INFORMATION TO USERS

The most advanced technology has been used to photograph and reproduce this manuscript 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.

UMI

University Microfilms International
A Bell & Howell Information Company
300 North Zeeb Road. Ann Arbor. MI 48106-1346 USA
313-761-4700 800-521-0600
A workbench for developing logic programs by stepwise enhancement

Lakhotia, Arun, Ph.D.
Case Western Reserve University, 1990
A WORKBENCH FOR DEVELOPING LOGIC PROGRAMS BY STEPWISE ENHANCEMENT

by

ARUN LAKHOTIA

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
January 1990
CASE WESTERN RESERVE UNIVERSITY

GRADUATE STUDIES

We hereby approve the thesis of

ARUN LAKHOTIA

candidate for the Ph.D. in Computer Science
degree.*

Signed:  

(Chairman)

Date 6/30/89

*We also certify that written approval has
been obtained for any proprietary material
contained therein.
A WORKBENCH FOR DEVELOPING LOGIC PROGRAMS BY
STEPWISE ENHANCEMENT

Abstract

by

ARUN LAKIOITA

Logic programming languages provide a higher level of data and control abstraction compared to procedural languages. The greater abstraction has made visible some new patterns in programs and in the process of their construction. Programs may be conceived of containing a skeleton program that provides its primary flow of control. A program is developed by adding computation around its skeleton without altering the control flow it provides. This is called enhancement and is manifested by the addition of extra arguments and goals in a program.

Stepwise enhancement takes advantage of this decomposition of a program into skeleton and enhancements. It is a method of programming that suggests the development of programs by first constructing a skeleton program that provides its main flow of control. The final program is constructed by applying a sequence of enhancements to this skeleton. Further, if the successive enhancements are mutually independent the development of program is forked into independent branches. The resulting programs are composed into one at
a later stage.

This incremental method of programming provides a new way for separating concerns during programming. It also identifies the need for additional support from a programming environment to aid in the process of constructing programs. The workbench designed in this thesis is equipped with the support necessary for developing logic programs by stepwise enhancement. It provides mechanical aid to perform enhancements. It also provides tools to compose programs that result from parallel but independent enhancements of the same program. The enhancement and composition tools use program transformation techniques for carrying out the respective operations. This use of program transformations contrasts with their conventional use to improve the efficiency of programs.
To late Shri S. N. Singh,
my high-school Mathematics teacher
for his stimulating teaching

and

to Geeta and Balchand,
my parents
for their love, affection, and countless sacrifices
ACKNOWLEDGMENTS

It has been my privilege to work under the guidance of Leon Sterling. He has been a very positive influence all through my stay at Case. The influence has been far beyond shaping the form and contents of this thesis. I have turned to him for advice on matters beyond research activity. On all occasions I have found his views very pragmatic and encouraging.

I have also been encouraged from Meral Ozsoyoglu’s confidence in my research capabilities. I have admired her high professional and personal standards and hope I have lived up to them. I have also benefited from the wisdom and constructive criticisms of Lee White and Charles Wells. I thank Meral, Lee, and Charles for being on my dissertation committee.

The Composers Group consisting of Ümit Yalçınalp, Martin Marshall, Dimitar Bojantchev, and Arvind Bansal provided a very creative environment for work. I enjoyed sharing the office space with these individuals and had several valuable discussions with them.

Ümit Yalçınalp put the partial evaluator developed in Chapter 5 to the first real test. Her experiment with partially evaluating an explanation system [101] provided the first experiment in using the partial evaluator for specializing a non-trivial and real-life Prolog program. Several issues related to handling of Prolog’s extra-logical behavior became clear from her experiment. In the process she went through the frustrations of a first time user of any software system. I thank Ümit for her efforts and patience.

Aditya Srivastava provided the initial insights into the problems as well as programming tricks related to writing a Prolog tracer. Several issues became clearer to me by looking at the code he had developed for TI-Explorer Prolog.
Dimitar Bojantchev made several improvements over my tracer code. Simple insights such as a "repeat-fail" loop provides the requisite control flow for implementing a "retry" command came out from discussions with Leon and him.

