present our version of this facility.
We assume that a meta-level predicate, `askable` is added to the program, whose argument is a predicate name that can be askable to the user. The predicates which are askable are identified by utilizing the axioms defining this meta-level predicate.

```prolog
solve(Goal) ←
    askable(Goal),
    query_the_user(Goal).
```

With the addition of this predicate, we naturally diverge from the closed-world assumption. Additional information can be asserted to the knowledge base when information about such predicates are not found and obtained from the user.

There are two points to consider in writing the axioms for `query_the_user`. The first one is obtaining and the representation of the user's assertions about the predicate. The other one is the interaction of this new `solve` predicate with the others. These points are closely coupled. The clauses of the meta-interpreter were `mutually exclusive` before. However, by this addition, we have to either distinguish the program clauses and the user answers by representation. A predicate can be both in the theory/program and be askable. These points are not discussed in previous research. Here, we extend the existing proposals for `query-the-user-facility`.

A simple way of representing the positive and negative assertions about a predicate is by using two different predicates as `fact` and `untrue` respectively. The predicates `known_to_be_true` and `known_to_be_false` are used to test whether there are existing assertions about a given predicate. A simple extension is as follows.

```prolog
query_user(A) ←
    known_to_be_true(A), !.
query_user(A) ←
    known_to_be_false(A), !, fail.
query_user(A) ←
    prompt_user([?'Is ', A, ' true ?'], Answer),
```

reply_user(Answer, A).

reply_user(no, A) ←
    assert('untrue'(A)), !, fail.
reply_user(yes, A) ←
    assert('fact'(A)), !.
reply_user(X, A) ←
    not valid_answer(X), writeln([' >Invalid answer!'])),
    query_user(A).

known_to_be_true(A) ← 'fact'(A).
known_to_be_false(A) ← 'untrue'(A).

In this extension, we assume the user will supply a yes or no answer. However, this is not a reasonable assumption. Lalee extended the query-the
user facility to allow greater flexibility for the user to specify certain values of the arguments of a predicate in his thesis[Lal88]. In addition, this facility might seem somewhat similar to trapping unknown goals and activating the debugger in Sicstus and Quintus Prologs. This way the user is given a chance to add additional clauses or consult a file by suspending the execution at that point. Here, however the shell can be tailored for all the unknown or for some specific predicates.

If the facts are assumed to be in mutually exclusive classes, as indicated by predicates like color or size, then the definition of the predicate known_to_be_false can be extended to prevent the interpreter from asking redundant questions. We can use meta-level predicates to describe the properties of facts. A simple extension is given below by using a predicate called function.

known_to_be_false(A) ←
    not 'fact'(Predicate),
    function(Predicate,GenForm),
    'fact'(GenForm).
Here the function creates a new predicate with the same name and the same arguments except one argument. The predicates which are functions can be defined by a tuple, is_function, where the name of the predicate and the number of the argument that it acts as a function are the arguments indicated by Num. For example, is_function(color,2) will be used to indicate that color is a function of the second argument in facts like color(table,blue).

```prolog
function(Predicate, GenFrom) ←
    functor(Predicate, TermName, Num),
    is_function(TermName, IthArgument),
    functor(NewPredicate, Term, Num),
    replace(1, Num, Predicate, IthArgument, GenForm, NewIthArgument).
```

```prolog
replace(K, L, Predicate, I, NewPredicate, NewIthArgument) ←
    K > L, !.
replace(K, L, Predicate, I, NewPredicate, NewIthArgument) ←
    K <= L,
    K = I, !,
    arg(I, NewPredicate, NewIthArgument),
    KNew is K + 1,
replace(K, L, Predicate, I, NewPredicate, NewIthArgument) ←
    K <= L, !,
    arg(K, Predicate, Arg),
    arg(K, NewPredicate, Arg),
    KNew is K + 1,
```

The implications of this extension is twofolds. First, the user dialogue is improved and unnecessary queries and assertions are eliminated based on the properties of functions defined. Second, these properties of the predicates can be used in the explanation phase of the dialogue for indicating the inferences of the system.
solve(true, P) ← !.
solve(A, B, P) ← !,
    solve(A, P),
    solve(B, P).
solve(A, P) ←
    clause(A, Body),
    solve(Body, [clause(A, Body)|P]).
solve(A, P) ←
    askable(A),
    query_user(A, P).

Figure 6.1: Technique for Collecting the Current Path

Why explanations are provided by extending the query_the_user axioms. However, the basis skeleton should be enhanced to collect the current line of reasoning and displayed by an enhancing query_the_user with a second input argument. This enhancement is by collecting the rules traversed to show the current line of reasoning. A sample extension based on the vanilla interpreter is given in Figure 6.1. This method uses an accumulator to collect the current path in the search tree as a stack of goals.

As seen in Figure 6.1, the path argument P is passed to the query_the_user to be presented for why explanation. In [SL86, SS86], this stack is popped out every time the user poses a why question to illustrate the current line of reasoning in chronological order. Naturally, the explanation based on only the path leading to the current goal cannot illustrate the completed portions of the search tree, the completed deductions for previously solved goals. A "global picture" of the computation cannot be provided with this method.

Recall that a proof tree represents the successful deductions in the computation after it terminates. A fully instantiated proof tree is not available for explanations during the computation. However, a partially instantiated tree which has full branches for completed portions of the computation and uninstantiated nodes for incomplete portions can be made available. We call this structure a partial proof tree.

A partial proof tree is easily generated by adding two extra arguments to
a skeletal interpreter, i.e. vanilla. It should be considered as an extension to the shell obtained by using the collect technique to construct a Proof. The first argument represents the partial proof tree and the second one represents the proof tree just like the Proof argument described for collecting the proof tree.

In the beginning of the computation, these two additional arguments represent the same entity, the uninstantiated tree. This is indicated by a call to the interpreter given below.

\[ \text{solve} \text{\_goal}(\text{Goal}, \text{Tree}, \text{Tree}). \]

However, the branches of the proof tree are collected by using the second argument which is the current node of the partial proof tree. The nodes are constructed similarly by the Proof argument during the computation by the second extra argument and the full partial proof tree is available in the computation by using a global variable Tree. The tree is instantiated at its leaves when computation finitely succeeds or fails. For example, the partial proof tree is generated as follows for a conjunctive goal. Using the same technique to extend the other clauses of the interpreter is obvious.

\[
\text{solve}((\text{A,B}),\text{Tree},(\text{PA,PB})) \leftarrow !, \\
\text{solve}((\text{A}),\text{Tree},\text{PA}), \\
\text{solve}((\text{B}),\text{Tree},\text{PB}).
\]

The partial proof tree is a useful structure for why explanations. We believe that this is another form of collect technique. The variable Tree makes the full tree available for inspection at any time during the computation. A how explanation of the partial proof tree can be given during the computation for a global view. The explanation has to incorporate the partial instantiations for clauses and informs the user which goals which have not yet been solved.

This technique is actually quite similar to constructing the search tree presented in Section 3.4.2. However, as we pointed out, the idea in constructing the tree is for inspection. Unlike the interpreter presented there, we use the base level interpretation, which is Prolog's, to hook up the leaves of the tree, etc. for constructing the tree which is automatically provided by unification.
The other difference is the use of non-ground representation. There is one problem which is a well known problem for using non ground representation here. In presenting a how explanation, the uninstantiated nodes in the tree should not be instantiated during the explanation phase. Our goal is to inspect the tree during explanation. Modification is handled by the Prolog interpreter.

### 6.2.3 Techniques for Whynot Explanation

The *whynot* explanation requires a trace of the failed branches in the search tree. This type of explanation is given when the system can not find a solution to a query. It is the trickiest kind of explanation. Two different aspects of failure trace are important for providing whynot explanations.

- obtaining the trace.
- to select and abstract from the failure branches and employ a strategy in presenting them.

Obtaining a trace might look obvious at the abstract level. However, it is crucial in terms of using the appropriate skeleton for this task. There are two key points in obtaining the trace of failure branches in the computation.

- Explicit representation of failure branches in the computation.
- Representation of multiple lines of reasoning.

We argued in Chapter 4 that vanilla interpreter is not an adequate skeleton for multiple lines of reasoning with both successes and failures. The vanilla interpreter does not compute any results on failure. In order to provide explanations for Prolog programs, researchers have used a second interpreter to re-search the search tree to construct a *proof of failure*. For example, a separate interpreter `solve_failure` is used as a skeleton to construct a proof of failure as given below.

```prolog
solve_goal(Goal, Proof) ←
    solve(Goal, Proof);
    solve_failure(Goal, Proof).
```
This method is used in [Ham84, Nib84, SL86] and the skeleton is discussed in detail with the name prove_failure in [BH88].

There is one problem in having two different skeletons as basis to collect proofs of both success and failures. If the system allows user interaction and thus assertions to/deletions from the knowledge base, re-searching the search tree will not generate the actual proof of the deduction if the search tree changes. Further, a recomputation can propagate more changes. This method is not adequate if we want to generate the original proof tree. In order to overcome this problem we can use a skeleton which can model both successes and failures without recomputations. We have shown in Chapter 4 that the layered abstraction provides us to model both single line of reasoning and multiple line of reasoning. In the next section, we illustrate the utility of the layered interpreter central to an explanation system.

6.2.4 Other Types of Explanation

The types of explanation that we have discussed so far have been based on an abstraction of the trace of the whole deduction. Except for the by whom explanations, the other type of explanations are not addressed in Prolog expert systems in detail. They require additional information about the facts/rules in the system. We can consider them as meta-level specification of the rules/facts, such as what kind of explanations. In order to be able provide explanations for when type of explanations, the skeleton has to extended to represent temporal concepts explicitly.

6.3 XPL Shell for Explanation

6.3.1 Overview of XPL

We use the layered interpreter as the basis for an improved explanation system for Prolog XPL (Explanation for Prolog) in Prolog. We also developed a

\footnote{The interpreters presented there are meta-level interpreters not meta-interpreters since they use the cut but do not formulate the cut in the definition of the object language.}
simple interface which allows the shell to run under X windows.

The motivation in developing XPL is to

- provide explanations for successes and failures by using the same skeleton and therefore avoid generating wrong explanations when user assertions/deletions are allowed.

- extend the explanation facilities that are previously explored in a unified framework.

- illustrate the use of non ground representation for constructing the shell to explore the advantages and limits of this approach.

The explanation shell has the structure that are presented in Figure 6.2.

It is very straightforward to develop the shell's top level interface in Prolog as explained in [SS86]. The user interaction with the system is similar to interaction with Prolog. A Prolog query can be directly solved by using Prolog's interpreter or by the layered interpreter. The meta-interpreter is activated for the queries as follows.

\[ \leftarrow \text{explain}(\text{Query}). \]

The layered interpreter constructs a derivation tree for generating explanations. A derivation tree is a portion of the search tree depending on how the computation terminates. If a computation terminates successfully, a proof tree is constructed. Otherwise, if the computation finitely fails, a search tree is constructed. If there was a solution to a query and there are no alternate solutions to the query, the portion of the search tree which is traversed since the last solution is constructed for explanation. Recall that explanations are provided both during and after the termination of the computation. Explanation is based on either the partial proof tree or the proof or the partial search tree depending whether a solution could be obtained for the Query. As this is a generalized structure, we call it a derivation tree. The explanation module provides extended browsing capabilities which are described later.
Figure 6.2: The Architecture of XPL Shell
6.3.2 Extended Layered Interpreter for Constructing Derivation Trees

The heart of the explanation shell is the layered interpreter. We use the interpreter in Figure 4.5 to generate the derivation tree. Recall that the interpreter simulates Prolog's behaviour. It is backtrackable, it can be used to find alternate solutions to a query just like Prolog. There is one difference in what the layered interpreter models in contrast to Prolog since backtracking is explicit. The no result obtained at the top layer can be regarded as follows:

- there are no solutions for this query.
- there are no more solutions for this query upon backtracking.

Therefore, the interpreter can be used to represent the predicates explicitly which fail to generate alternate solutions. Representing this information can be useful in explanation to illustrate the rules which are forgotten to be included in the knowledge base.

We have discussed the collect technique to construct a proof tree in Chapter 4. The layered skeleton is enhanced to collect derivations similar to the vanilla skeleton. However, there is one issue we have to address in using the layered interpreter, how to represent the failure branches in the derivation tree. The layered interpreter constructs a single tree as proof for both successes and failures. If the computation successfully terminates, the proof tree is obtained. If the computation terminates with failure, a search tree is obtained. A derivation tree has subtrees that correspond to the proof tree of a successful computation of a goal. The failure nodes and success nodes are identified differently in building the tree. The explainer uses the type of the node to identify whether a node indicates successful or failed computations.

The top layer of the enhanced interpreter is given below.

```prolog
solve_top(Result, Goal, Proof, Tree) ←
    label(Goal, Name),
    solve_bottom(Result, Goal, Proof, Tree),
```
filter_result(Result, Goal, Name, Proof).
solve_top(Result, Goal, Proof, Tree) ←
    label(Goal, Name),
    determine_result(Name, Result, Goal, Proof).

determine_result(Name, no, Goal, fail(Goal,Failset)) ←
    get_proof(no, Name, Failset\[]).

We do not have a look ahead mechanism, just like the Prolog interpreter, to indicate whether we have a solution or not till we find a solution, or completely exhaust all the possible search paths. Similarly we collect the branches in the search tree until we find a solution. If a solution is obtained at the bottom layer, the branches which are collected so far at the top layer are discarded. Otherwise, all the failure branches are collected to form a subtree and identified with the node fail(Goal,Proofs), where Proofs indicate the set of failed branches ordered chronologically. If we do not want to keep backtracking information explicit, the constructed derivation tree can be used by the interpreter itself to adjust its interpretation. For example, obtaining an empty set of derivations, i.e. fail(Goal, []) for a Goal indicates that there are no more solutions for this goal. At the top layer, we can eliminate these goals by simple failure.

The label predicate assigns a Name for each Goal. In this extension, the ground representation of a goal is used as a name to identify the set of derivation paths that are constructed in searching a solution for the goal. This set consists of chronologically asserted paths to Prolog's database by the using the name of the goal as an name (index).

The interpreter asserts derivation paths and trees by get_proof when the computation terminates successfully, and retracts them using store_proof when it terminates by failure at the top layer. The predicate store_proof is used to store the branches of the proof temporarily based on the Name which uniquely identifies a goal. Filtering discards the previous failure branches as indicated below.
filter_result(yes, RealGoal, Name, Proof) ← !,
    get_proof(yes, Name, ProofList[[ ]]).
filter_result(commit, Goal, _, Proof) ←
    not predicate(Goal), !.
filter_result(commit, Goal, Name, Proof) ←
    store_proof(Name, commitgoal(Goal, Proof)), !, fail.
filter_result(_, Goal, Name, Proof) ←
    store_proof(Name, Proof), !, fail.

Next we ask the question: Is ground representation sufficient to identify
the derivation paths uniquely that are collected for derivation of each goal? This is a dynamically asserted set and we have to guarantee that the derivation
paths we are obtaining from assertions do actually correspond to the solution of
the current goal. This naming scheme is sufficient as follows. Only a goal and
its variants are mapped to the same name. Therefore, it is sufficient to show
that variants of the same goal are not accessing the derivation paths of one
other dynamically in the computation. We assume that the search strategy is
fixed. The solve_bottom represents the traversal of the search tree one branch
at a time. The variants of the same goal can appear in the same search tree.
However, unless a variant of a goal appears as one of the descendants of the
goal in the tree, they can not interact by asserting derivation paths. When the
variants of a goal appear in different paths in the search tree, their access to the
database will be independent of each other as the search model is fixed and the
search strategy does not access to different branches of the same tree in parallel.
As we rely on Prolog's model, it is safe to make this assumption. Therefore
only variants which appear as a descendants of each other can interfere. Such
occurences in the search tree indicate a computational loop. This shell does
not address loop handling. A loop in the object level interpretation causes
looping at the layered meta-interpreter although finite failure is represented
explicitly. As discussed in Chapter 4 for Warren's OLD skeleton, an extension
for goal suspension is necessary to extend the layered interpreter's capabilities.

---

2This naming scheme is discussed in Section 3.1.2.
However, the variants of the same goal are again used as a single entry in the extension tables in Warren's approach as well.

The derivation tree has the following type of nodes, which are represented as *Proof*:

- *clause(Goal, Body, Proof)* indicates a Prolog clause and the proof of derivation of the body. The body is explicitly represented to make explanation generation easier.

- *sys(Goal)* indicates a Goal which is solved successfully at the object level.

- *findall(Vars, Goal, List, ProofSet)* indicates the successfully terminated set predicate findall and the set of Proofs, which can be empty\(^3\). *ProofSet* is a non-empty, i.e. \([\text{Proof}[\text{ProofSet}])*, set of Proofs ordered chronologically.

- *commitgoal(Goal, Proof)* indicates a failed computation for a Prolog clause due to a commitment in the body. This distinction is used in order to distinctively identify the cause of failure.

- *fail(Goal, ProofSet)* indicates the set of derivations as Proofs for finite failure of the goal.

- *notclause(Goal)* indicates a Goal which is defined as a Prolog clause.

- *sysfail(Goal)* a Goal which could not be solved at the object level successfully.

- *unsearched* indicates an unsearched portion of the tree.

- *cut* indicates cuts.

- *userfact(Goal)* indicates facts asserted by the user.

- *userfail(Goal)* indicates the assertions for Goals which are specified as untrue by the user.

\(^3\)Recall that findall always succeeds even if the List is empty.
The query

\[ \text{succeed}(R, p(X), \text{Proof}). \]

% Given the program

\[ p(X) \leftarrow q(X), !, r(X). \]
\[ q(b). \]
\[ p(a). \]

% Generates a derivation tree with the computation results:

\[ \text{Proof} = \text{fail}(p,.64), \]
\[ [\text{commitgoal}(p(b),\]
\[ \text{clause}(p(b),(q(b),!,r(b)),\]
\[ (\text{clause}(q(b),\text{true},\text{fact}),\text{cut},\]
\[ \text{fail}(r(b),[\text{notclause}(r(b))]))))]),\]
\[ R = \text{no}, \]
\[ X = .64 ? \]

Figure 6.3: Example derivation tree of a failure due to a cut

- \text{not_goal}(\text{Goal}, \text{Result}, \text{Proof}) indicates a negated goal and its proof.
- \text{fact} indicates a fact (when the body of a clause is empty).
- \text{(PA, PB)} indicates conjunction of proofs.
- \text{(PA; PB)} indicates disjunction of proofs.
- \text{(PA -> PB)} indicates conditional (if-then) proofs.

The failure nodes can be nested. The context in which the nodes appear determine whether they are success or failure nodes. Consider the example in Figure 6.3.

This simple example illustrates another point that is important in handling variable representations and their instantiations in the tree. We used Prolog's non-ground representation by constructing the tree. For representing the proof
of the object level computation, the vanilla skeleton might be sufficient. However, here the failure branches should be represented with appropriate instantiations without instantiating the original query. In the example above, the layered interpreter neither fails by the failure of the object level computation nor computes a result except indicating the result of the computation. The terms and the substitutions should be represented explicitly in order to give an explanation of the failure. We can not represent an uninstantiated term and the same term after applying a substitution in the same data structure without using the ground representation for multiple lines of reasoning, i.e. to represent several failure paths where the same goal can be instantiated. Using ground representation of a term at the meta-level and explicitly representing the unification of a term given a substitution is discussed in [BHL90]. The object level interpreter in Section 3.4.2 also uses the same approach [van89b].