Thanks to the electronic mail facility, I have had the opportunity to exchange ideas with several people around the globe. Frank van Harmelen, Peter Kursawe, Gustaf Neumann, and Dan Sahlin shared their partial evaluation related experiences with me. Martin Feather and Thomas Reps provided several helpful pointers to papers relevant to my work. I would have missed some of these works otherwise.

I am thankful to Djahida Smati for her friendship. She gave me the support to survive the stress of research activity. Amongst others, Tejas Desai, Cris Koutsougeras, Yanmin Liu, Koteshwara Rao, Murali Sitaram, and Dhirendra Verma made life as a graduate student tolerable.

Jawahar Bammi and Amitava Datta made the programming environment in the department into a ‘real’ environment. Like several others, I benefited from their devotion to computing. They introduced us and kept us updated with the most recent releases of every conceivable software around the world. I thank them and others like them who write, maintain, and distribute public domain software such as GNU-Emacs, TeX, \LaTeX, and Fig. These tools, UNIX\(^1\) utilities such as awk and make, and the programming environment on SUN Workstations contributed significantly to my productivity in conducting the research and writing this thesis.

Finally, I thank Marcy Sanford, Rose Chandler, Lynn Austin, and Frederica Flanagan for always being willing to help. I also wish to thank the National

---

\(^1\)UNIX is a trademark of Bell Laboratories.
Science Foundation whose grant 1R187-03911 supported me for the last two years.
3. Skeleton Programs

3.1 Data Structures Traversers

3.2 Syntax Analyzers

3.3 Meta-interpreters

3.4 Clichés of Control Flow

3.5 Algorithms

3.6 Background

4. Types of Enhancements

4.1 Background definitions

4.1.1 The Anatomy of a Clause

4.1.2 Program Transformation Rules

4.2 Program Modulants

4.3 Program Extension

4.4 Program Mutants

4.5 Multiple Enhancements

4.6 Related Works

4.6.1 Program slices

4.6.2 Object-oriented programming

4.6.3 Enhancements as in software engineering

5. Prolog Partial Evaluator

5.1 The Principle of Partial Evaluation

5.2 Example

5.3 Issues in Partial Evaluation
8.4.1 Enhancement to identify ports .................................. 181
8.4.2 Enhancement to compute depth .................................. 185
8.4.3 Enhancement for numbering node ................................. 187
8.4.4 Composing enhanced interpreters ................................. 188
8.5 Predicate Naming Conventions ...................................... 190
8.6 Developing Stepper from Pre-Stepper ............................. 191
  8.6.1 Generic user interface ........................................... 191
  8.6.2 Generic command interface ..................................... 194
  8.6.3 Hooks ............................................................ 194
8.7 Developing Tracer from Stepper ................................... 197
  8.7.1 Enhancements for leap and quasi-skip ......................... 197
  8.7.2 Enhancements for skip, unconditional true ................. 200
  8.7.3 Enhancements for retry and fail ............................... 202

9. Closing Remarks ........................................................... 207
  9.1 Choosing a Programming Method ................................... 208
  9.2 Future Works ......................................................... 210

A. Overview of Appendices ................................................. 212

B. Interpreter for “full” Prolog .......................................... 215

C. ProMiX Partial Evaluation System ................................. 219
  C.1 Kernel ............................................................. 219
     C.1.1 Mix .......................................................... 219
     C.1.2 Partially evaluate meta-predicates ....................... 226
     C.1.3 Partially evaluate set-predicates ....................... 229
C.1.4 Procedures for manipulating residue .................................. 231
C.1.5 Rewrite .............................................................................. 238
C.2 Driver ............................................................................... 240
C.3 Knowledge for controlling unfolding ...................................... 244