In contrast, we exploit some of the features of the layered interpreter by using an implicit scoping mechanism. The second clause at the top layer generates a node fail(\textit{Goal}, \textit{ProofSet}) to explicitly label the failure branches of the particular goal. Instead of keeping a set of variables that are instantiated, we keep the actual goal which is used to reunify the variables during the explanation. Since the explanation provides browsing capability in the tree, the underlying unification is exploited for reconciling the variable bindings. In Prolog, assertions/deletions also generate new variables which was discussed in Section 3.1.2. As the individual derivation paths are stored by using this method, the labelling method is proven as a useful technique. The labels of the same goal at higher nodes in the tree have precedence in unification. Consider the example in Figure 6.4.

By using the types of nodes in the tree, we actually unify .64, .224, and .237 during the explanation. There is one point we exploit which is common in Prologs which makes life easier. Chronologically, the variables that are generated earlier take precedence in unification. Therefore, the precedence in the tree is preserved while retraversing the tree in the explanation.

Using the nonground representation has one very significant disadvantage in representing derivations: representation of disjunctive goals and their
The query

\[ \text{solve_top}(R, p(X), \text{Proof}). \]

Given the program

\[ p(X) \leftarrow q(X). \]

Generates a derivation tree with the computation results:

\[ \text{Proof} = \text{fail}(p(\_64), \]
\[ \text{[clause}(p(\_224),q(\_224), \]
\[ \text{fail}(q(\_224),[\text{notclause}(q(\_237))]))].) \]

\[ R = \text{no}, \]
\[ X = \_64 ? \]

Figure 6.4: Example derivation tree for failure due to non existing axioms derivations. For example, when trying to find solutions for \((A;B)\), if A fails and we obtain a solution for B, we have to be very careful about keeping the variables disjoint in the disjuncts. The derivation tree has failed branches for A, and the proof subtree for B. We used a copy of the disjuncts to avoid instantiating the variables in the tree.

6.3.3 The Explainer

The explainer uses the derivation tree as the basis for generating explanations. The browsing capabilities of the explainer is based on two things.

- the type of nodes in the tree.
- the strategy that is used to browse and display the tree.

Hammond [Ham84] suggests that a how explanation can be given when a user asks for a whynot explanation for a successful goal. Recall that we use the same data structure to generate the explanations unlike previous research. We represent the successes and failures in the same tree and distinguish them
by the type of nodes in the tree, therefore the success or failure of the computation regarding the current node in the tree is known. Therefore, we did not distinguish between how and whynot type of explanations since we know the result of the computation and our basis for explanation in based on browsing through the tree.

The easiest one, perhaps, is to imitate Prolog's behaviour in displaying the search and failure. In [SS86], such an exhaustive way of displaying the tree is presented. Therefore, the explanation generation might look similar to the implicit search of the search tree. We based the explanations on level by level presentation of the goals in the tree. At any point

- detailed explanation of deduction of one of the subgoals, if any, is possible. This is similar to zooming discussed in [Duc88].

- visiting previously visited nodes.

Our search strategy in displaying the tree is different, it is not depth first search. It is guided by the user to zooming in and zooming out of different portions of the tree. Therefore, the skeleton for displaying the tree is not similar to the one which generated the tree.

At a node, zooming is handled by assigning a number as a label to each non terminal subtree in the derivation tree, at one level below the current node. The subtrees which do not require zooming are presented immediately and not labelled for explanation. For example, the facts are handled in this manner. We deliberately chose to display the rules and the order they were considered in the inference at each level, and have given the user the chance to browse a particular subtree. However, it is possible to change this behaviour and adopt a different explanation strategy.

Hammond correctly observes that explanation of failure requires not only displaying the first failure [Ham84]. Explaining failures can be tricky, as there is no established strategy to determine which way we should explain failures. For example, the failure of a goal can be related to the first failure [Ham84], the last failure branch [Nib84] or the longest failure branch [SL86] in the tree depending on the problem domain. It is rather hard to have a pre-established
strategy to identify the cause of failure based on a property of the derivation tree. In presentation of the cause of failure, it is possible to adopt one of these schemes. We provide uninstantiated rules in failure branches and display the computation at one level in the order the derivations are generated for displaying failures.

In Figure 6.5 and Figure 6.6, we give some of the code for the explainer. The main purpose of the exp_one function is to label the subtrees and present them to the user, i.e., which are the Choices, get the users requests for directing the explanations ask_user, and carry out these instructions act_on.

The skeleton for generating explanations is explain enhanced with a tabbing function which enables updating the tab stops for pretty printing of rules and facts, and the labelled nodes in the tree as a difference list. The Choices determine the type of questions or the browsing capabilities of the explainer. The nodes in the tree which might be explained further are collected in the list of active choices, which might be guiding the explainer. The nodes which might be further explained are labelled during the browsing and presentation of the tree to the user. The user might request more information about the predicate at the current node if additional information regarding the node can be given. This is especially useful for whynot type of explanations.

Depending on the type of the node, the success or fail nodes of the derivation tree are identified. At the top level, explain assigns explanation to the appropriate rules. The rules for exp_one_level are used by providing explanations to illustrate both successes and failures at one level below the current node in the derivation tree. This predicate explains only some the leaves of the tree directly as illustrated in Figure 6.6. When there might be more elaborate explanations that can be provided about a Goal, such as failed branches due to non-existing clauses in the knowledge base, this predicate labels the node in the tree and defers the explanation which might be requested by the user. The detailed explanation can be requested by the user later if desired.

In Figure 6.8, we illustrate the general structure of handling failure branches by the handle_failure predicate. As we have discussed earlier, there are different strategies that we can choose to display failure. Given a particular
exp_one(Goal,Proof) ←
  explain(Goal,Proof,0,0,N,Choices\[ ]),
  determine_context(N, Choices),
  ask_user(N, Choice, Choices), !,
  act_on(fulltree,Choice,N,Goal,Proof,Choices).

explain(true, fact, Tab, N, M, Xs\Xs) ←
  writes_truth, !.
explain((GoalA,GoalB), (ProofA,ProofB), Tab, N, M, List) ←
  exp_one_level((GoalA,GoalB), (ProofA,ProofB), Tab, N,M,List).
explain(Goal, Proof, Tab, N, M, Xs\Ys) ← % success rule
  rule(Proof, Head, Ref, ProofBody), !,
  write_head(Tab, Head),
  NewTab is Tab + 6,
  exp_one_level(Ref,Proof, NewTab, N, M, Xs\Ys).
explain(Goal, Proof, Tab, N, M, Xs\Ys) ←
  failure(Proof, FailGoal, FailureSet), !,
  handle_failure(Goal, FailGoal, Failures, Tab, N, M, Xs\Ys).
explain(ExtGoal,Proof,Tab,N,M,Choices) ← % explain higher level
  is_extension_goal(ExtGoal,Proof),!,
  explain_ext(ExtGoal,Proof,Tab,N,M,Choices).
explain(Goal,OtherType, Tab, N, M, Choices) ←
  % all the rest can be explained by
  % the lower level
  exp_one_level(Goal,OtherType,Tab,N,M,Choices).

Figure 6.5: Code for the Explainer: Top Level
exp_one_level((GoalA,GoalB),(ProofA,ProofB),Tab,N,M,Xs\Zs) ←
exp_one_level((GoalA,ProofA),Tab,N,NNew,Xs\Ys),
exp_one_level((GoalB,ProofB),Tab,NNew,M,Ys\Zs).

exp_one_level(Goal, Proof, Tab, N, N, Xs\Xs) ←
  user_untrue(Goal, Proof),
  write_user_untrue(Tab, Goal).

exp_one_level(Goal,Proof,Tab,N,M,[M-fail(Goal,[Proof])|Xs]\Xs) ←
  fail_due_commitment(Proof, _, _), !, M is N+1,
  write_failing_goal(Tab, Goal, 1, M).

exp_one_level(Goal,Proof,Tab,N,N,Xs\Xs) ←
  no_alternative_solution(Goal, Head, Proof), !,
  write_no_alternative_solution(Tab, Goal, Head).

exp_one_level(Goal,cut,Tab,N,N,Xs\Xs) ← !.

exp_one_level(Goal,unsearched,Tab,N,N,Xs\Xs) ← !,
  write_non_investigated(Tab,Goal).

exp_one_level(Goal,Proof,Tab,N,N,Xs\Xs) ←
  fact(Proof, Head), !,
  write_fact(Tab,Head).

exp_one_level(Goal, userfact(A), Tab, N, N, Xs\Xs) ← !,
  write_user_fact(Tab,A).

exp_one_level(Goal,Proof,Tab,N,N,Xs\Xs) ←
  single_solution(Proof, Head), !,
  write_fact(Tab, Head).

exp_one_level(Goal, Proof, Tab,N,M,[M-Proof|Xs]\Xs) ←
  rule(Proof, Head, _Ref, _ProofBody), !,
  M is N+1,
  write_clause(Tab, M, Head).

exp_one_level(Goal, Proof, Tab,N,M,Xs)←
  failure(Proof, FailGoal, FailSet), ! ;% failures for goal.
  if_necessary_unify(Goal, FailGoal),
  count_failures(FailSet, FailCount),
  upon_count(FailCount, FailGoal, FailSet, N, M, Xs).

upon_count(1, (A,B), [(FailA,FailB)], N, M, Xs\Ys) ← !,
  % single failure explain immediately.
  exp_one_level(A,FailA,Tab,N,Next,Xs\Zs),
  exp_one_level(B,FailB,Tab,Next,M,Zs\Ys).

upon_count(N, FailGoal, FailSet, N, M,
  [M-fail(FailGoal,FailSet)|Xs]\Xs) ←
  N > 1, M is N + 1,
  write_failing_goal(Tab, FailGoal, 1, M).

Figure 6.6: Code for the Explainer: One Level Explanations
% Miscellaneous definitions for distinguishing nodes in the tree

rule(clause(Head, Ref, Proof), Head, Ref, Proof).
fact(clause(Head,X,fact), Head).
failure(fail(Goal,FailSet), Goal, FailSet).
single_solution(clause(Head,Ref,cut), Head)).
user_untrue(Goal, fail(Goal, [userfail(Goal)])).
fail_due_commitment(commitgoal(Goal, Proof), Goal, Proof).
no_alternative_solution(Goal,Head,clause(Head,true,fail(true,[],[]))).
no_alternative_solution(Goal,Head,fail(Goal,[])).

if_necessary_unify(GoalA, GoalB) ←
(more_general(GoalB,GoalA) → GoalA = GoalB);true).

Figure 6.7: Code for the Explainer: Identifying Nodes

predicate

• There are different rules whose heads will unify with this rule, but fail.

• For each such failed rule, there are different reasons why a body of a rule
  will fail.

The difference between these cases are identified. For example, the former
is handled by get_all_rules, and the latter is handled by body_of_goal.

This predicate again uses the explanations provided by exp_one_level.
Therefore, explanations of success and failure nodes are interleaved. The
Choice list, which is a difference list, has the nodes in the tree which can
be further explained if requested.

In Figure 6.8, some of the code for describing the failure types are given.
For example, there might be no clauses for the given predicate in the knowledge
base, or there might be other predicates with the same name and arity, but
which do not unify with the current goal. The latter might indicate that
certain facts might be missed from the knowledge base, therefore they are
distinguished by the two clauses given below. Later, if the user requests, the
existing predicates can be identified by using the Choices list. An example is
presented in Section 6.3.4.
handle_failure(Goal, FG, [Failure], Tab, N, M, List) ← !,
    compare(Goal, FG, Failure, Tab, N, M, List).
% explain a single failure immediately
handle_failure(Goal, FG, Failures, Tab, N, M, List) ←
    predicate(FG),
    get_all_rules(Goal, FG, Failures, Tab, N, M, List).
handle_failure(Goal, FG, Failures, T, N, M, List) ← % |failures| > 1
    mark_failures(Goal, FG, Failures, T, N, M, List).

mark_failures(Goal, FailGoal, [ ], Tab, N, N, Xs\Xs) ← !.
mark_failures(Goal, FailGoal, [FailReason|Failures], Tab, N, M, Xs\Zs) ←
    compare(Goal, FailGoal, FailReason, Tab, N, NN, Xs\Ys),
    mark_failures(Goal, FailGoal, Failures, Tab, NN, M, Ys\Zs).

% explain a single failure reason depending the type of failure

compare(Goal, FailGoal, FailReason, Tab, N, NN, Choices) ←
    non_existing_goal(Goal, FailGoal, FailReason, Tab, N, NN, Choices), !.
compare(Goal, FailGoal, FailReason, Tab, N, NN, Choices) ←
    body_of_goal(Goal, FailGoal, FailReason, Tab, N, NN, Choices), !.
compare(Goal, FailGoal, FailReason, Tab, N, NN, Choices) ←
    due_cut_in_body(Goal, FailGoal, FailReason, Tab, N, NN, Choices), !.
compare(Goal, FailGoal, FailReason, Tab, N, NN, Choices) ←
    is_meta_goal(FailGoal), !,
    explain_meta(fail, FailGoal, FailReason, Tab, N, NN, Choices).
compare(Goal, FailGoal, FailReason, Tab, N, NN, Choices) ←
    goal_itself(Goal, FailGoal, FailReason, Tab, N, NN, Choices), !.

Figure 6.8: Failure Handler
The `body_of_goal` predicate presents all the reasons of failure at one level below the current failed node in the tree one by one. As each failure path indicates different instantiations for the clause, we number each path and present the instantiated version of the rule. This is similar to a replay of generating all the search paths at one level without researching the whole tree. Since we are using a non ground representation, we have to unify each failure path for the body of the goal and resolve the different variable instantiations which occur due to `assert/retracting` the proof paths. The predicate `explain_all` indexes each such path and generates them one by one in a failure driven repeat loop. If we have adopted a different strategy, like displaying each failure one by one, we could have written this more declaratively. The `flat` predicate flattens out the failure nodes, where failed paths might be nested to get individual failure paths. Due to failure driven loop, the `Choices` list has all the failed branches, which are regenerated if the user wants further explanations about these branches.
The explanation shell presents the rules that fail due to commitment first. If the user requires to see the rules which are not reachable due to commitment, this is handled by `act_on` predicate which displays the set of rules, based on the Choices collected by the explainer.

%%% cut

due_cut_in_body(Goal, GoalCopy, Proof, Tab, N, M, 
    [o-(GoalCopy, Ref)|Ys]\Zs) ←
    fail_due_commitment(Proof, GoalCopy, clause(GoalCopy, Ref, Failure)),
    write_goal_with_cut(Tab, Goal, GoalCopy, Ref),
    body_of_goal(Goal, GoalCopy, 
        clause(GoalCopy, Ref, fail(Ref, [Failure])), Tab, N, M, Ys\Zs),
    find_cut(Ref, empty, Place_Cut),
    write_place_of_cut(Tab, Place_Cut).

Identifying such goals requires explicit knowledge about the search strategy since the order of the clauses is important in determining which clauses are not reachable due to the occurrence of a cut.

unreachables(Goal, Ref) ←
get_unreachable_goals(Goal,Ref,Unreachable).

get_unreachable_goals(Goal, Body, Unreachable)←
findall(R, clause(Goal,Body), Things),
partition(Body, Before, Unreachable).

The display of the partial proof tree for why explanations is similar to the
displaying the derivation tree, however the uninstantiated nodes in the tree de-
terminate whether we can explain a node fully or partially. A variable occurring
as a node of the derivation tree indicates that we have not investigated this
node. Therefore, the explanation of these nodes indicate that we are currently
trying to prove these nodes and which of the branches are not fully searched
yet. For example, in a conjunctive node (ProofA,ProofB) if the second con-
junct ProofB is a variable, we are still trying to solve the first conjunct as
given by the program segment for exp_par below. An example for an
explanation in section 6.3.4 within the context of explaining negation.

exp_par((A,B), (PA, PB), Tab, N, M, C, Next) ←
not_yet(PB), !, %that means we are still evaluating the first conjunct
exp_par(A, PA, Tab, N, M, C, Next),
write_rest(Tab,B).
not_yet(X) ← var(X).

We have differentiated between explaining complete derivation trees or
partial proof trees, although the skeletons are similar. The rules for explaining
the derivation tree are used to explain the nodes in the partial proof tree that
represent completed computations.

We have used a slightly different strategy for generating explanations for ex-
tension goals, such as if-then-else, disjunctions, etc. The rules for explain_ext
identifies the goal type and the result of the computation, generates an expla-
nation about the status of the computation by examining the proof and result
of the computation based on predefined explanation tables for such goals. Af-
terwards, explanation illustrates specifics of the computation depending on
how we chose to write explain the current node. For example, a full explanation of a negated goal is given after displaying the result of the computation of the negated goal as illustrated below. The explanation of findall goal depends on the result of the computation, which is determined by examining the Proof. This goal will always succeed even if it does not compute any results, i.e. by returning an empty list of solutions. The first clause is used to explain this case.

\begin{verbatim}
exp_ext(findall(empty), findall(X, Y, Z),
       findall(X,Y,Z, [Proof]), Tab, N, M, Choices) ←
   write_tab(Tab, ['Here is the proof of the failure : ']),
   NewTab is Tab + 5,
   explain(Y, Proof, NewTab, N, M, Choices).
exp_ext(findall(success), findall(X,Y,Z),
         findall(X,Y,Z, Proof), Tab, N, M, Choices) ←
   do_all_list(1, X, Y, Z, Proof, Tab, N, M, Choices).
\end{verbatim}

\noindent \% not
\begin{verbatim}
exp_ext(not(Result), not Goal, not_goal(Goal, Result, Proof),
        Tab, N, M, Choices) ←
   NewTab is Tab + 5,
   explain(Goal, Proof, NewTab, N, M, Choices).
\end{verbatim}

\begin{verbatim}
do_all_list(Num,X,Y,[ ],_,Tab,N,N,Xs\Xs)←!.
do_all_list(Num,X,Y,[First|Rest],[Proof|Proofs],Tab,N,M,Xs\Ys)←
   write_tab(Tab,[Num, '  ', ' = ', First, ' as : ']),
   get_goal(Proof, Goal),
   NewTab is Tab + 5,
   NewNum is Num + 1,
   exp_one_level(Goal, Proof, NewTab, N, NN, Xs\Zs),
   do_all_list(NewNum, X, Y, Rest, Proofs, Tab, NN, M, Zs\Ys).
\end{verbatim}
The explanation shell uses predefined descriptions or formats of facts by exploiting the portray function available. The hook for the explainer is a meta-rule \texttt{\$convert(\textit{Goal, Translatedform})}, which described the format of printing the predicates and questions. If there are no descriptions for predicates, they are displayed as is.

We also developed a help facility based on the Choices that are available during explanation. Depending on the available choices, we have extended the browsing capabilities of the explainer by additional commands which illustrate missed goals. An example is given in Section 6.3.4.