D. Technique Interpreters .............................................................. 250
D.1 Accumulator Technique ............................................................. 250
D.2 Propagate Context Up ............................................................... 251
D.3 Propagate Context Down ........................................................... 252

E. Program Composition ............................................................... 254
E.1 Compose Corresponding Clause ............................................... 254
E.2 Compose Procedures ............................................................... 257

F. Tracer: Result of composition .................................................... 259

G. Support Procedures ................................................................. 263

Bibliography ............................................................................... 273

Index of Predicates ..................................................................... 282
List of Figures

2.1 BNF for language PL ............................................. 12
2.2 Syntax analyser for PL in DCG ................................. 13
2.3 Schematic for developing a parser ............................. 16
2.4 Enhancement to create derivation tree ....................... 17
2.5 Enhancement for symbol table construction .................. 18
2.6 A parser for PL .................................................. 19
2.7 Schematic for developing a static-semantic analyzer ........ 25
2.8 Extending the parser to perform semantic analysis .......... 28
2.9 Correspondence between terminology with Feather's method . 31

3.1 Skeleton for list traversal programs ........................ 10
3.2 Vanilla meta-interpreter ....................................... 47
3.3 Euclid's algorithm .............................................. 53
3.4 Extended Euclid's algorithm .................................. 53

4.1 Check graph connectivity ...................................... 58
4.2 Modulated program for checking connectivity ............... 59
4.3 Check connectivity without cycles ........................... 65

5.1 count - a Prolog interpreter to count reductions ........... 75
5.2 should_take - a rudimentary medical expert system .......... 76
5.3 Result of compiling count with respect to should_take ...... 77
5.4 Result of transforming residue .............................. 78
<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>5.5</td>
<td>Mix kernel for pure Prolog</td>
<td>83</td>
</tr>
<tr>
<td>6.1</td>
<td>Programs 'propagating context up'</td>
<td>133</td>
</tr>
<tr>
<td>6.2</td>
<td>Programs using 'accumulators'</td>
<td>133</td>
</tr>
<tr>
<td>6.3</td>
<td>An accumulator technique interpreter schema</td>
<td>138</td>
</tr>
<tr>
<td>6.4</td>
<td>Accumulator specialized for list handling</td>
<td>141</td>
</tr>
<tr>
<td>7.1</td>
<td>Compose bodies of corresponding clauses</td>
<td>157</td>
</tr>
<tr>
<td>8.1</td>
<td>Byrd box model</td>
<td>174</td>
</tr>
<tr>
<td>8.2</td>
<td>Schematic for developing a tracer</td>
<td>183</td>
</tr>
<tr>
<td>8.3</td>
<td>Identify 'ports' of object program</td>
<td>186</td>
</tr>
<tr>
<td>8.4</td>
<td>Compute depth</td>
<td>187</td>
</tr>
<tr>
<td>8.5</td>
<td>Associate unique node number</td>
<td>189</td>
</tr>
<tr>
<td>8.6</td>
<td>Pre-stepper</td>
<td>190</td>
</tr>
<tr>
<td>8.7</td>
<td>Port handling</td>
<td>193</td>
</tr>
<tr>
<td>8.8</td>
<td>Generic command interface</td>
<td>195</td>
</tr>
</tbody>
</table>
Chapter 1

Introduction

1.1 Motivation

Programming has come a long way since the days of vacuum tube computers. From an activity that was performed by a handful of people to tame mammoth computers, it has already proliferated the fabric of our entire social system. If our present dependence on software is alarming then the projections of what is yet to come is beyond comprehension. An article on software engineering education for the future states that “... by early in the next century, every man, woman, and child in the United States will have to be a software engineer” [36]. To prevent this from happening the article stresses the need to improve the productivity of the current and next generation software engineers.

Invention of programming languages with greater capabilities for abstraction has always been an important first step in improving the productivity of programmers. Each level of abstraction provided us deeper insight into programming and triggered the development of newer methods for developing programs. Meanwhile advances in hardware technology made it feasible to use computers for more complex applications. It also made feasible the development of software tools for assisting in the programming process itself. The new programming methods and tools invariably required the enforcement of some programming discipline. This discipline enabled the programmers to channel their mental energy into deeper intellectual issues. The lessons learned led to
the development of languages or language constructs that explicitly supported
the discipline enforced by a programmer.