6.3.4 Examples

In this section, we will try to illustrate the interface and explanations provided by \textit{XPL} shell.

Consider the following simple rules for selecting a school for graduate studies in Figure 6.9 and the sample data set in Figure 6.10.

Below we present an example dialogue to illustrate the capabilities of \textit{XPL} shell using the rules as knowledge base. The user responses are emphasized.

The user poses the query \texttt{\$explain(\textit{consider(ann,X)})} to \textit{XPL} and the proof of deduction is displayed. After a solution is found the user requests the shell to find alternate solutions similar to Prolog. The derivation tree for failure to find an alternative solution is displayed. This is the portion of the search tree traversed since the last successful derivation.

In addition, we displayed that there are rules for the \texttt{\textit{wants/2}} relation in the knowledgebase, but not regarding \textit{ann}. Note that we have discarded all the failed rules, i.e. the first two rules for \textit{consider} when we found a solution. The layered skeleton might be modified to explicitly represent such failed derivations to represent the whole search tree if desired.

\texttt{\$\$? explain(\textit{consider(ann,X)})}.

\textit{Your Query ann should consider nyu has succeeded.}
consider(Student, School) ←
  interested(Student, Subject),
  program(School, Subject),
  not conflict(Student, School).
consider(Student, School) ←
  personality(Student, shy),
  size(School, small).
consider(Student, School) ←
  wants_in.town(Student, City),
  is_in(School, City).
consider(Student, School) ←
  wants(Student, Quality),
  has(School, Quality),
  not conflict(Student, School).

conflict(Student, School) ←
  cost(School, Tuition),
  afford(Student, Price),
  Tuition > Price.
conflict(Student, School) ←
  dislike(Student, School).

has(School, Quality) ←
  is_in(School, City),
  area_has(City, Quality).

% Meta level predicates for defining askable predicates
askable(afford(X,Y)) ← nonvar(X).
askable(dislike(X,Y)) ← nonvar(X), nonvar(Y).

Figure 6.9: Small Expert System for Selecting Graduate Schools
% School data
program(cwru, ai). program(mit, ai). program(cornell, math).
program(oberlin, Subject) ← type(Subject, liberal_arts).

size(cwru, small). size(osu, big).

has(osu, greek_life). has(csu, commuter_school).

interested(umit, ai). interested(umit, music).
interested(bob, ai). interested(ann, art).
cost(oberlin, 12000).

type(art, liberal_arts). type(music, liberal_arts).
type(philosophy, liberal_arts).

is_in(cwru, cleveland). is_in(csu, cleveland).
is_in(osu, columbus). is_in(northwestern, chicago).
is_in(cornell, ithaca). is_in(nyu, new_york).

area_has(cleveland, arts). area_has(cleveland, culture).
area_has(chicago, big_city). area_has(amhurst, small_town).

afford(bill, 12000). afford(bob, 11500).
afford(jill, 100). afford(ann, 10000).

wants_in_town(ann, new_york). wants_in_town(bill, chicago).
wants_in_town(jill, new_york).

dislike(ann, csu).

personality(joe, shy). personality(dave, popular).
personality(jill, popular).

wants(steve, arts). wants(umit, arts).
wants(umit, culture). wants(matt, commuter_school).
wants(tim, big_city).

Figure 6.10: Data for Graduate Schools and Students
ann should consider nyu has been found to be true because
    ann wants to go to a school in new.york is a fact.
yuy is located in new.york is a fact.

> quit.

??>consider(ann,nyu) ? ;

Your Query ann should consider _3607 has failed!

ann should consider _3607 fails for 2 possible rule(s) that are applicable.
ann should consider _3672 fails because
    ann wants to go to a school in _3693 AND
    _3672 is located in _3693 fails (1)
ann should consider _3787 fails because
    ann wants the quality of _3808 AND
    _3787 has the quality of _3808 AND
    ann does not have a conflict with going to _3787 fails (2)

In this example, we have chosen to represent the backtracking information explicitly to illustrate that the user that additional rules/facts in the knowledge base might help finding alternate solution. Zooming through the failed rules illustrates this below. When we zoom to understand why the first rule failed, the computation in the order that the failures occur are displayed for that particular rule.

>1. % for the first rule

1*ann should consider _3672 fails because
    ann wants to go to a school in new.york is a fact.
    _3672 is located in new.york fails (1)
2*ann should consider _3672 fails because
ann wants to go to a school in .3693 fails (2)
._3672 is located in .3693 is not investigated.

>1.

The last solution for _3672 is located in new.york was

    nyu is located in new.york
    and there are no alternative solutions on backtracking!

Later, when we want to see why the second rule fails, we see that although
there are additional rules which failed to unify with the current goal, and hence
illustrate that we have information about other rules for the relation wants
but not for ann. This example also illustrates the help facility which shows
what could be done in this context.

>2.  % for the second rule

1*ann should consider _3787 fails because

    ann wants the quality of _3808 fails (1)
    _3787 has the quality of _3808 AND
    ann does not have a conflict with going to _3787 is not
    investigated

>1.

I don't have any information about ann wants the quality of _3808.
The current goal does not match with the rules of wants/2.

>help.

You have the following options here :

    quit : stops the explanation session
    previous : lets you examine the previous rule
at the top level displays the first rule again...
not_applicable_rules : shows the non unifying rules
with the current goal

> not_applicable_rules.

There is/are 6 facts(s) in the database for wants/2

steve wants the quality of arts is a fact.
umit wants the quality of arts is a fact.
umit wants the quality of culture is a fact.
mary wants the quality of culture is a fact.
matt wants the quality of commuter_school is a fact.
tim wants the quality of big_city is a fact.

quit.

------------------------------------------------------------------------------------------------
no

The no answers indicates that there are no more answers for the query as in Prolog.

Let us give an example for why type of explanation when the user does not understand why the shell is asking a question. The partial proof tree is displayed to show all the computations that are completed so far on the current line of reasoning. For example, we can get a full explanation for how the shell found out that oberlin has a good program in music. Illustrating the failed computations so far, namely the current status of the full search tree is not targeted. We discuss this issue in the last section again.

$?> explain(consider(umit,X)).

Is it true that umit does not like oberlin? why.
In order to prove umit should consider oberlin,
I proved umit is interested in music is a fact.
I proved oberlin has a good program in music (1)
I am trying to prove that
unit does not have a conflict with going to oberlin
by trying to show that this goal fails.

I am trying to see whether a solution for
unit has a conflict with going to oberlin
exists to prove the goal.

In order to prove unit has a conflict with going to oberlin,
you have to tell me whether umit does not like oberlin is true.

A full explanation of negation-as-failure is given below for a negated goal. Assume that we had a fact regarding how much umit can afford in the knowledge base. The following response illustrates that after finding the first success, the shell concludes that there is evidence about the conflict and can not say the negated goal. The explanation is taken from the first rule that defines consider.


1=unit should consider cwruc fails because
unit is interested in ai is a fact.
cwruc has a good program in ai is a fact.
I tried to prove that
unit does not have a conflict with going to cwruc
is correct here. However
I have proven unit has a conflict with going to cwruc (1)

The explanation of the failed branch yields.

unit has a conflict with going to cwruc has been found to be true
because
cwru has a tuition cost of 11000 is a fact.
unit can afford to pay 100 is a fact.
11000>100 is a system predicate which succeeded.

Below we give an example for illustrating the effects of cuts in the system.
Assume that we have an expert system which determines the people that
qualify for Ph.D candidacy with the following rules.

\[
\text{qualifies} \text{(Grad, yes)} \leftarrow \\
\text{passed.math.qualifier} \text{(Grad), !,} \\
\text{qualifies.in.areas} \text{(Grad).}
\]

\[
\text{qualifies} \text{(Grad, no)} \leftarrow \\
\text{bad.committee.consideration} \text{(math, Grad), !.}
\]

Below we give an explanation about how XPL illustrates the reason of failure
due to commitment to the current rule when we try to understand whether
johns qualifies.

$?> \text{explain(qualifies(johns, X))}.$

Your Query whether johns is qualified or not as a Ph.D candidate
has failed!

johns is qualified as a Ph.D candidate fails
due to the presence of a cut in the rule.
1* johns is qualified as a Ph.D candidate fails because
   johns has passed the math qualifier (1)
   johns qualifies for breath and depth requirements fails (2)
   There is a cut after johns has passed the math qualifier

$> \text{other.rules}.$

A is not qualified as a Ph.D candidate is true if
The committee considers the qualification of A in math
insatisfactory.

The other rules query can illustrate the rules which can not be considered due
to the presence of the commitment in the current rule. How we find these
rules were illustrated in Section 6.3.3.

6.3.5 Discussion

The explanation shell $XPL$ has several advantages compared to the previous
explanation shells reported.

- $XPL$ is the first integrated explanation shell which can explain the suc-
  cessful and failed computations, which also might involve cuts, for Prolog
  based expert systems. This feature is important to be able to provide
  accurate explanations when the user interaction for supplying missing in-
  formation is needed. We do not expect the expert system to be a closed
  system, therefore the layered approach provides accurate explanations
  as it is not based on re-computation and losing information in contrast
  to using different interpreters for success and failure.

- The cuts can be used in the knowledge base and its effects for the com-
  putation can be explained.

- The explanation is extended for almost all standard constructs of Prolog.

- The concept of partial proof tree is found useful to provide a global
  picture of the computation for why type of explanations. This approach
  was not taken in previous research.

As any system, $XPL$ has room for improvements. For example, we have
not presented the rules whose heads previously unified with the current goal
but failed if we are in the process of trying to prove a rule for this goal. As its
name suggests, it is the partial proof tree we are presenting, not the partial
search tree. However, by another extension to the layered skeleton, the names
for the goals can be kept in the derivation tree to find the previously failed and stored branches of the derivation tree corresponding to this goal. Although this can be done, a \texttt{melt} is necessary to obtain the variant of the current goal in order to illustrate the branches when the name is given. One idea is to store a variant of a goal along with its name as a separate table and obtain copies by using the \texttt{copy_term} predicate available in Prologs. This method can be used to achieve the function of \texttt{melt_new} to obtain a fresh copy of the goal each time to display the branches and reconciling the variable bindings.

We also did not employ planning explicitly. An explicit plan strategy and a simple trace of the explainer's behaviour would improve the explainer's interaction with the user.

There are certain disadvantages due to the non ground representation. For example, the explanation shell has to induce failure onto itself in order to undo the unifications when we want to view the previously visited nodes. Therefore the failure goals are generated by repeat loops and explicit failure which gives it declarative properties. Again, when disjunctive goals share variables it is not possible to represent the failed disjunct by using the actual variables as they might be instantiated by the succeeding disjunct at the object level. A copy of the disjuncts are used to overcome this problem.
Chapter 7

A Shell for Uncertainty Reasoning in Prolog

7.1 Reasoning with Uncertainty and Prolog

Representing beliefs and reasoning with uncertainty has been a popular topic in AI starting with the famous expert system MYCIN [BS84]. As our understanding of representing knowledge has progressed, the representation of uncertainty, its calculus and its theory has emerged as a field of its own.

The issues in reasoning with uncertainty are twofold, representing the knowledge with uncertainty and the uncertainty reasoning calculus which describe its semantics. Certainty factors are used to represent our degree of belief that our inference is true or the degree of evidence that is supporting the truth of a particular premise. The methods used in reasoning with uncertainty and defining its calculus are divided into three main groups

- Ad-hoc representation and computation of uncertainty. A well known example in this category is the certainty factors used in MYCIN [BS84].
- Bayesian models based on point probabilities.
- Non-Bayesian models using multi valued uncertainties, probabilities.

The examples for multi valued models is using Dempster-Shafer (D-S) theory, incidence calculus and probabilistic logic. A good source which comprises these models and discusses their limitations is by Pearl [Pea88].

Among non-Bayesian models for managing uncertainty, D-S is popular for representing beliefs because it is compatible with the classical proof-based style of inference. D-S computes probabilities of probability rather then computing probabilities of truths. The representation of belief of a proposition A in D-S is given by the interval \([Bel(A), Pl(A)]\), where \(Bel(A)\) represents the strength of
the argument in favor of \( A \) and \( Pl(p) \) indicates the plausibility of the predicate, or the degree of belief that it can not be refuted. \( Pl(A) \) is thus defined as \( Pl = 1 - (Belnot(A)) \). The difference between the belief and plausibility, i.e. the interval

\[
Pl(A) - Bel(A) = 1 - (Bel(A) + Belnot(A))
\]
corresponds to the probability where both \( A \) and not \( A \) are compatible with available evidence. Therefore, the tuple indicates the interval where the probability might lie. The model works with partial answers when posterior and prior probabilities can not be assigned in contrast to Bayesian models.

Incidence Calculus is compatible with D-S where the evidence is collected and counted by using an automated theorem prover [Bun84].

Methods for incorporating uncertainty within logic programming have also been proposed. The representation of uncertainty is achieved by augmenting the representation of each clause by a suitable representation of certainty, which can be single or multi-valued. The first attempt incorporated uncertainty reasoning by annotating each clause by a single value (Clark and McCabe [CM82]). This value indicates the certainty value associated with the solution of the clause if the conditions (premises) of a clause are solvable. However, an additional goal for the calculation of uncertainty has to be given within the definition of each clause in [CM82]. This is due to the attempt to integrate the calculus within the language.

It is well known to use Prolog meta-interpreters for expert system shells for controlling the computation and making selected properties of the deduction explicit. Shapiro [Sha83], separated the calculation and propagation of uncertainty via a meta-interpreter. The major limitation of this work is the inability to combine multiple lines of reasoning that indicate multiple evidences for obtaining the solution of a goal. Examples for integrating this simple form of uncertainties within Prolog computations are given in the text [SS86].

Various ad-hoc uncertainty methods are implemented in Prolog. For example, Prolog versions of MYCIN floating around in the logic programming community inherit MYCIN's uncertainty calculus. There have been also frame-based uncertainty reasoning schemes, such as [YS90].
Integration of uncertainty reasoning with Dempster-Shafer theory with Prolog is proposed by Baldwin under the name Support logic programming in [BM87] based on the open world assumption. Each clause \( p \) is augmented by a two-valued certainty factor \([\text{Bel}(p), \text{Pl}(p)]\). Baldwin also discusses how to combine uncertainties in clauses that are not mutually exclusive by using Dempster's rule of combination. Recently Shekhar and Ramamoorthy provide a meta-interpreter along the same lines[BM87].

Yet another approach is the recent ongoing work of Ng and Subrahmanian [NS90] with annotated two-valued logics. The probability pair value attached to clauses shows the range in which a probability of an event must lie in contrast to D-S formalism. Their intent is to define a language which is totally faithful to probabilistic logic and describe its model theory. Their work is the first to give a probabilistic semantics for expressing quantification in logic programming.

Our intent in this chapter is not describing another logic for integrating uncertainty within logic programming. We take a pragmatic approach and argue that reasoning with uncertainty can be achieved by a flexible meta-architecture that is suitable for describing and reasoning with different representations of uncertainties, combining multiple lines of reasoning and reasoning with unknowns and negation. Shoken and Finin [SF90] propose a similar approach and use a meta-interpreter for describing ad-hoc and bayesian models. Finin's formulation of reasoning with beliefs for expert systems in [SF90] is

\[
\text{inference} = \text{inference} + \text{belief}
\]

\[
\text{mechanism} = \text{engine} + \text{calculus}
\]

The inference engine is based on a meta-level interpreter. The idea is to achieve different inference mechanism by supplying different belief calculi. By reformulation within the context of skeletons and techniques, this formula can be re-expressed

\[
\text{meta-interpreters} = \text{meta-interpreter} + \text{belief}
\]

\[
\text{for uncertainty reasoning} = \text{for} + \text{enhancement} + \text{calculus}
\]

\[
\text{control}
\]
Separate representation of belief calculus from the meta-interpreter is also addressed in [FS85] and [Bot90]. However, the underlying meta-interpreter for both approaches is a vanilla like interpreter, which is incapable of representing and reasoning with multiple lines of reasoning.

We have shown that a layered meta-interpreter can be used as a skeleton to represent multiple lines of reasoning in Chapter 4. We claim that it provides a clean framework which subsumes the earlier work based on meta-interpreters in [Sha83, SS86, FS85, BM87, SR89, SF90, Bot90].

7.2 Issues for Reasoning with Uncertainty in Prolog

In this section, we discuss some of the issues that should be addressed for practical uncertainty reasoning with Prolog\(^1\).

For representing uncertainty within the object language, we assume that the representation of a Prolog clause/rule is augmented by a certainty factor which indicates a conditional certainty of the head of the clause when the body of the clause is true. For example, the clause

$$A \leftarrow cf(CF), B_1, ..., B_n.$$ 

the certainty factor is given by $cf(CF)$ of $A$ when the body of the clause is true. The clauses without certainty factors correspond to clauses with the maximum certainty factor, i.e 1, or [1, 1].

We identify the following issues for calculating the certainty of a predicate.

1. Representing and Combining Multiple Evidence: The result of a successful Prolog deduction simply involves one branch in the proof tree. However, the other branches which provide the same answer substitution in the search tree correspond to multiple evidence that indicates the particular answer and they should be accounted for cumulatively. Operationally, this is

\(^1\)In our discussion, we will use the term \textit{certainty} of a predicate somewhat freely to illustrate the degree of belief that a proposition represented by a predicate is true.
not possible by calculating uncertainties based on a single line of reasoning as in [Sha83, SS86, FS83, Bot90]. Therefore, the architecture should provide multiple lines of reasoning to account for multiple evidence.

There are different calculi for assigning a certainty value to combine multiple evidence [SF90]. In [MW88], the maximum of the pooled certainties from multiple evidence is assumed. For multi-valued certainties, different intervals for multiple evidence are also used for resolving conflicts in combining evidence as in [BM87].

Operationally, getting all the lines of computation and their respective certainty values can be obtained with a set predicate, such as findall. This approach is taken in two current studies of handling uncertainty by using meta-interpreters [SR89, SF90]. In the former, multiple evidence is considered only for the top level query. However, this approach should be integrated at different levels of the computation, by considering different lines of reasoning to compute a composite uncertainty for each rule in the computation. This approach is taken in [SF90] by using findall. However, multiple evidence is not grouped together for each answer substitution. Therefore, their approach is suitable for only ground queries.

2. Combining Answer Substitutions: The second issue concerns reconciling variable bindings. For example, suppose a predicate p/1 has two different solutions, p(1) with certainty p₁ and p(2) with certainty p₂. If we pose the query p(X), it is not clear what the certainty of the answer should be. Since we can not expect the certainty to resemble a distribution function, every ground answer should be handled separately.

3. Handling Negation: Negation for Prolog programs is based on the negation-as-failure rule. Operationally, dealing with negation requires the computation of all lines of reasoning for a goal. All possible lines of reasoning for a negated goal must be obtained in order to show that all of them result in failure as we discussed for the explanation shell in Chapter 6. Handling negation for reasoning with uncertainty requires a similar but a revised approach. We need to consider two issues which are closely related.

a. Handling the missing facts or unknowns in the computation is an im-
portant issue. For example, for a program with facts p(a) and p(b), what can we say about the certainty involved with not p(c)? Our assumptions about the unknown facts naturally shapes how we deal with negation. Single valued probabilities do not allow a distinction between false and unknown values. When there is neither evidence to support or refute a fact, 0 can be assigned conservatively by using a point valued certainty. A multi-valued representation, on the other hand, distinguishes between true, false and unknowns explicitly. A [0,1] pair to indicate full unsureness in the proposition indicated by the missing rule or fact. Rules/facts that are definitely known to be false are represented by [0, 0] or known to be true by [1, 1] explicitly with multi-valued certainties as in the references [NS90],[BM87].

b. In the context of reasoning with uncertainty, finite failure of all the branches of the search tree for a negated predicate is meaningless. However, combining uncertainties for all lines of reasoning giving the same answer substitution for the negated predicate is necessary, including the unknown facts. After combining the multiple evidence, the certainty for the negated predicate is determined by the rules of the uncertainty calculus. For example, for single valued certainties, a certainty of not p can be obtained after a composite certainty for p acquired, such as prob(not p) = 1 - prob(p). For multi-valued certainty [α, β], then its negation, not p simply has the certainty in the interval [1 - β, 1 - α] in [BM87].

The earlier approaches [Sha83, SS86], do not deal with negations correctly or not at all, since there is no mechanism to incorporate multiple evidence. Later approaches to uncertainty reasoning in [SR89, SF90] do not address negation.

7.3 Layered Interpreter for Uncertainty Reasoning

As shown in [SF90], the inference which is capable of handling multiple lines of reasoning can be represented independent of the uncertainty calculus. A meta-interpreter can be used for this purpose as a meta-meta specification
which can be tailored for the specific application in hand by specifying the
uncertainty calculus. It is important to integrate handling unknown facts and
negation with multiple lines of reasoning. In this section, we present a modified
version of the layered meta-interpreter as an alternative skeleton for providing
a unified framework.

In Chapter 4, we have shown the inadequacy of the extensions to the well
known vanilla meta-interpreter as basis for expert systems shells and also il-
luistrated the utility of the layered skeleton for incorporating negation which
requires handling multiple lines of computation and missing facts. The en-
hancement of the layered architecture for uncertainty reasoning is as follows.
In contrast to the layered interpreter that we used as basis for explanation,
our goal in a computation that involves certainties is to compute a certainty
value considering multiple evidence, hence combining all lines of reasoning.
The function of the bottom layer is to generate a single line of reasoning and
calculate its certainty as well as composition of conjunctions, disjunctions, im-
plications, etc. The top layer monitors the computation as well as computing
the composite certainty from different lines of reasoning. In this extension,
our focus is to combine all lines of reasoning. The computation proceeds in
two layers until all the possible branches of reasoning are exhausted. In con-
tact, the layered interpreter here computes an certainty value. The predicate
determine_result determines the certainty value of each solution.

The layered interpreter is given in Figure 7.1. This interpreter also com-
putes a proof of the deduction. We use the predicate clause_new to obtain each
rule in the assumed format as described in earlier. The filter_result predi-
cate in the top layer handles the certainty calculations for conjunctions, dis-
junctions and negations and initiates the computation of all possible branches.
They are used at the end of the computation along with the computed cer-
tainty values attached to each branch. The computation terminates at the
top level when the first clause of solve_top ultimately fails. At the end of
the computation for each predicate, all the certainty values attached to differ-
ent lines of reasoning are extracted by the combine_prob predicate at the top
layer.
solve_top(U, Goal, Proof) ←
  label(Goal, Name),
  solve_bottom(UBranch, Goal, Proof),
  filter_result(UBranch, Name, U, Proof).

solve_top(U, Goal, cl(Goal, U, ProofSet)) ←
  label(Goal, Name),
  determine_result(Name, U, Goal, ProofSet).

determine_result(Name, U, Goal, ProofSet) ←
  combine_prob(Name, U, Goal, ProofSet).

solve_bottom(T, true, fact) ← truthval(T), !.
solve_bottom(and(CA,CB), (A,B), (PA, PB)) ←
  !, % A and B
  solve_top(CA, A, PA),
  solve_bottom_and(CA, CB, B, PB).
solve_bottom(invert(CA), not A, not(A, PA)) ←
  !, % negation
  solve_top(CA, A, PA).
solve_bottom(U, A, sys(A)) ←
  sys(A), truthval(T), falseval(F), !,
  (A → U = T ; U = F).
solve_bottom(F, A, notclause(A)) ←
  % does not exist in the knowledge base...
  not clause_new(A, Body, Ci), unknownval(F), !.
solve_bottom(body(Ci,CBody), A,
  clause(A,Body,ProofB)) ←
  clause_new(A, Body, Ci),
  solve_top(CBody, Body, ProofB).

solve_bottom_and(C, C, B, PB) ←
  not continue_conj(C), !.
  % stop calculation !...
solve_bottom_and(C, CB, B, PB) ←
  continue_conj(C), !,
  % if the first conjunct has not failed, continue...
  solve_top(CB, B, PB).

Figure 7.1: Layered Meta-Interpreter for Uncertainty Reasoning
We require both of the conjuncts in a conjunction to be computed at the top level as illustrated by `solve_bottom_and` in Figure 7.1. Recall that we made the assumption to provide each solution one by one at the top level, where each answer substitution is given with its composite certainty. This should be taken into account in calculating the certainty of a conjunct, by using `solve_top` for each conjunct in a conjunction.

Unknown facts are represented explicitly at the bottom layer of the interpreter as illustrated in Figure 7.1. The COOP shell [SR89] represents unknowns by adding an auxiliary clause in the knowledge base. This method requires updating the knowledge base and is not necessary as illustrated here by using the appropriate representation within the meta-interpreter.

Posing negated nonground goals as queries, such as `not p(X)?`, is similar to unsound uses of negation in Prolog. By using the meta-interpreter, such uses can be reported and prevented.

The programmable sections of this interpreter are the `filter_failure`, `combine_prob`, `truthval`, `falseval` and `unknownval` which don't affect the nature of the inference. They are used to define the uncertainty reasoning calculus. In addition, a threshold can be used to continue computing a conjunction, by defining `continue_conj/1` appropriately, to indicate whether the second conjunct should be solved. Note that this prevents the search procedure to be terminated prematurely.

The function of `combine_prob` is

- to group the solutions with respect to different answer substitutions. For example, a query `p(X)`, when there are multiple solutions, i.e., `p(1), p(2), ...,` we choose to provide each answer substitution with the composite certainty corresponding to that substitution. The reasons for choosing this approach is obvious since we would like to obtain answer substitutions from our queries by using a Prolog program. However, knowing and grouping the answers and their corresponding proofs in advance is not possible. Therefore, all the possible answers are computed for a predicate and then grouped together by `combine_prob`. 
filter_result(Val, fact, Val, fact) ← !.
filter_result(U, Name, U, sys(G)) ← !.
filter_result(F, Name, _, Proof) ←
    (falseval(F); unknownval(F)),
    store_proof(Name, Proof, F), !, fail.
filter_result(invert(Cf), _, U, _Proof) ← !,
    neg(U, Cf).
filter_result(and(U1, U2), _, U, _Proof) ← !,
    and(U, U1, U2).
filter_result(body(CBody, Ci), Name, _,
    Proof) ←
    imp(U, CBody, Ci),
    store_proof(Name, Proof, U), !, fail.

Figure 7.2: Hooks to the Top Layer for Specification of Belief Calculus

- to eliminate the branches that do not support an answer, i.e. branches
  with 0 or [0,1] certainty from calculations of uncertainty. These branches
  are not eliminated from the proof, with the intent that it can demon-
  strate whether inclusion of additional information, such as facts, rules
  can improve the belief.

- to combine the probabilities for each answer substitution from different
  lines of reasoning. This is provided by a combine predicate.

- to return each solution in order as requested, i.e. p(1), then p(2), ...².

The general structure of filter_result is given in Figure 7.2. We provide
an example calculus for describing uncertainty by using the functions neg, and
and imp which describe negation, conjunction and implication respectively.

²We chose to provide the answers in the same order as Prolog. However, a different
scheme could be adopted easily.
7.4 An Example forAdjusting Reasoning

We give a simple example to illustrate different approaches that can be achieved by using the layered interpreter.

\[ p(X) \leftarrow cf(0.6), \text{not } q(X). \]
\[ q(X) \leftarrow cf(0.7), r(X). \]
\[ q(X) \leftarrow cf(0.8), s(X). \]
\[ r(a), r(b), s(a). \]

Combining two different independent solutions for the goal, \( p \), can be defined in several ways. For example, taking the set union of the answers is defined as the first rule for combination

\[ \text{combine([U1,U2,U3], U) } \leftarrow \]
\[ \text{U is U1 + U2 + U3} \]
\[ - U1\cdot U2 - U1\cdot U3 - U2\cdot U3 + U1\cdot U2\cdot U3. \]

or by using Maier and Warren’s method of using the maximum certainty. [MW88].

\[ \text{combine(List, Max) } \leftarrow \]
\[ \text{list(List),} \]
\[ \text{max_list(List, Max).} \]

We specify the following functions for the bottom layer and values for truth, false and unknowns as shown in Figure 7.3.

Based on these functions and the first rule for combination, the query \( p(a) \) is assigned 0.036 certainty. The certainty associated with \( q(a) \) is 0.94 combining both of the clauses that will give the solution \( q(a) \). The certainty factor for \( \text{not } p(a) \) is computed to be \( 1 - 0.94 = 0.06 \). For the subsequent queries, \( p(b) \) is assigned 0.18, and \( p(c) \) is assigned 0.6. Since there is no clause to prove \( q(c) \), the certainty associated with \( q(c) \) is 0. As a result,
\texttt{truthval(1). falseval(0). unknownval(0).}

\texttt{and(Bel, Bel1, Bel2) \leftarrow}
\texttt{Bel is Bel1 \ast Bel2.}

\texttt{neg(BelNeg, Bel) \leftarrow}
\texttt{BelNeg is 1-Bel.}

\texttt{imp(Bel, BelBody, BelClause) : -}
\texttt{Bel is BelBody*BelClause.}

Figure 7.3: Specification of Belief Calculus

not \ q(c) \ is \ computed \ as \ 1 - 0 = 1, \ which \ propagates \ to \ assign \ the \ maximum
\ certainty \ to \ the \ clause \ p(c).

Uncertainty is also easily described based on D-S theory as in [BM87] by
changing the functions in filtering, the truth intervals and combine accordingly.
The calculus for implementing D-S is given in Figure 7.4. Note that we can
consult the user when there is completely contradictory evidence in favor and
against a particular solution, which is indicated by the values \([0,0]\) and \([1,1]\)
to make the normalization factor 0.

7.5 Final Remarks

The layered meta-interpreter provides several advantages in extending Prolog
for uncertainty reasoning.

Our approach provides flexibility by combining multiple evidence leading to
the same conclusion for uncertainty reasoning. One of the important features
of our design is the ability to separate the inference and uncertainty reasoning
by an appropriate meta-interpreter. In our example, we have incorporated un-
certainty by using a single certainty value attached to each clause and defining
uncertainty calculus by modifying the functions used by filter\_result and
combine\_prob. The same architecture can be tailored to accommodate different
methods and representations for calculating uncertainty, i.e. a multi-valued
certainty representation as in D-S shown in Figure 7.4. In terms of the goals
in approaching flexibility, our approach is similar to [SF90]. Instead of using a
truthval([1,1]).
falseval([0,0]).
unknownval([0,1]).
elimineable([0,1]).

and([Bel,Pl], [Bel1, Pl1], [Bel2, Pl2]) ←
    Bel is Bel1 * Bel2,
    Pl is Pl1 * Pl2.

neg([BelNeg,PlNeg], [Bel, Pl]) ←
    BelNeg is 1-Pl,
    PlNeg is 1-Bel.

imp([Bel,Pl], [BelBody, PlBody], [BelClause, PlClause]) ←
    Bel is BelBody*BelClause,
    Pl is 1- (1-PlClause)*BelClause.

% Combination Axioms for D-S:

combine([BelVal,List], Ds) ←
    combine(BelVal, List, Ds).

combine(BelVal, [ ], BelVal).
combine([Bel1,Pl1],[Bel2,Pl2], [BelS,PlS]) ←
    Norm is 1-Bel2*(1-Pl1)-Bel1*(1-Pl2),
    (Norm = 0 ← inform_user(UNorm);
    UNorm = Norm),
    BelS is (Bel1*Pl2+Bel2*Pl1-Bel1+Bel2)/UNorm,
    PlS is Pl2*Pl1/UNorm.

Figure 7.4: Specification of D-S Calculus
set predicate to obtain all lines of reasoning, we obtain and reason with multiple evidences with the different granularity provided in layers. Unlike [SF90], this feature makes handling nonground queries possible. We used Prolog directly for specifying uncertainty calculus. In [SF90], a simple language which compiles into Prolog is provided for this purpose.

Having a layered approach not only provides a flexible architecture for rule based systems in reasoning with multiple evidence, but also incorporates handling negation and unknowns. These aspects are not present in the previous approaches based on meta-interpreters. The extended interpreter in Figure 7.1 handles negation by considering all lines of reasoning for a negated goal and by using axioms for calculating uncertainty for negation at the top layer. Furthermore, unknown facts are handled by the meta-interpreter without changing the knowledge, unlike [SR89]. A default certainty value is assigned to unknown facts by the fail-safe bottom layer where the unknowns are explicitly handled as one branch of reasoning. Alteration of the knowledge base is not required and the inference is defined independent of the domain.

We also achieve different levels of granularity. On one hand, we prune individual branches of the search tree at the bottom layer. On the other hand we access the branches together at the top layer to obtain a combined value of certainty. The certainty functions are assigned at the bottom level and calculated at the top level to incorporate different representations for uncertainty as well as managing proofs in the computation.

We represent the proofs in order to be able to provide an explanation for illustrating the uncertainty calculations for the rules. This is also an issue for resolving conflicts when multiple evidence is present and contradictory. [BM87] illustrates the use of Dempster's combination. However, when totally contradictory evidence exists, an explanation based on the proof tree might be provided to request user interaction. This feature can be integrated in the combine predicate within the definition of combine.prob by passing the proof tree to the inform.user predicate. In order to keep the axioms separate, this can be handled rewriting the axioms of combine during reading the input.

Another issue in handling uncertainty is the dependencies in evidence.
When the rule structure is not a tree [Pea88], this means that the same evidence can be used more than once in the inference of a particular result. We have assumed the independence of evidence in our work for handling multi-valued uncertainties.

Our approach is faithful to Prolog's computation. It differs from [NS90] for probabilistic logic programming where a new proof theoretic semantics is defined. We take a pragmatic approach by extending the layered meta-compiler and overcame the difficulties in the earlier meta-compiler approaches for integrating uncertainty reasoning in Prolog successfully.
Chapter 8

Conclusions and Directions for Future Work

8.1 Achievements

In Chapter 2, we provide yet another definition of meta-programming and illustrate why meta-programming should be considered as a paradigm for developing knowledge based systems. Our definition and understanding of meta-programming is not limited to providing a logic programming object language as the domain language to develop systems. We discuss how meta-programming utilizes the development of knowledge based systems based on meta-level representation of inference both for developing embedded languages and for developing special purpose shells with Prolog.

Chapter 3 illustrates the current approaches to meta-programming in logic programming, including Prolog's, and ongoing research in developing other logic languages for meta-programming. We discuss the representation issues for meta-programming in detail, both the representation of object language and the representation of inference by a meta-program with respect to each language in amalgamated, bilingual and overlapping cases. We discuss the strengths and weaknesses of other languages for meta-programming and compared them with Prolog's approach.

Chapter 4 discusses the existing meta-interpreters as skeletons and techniques for developing expert systems shells. A meta-interpreter serves as a skeleton for representing the control of an object language which is then enhanced as a special purpose shell. We illustrate that different search strategies as well as different abstractions of the same search strategy can be represented by meta-interpreters. We illustrate that vanilla interpreter does not provide an adequate abstraction of control. We introduce a novel interpreter, the layered meta-interpreter as a skeleton which explicitly reasons with success, failure
and hence negation. The skeleton is capable of reasoning with both single and multiple lines of reasoning. In addition, the skeleton returns the result of computation indicated by a Result variable. The result of the computation can indicate success, finite failure, commitment and can also be numeric values. In this chapter, we also discuss other skeletons, such as the one for OLDT resolution and techniques for extending them.

Chapter 5 illustrates the development of a simple embedded language for ordinal reasoning. We show how a Prolog meta-interpreter can be utilized to integrate this language with Prolog for a hybrid knowledge representation.

Chapter 6 illustrates the development of an explanation shell for a knowledge based system represented in Prolog. We use the version of the layered interpreter where computation results are either yes, no or commit as the basis of the explanation shell. This chapter illustrates the utility of the layered skeleton for providing integrated how, why, whynot and bywhom type of explanations. We show that collecting a partial proof tree is useful. The shell can be used to provide alternate solutions and overcomes the problems with the previous meta-interpreter based approaches.

Chapter 7 demonstrates the utility of the layered meta-interpreter as a basis for a tool to represent and reason with different belief calculi. We use the version of the layered interpreter where computation results are single valued or multiple valued numbers that indicate certainty. Our approach covers and extends all the known approaches based on meta-level interpreters, including reasoning with negation and unknowns.

8.2 Conclusions

We conclude that

- Prolog requires extensions for providing proper naming facilities other than obtained by numbervars. The approach proposed by Nakashima et al. [NTU84] will provide both programming with ground and nonground representations at the meta-level for developing applications.
• A layered interpreter is an appropriate skeleton for representing both single and multiple lines of reasoning within a monolingual language. It provides an appropriate abstraction of Prolog's computation model. It reasons with both success, failure and negation within the same framework and also computes yes, no and commitment as well as numeric values as the result of the computation. This feature makes the layered interpreter a good basis for developing shells for both explanation and uncertainty.

• Stepwise enhancement has simplified the construction of expert system shells based on meta-interpreters. We have devised a collect technique for representing partial proof trees which provided a global explanations for why questions.

### 8.3 Future Work

The link to source level representations of object level programs can be very beneficial for writing debuggers. The problems we faced in representing failed goals without actual unification and having to use re-unifications at the meta-level as we discuss in Chapter 6 might be overcome by adapting a ground representation. This topic requires further research within extending Prolog to have both ground and non-ground representation for meta-programming. Explanation shell can also be improved based on a ground representation which will eliminate undoing unifications by explicit backtracking.

We have not used an explicit planner for generating explanations. The explanation module will benefit from representing different plans and strategies for explanation. For example, level by level explanation is integrated within the shell. An approach similar to [Duc88] where strategies can be represented separately can be beneficial.

For combining multiple evidence, the hardest part is to write the combination axioms. These axioms are known for D-S and Bayesian approaches. For example, the D-S formulation approach is given as a result of solving sets
of constraints for determining the overall support. Representation of probabilistic logic within logic programming [NS90] also requires representation and reasoning with constraints. A shell based on constraint logic programming with meta-programming capabilities such as [SL90] can be utilized to extend our work.
Bibliography