The development of language, methods, and tools, each feeding on the
other, has helped in “breaking the complexity barrier” [99] over and over
again. This pattern is vividly clear in the development of the programming
languages Pascal and C; the methods of programming such as structured pro-
gramming, stepwise refinement, and module decomposition; and tools such as
structure and visual editors, program generation and maintenance assistants
provided in the UNIX programming environments. There is a similar con-
nection between the development of LISP, AI knowledge representation and
programming techniques, and the MACLISP and INTERLISP programming
environments.

We are now at the dawn of an era ushered in by the logic programming
paradigm. There is a proliferation of activity in designing languages based on
logic. Some examples are Prolog [21], GHC [91], PARLOG [20], λProlog [59],
and CLP(R) [41]. While practical and commercial implementations of Prolog
are already in existence; work is underway in implementing other logic lan-
guages. These languages provide a better abstraction for representing data and
control. Rather than juggling pointers and access functions, data structures
can be created and manipulated symbolically. They also attempt to obviate
the need to specify control in the program. A program, in these languages, is
a declarative statement of the problem rather than a description of operations
required to solve it.

The next step in making the use of a language prolific is the develop-
ment of methods and tools for programming in that language. One important
method of programming that has arisen from the logic programming paradigm is meta-programming [48]. This method suggests separating the logic and control component of a program where logic corresponds to the domain knowledge of a system and control to its inference engine. Meta-programming is found very conducive for expert systems construction [80, 81].

While meta-programming provides a very powerful vehicle for programming, it is not a general purpose programming method. Prolog programming skills are still learned by apprenticeship. This is evident from the approach taken by Prolog textbooks in teaching the use of Prolog in solving problems. The first generation textbooks of Prolog were little more than a collection of examples. The titles “How to Solve It in Prolog” of the book [22] and the title “More Example Programs” of a chapter in the classic primer by Clocksin and Mellish [21] speak for themselves. The second generation of Prolog textbooks added structure, but still largely adopted an apprenticeship approach. As stated in Section 3.3 of [84], "... we claim that the composition of logic programs is a skill learned by apprenticeship or osmosis..."

This thesis is motivated by the desire to take the next step in making the use of logic languages prolific. Experience with programming in Prolog has given us some insight in relationships between program structures and the process of programming. The result of assimilating our insight is a method for logic programming. This thesis presents this method of programming and also some tools that aid in the construction of programs using this method.

1.2 Results

The main results of this thesis may be summarised as follows:
1.2.1 Method of programming

The thesis presents stepwise enhancement, a method for incremental construction of logic programs. The method is based on the observation that a program may be conceived of as an enhancement to a simpler program. An enhancement is a restricted class of modifications commonly applied to logic programs. These modifications typically require addition of extra arguments and goals in a program\(^1\). Since the simpler program is itself a program, the development may proceed by applying a sequence of enhancements to some initial program.

When applying a sequence of enhancements in succession there may be some enhancements that do not depend on each other. Stepwise enhancement takes advantage of this independence of enhancements. Each independent enhancement is carried out on a separate copy of the simple program. This leads to the construction of several programs each implementing some partial behavior of the final desired program. These programs are merged at a later stage as and when required.

It is observed that the decomposition of a program \(P\) into a simpler program \(SP\) and an enhancement is not usually arbitrary. It corresponds to a natural decomposition of the problem being solved. Conventional programming methods are ineffective in utilizing such a decomposition of the problem to reduce the complexity of the programming exercise. They are ineffective because they assume that the natural decomposition of a problem leads to a physical decomposition of the program. In contrast, as \(P\) is derived by modifying \(SP\) the traces of this decomposition captured in \(SP\) are lost in the final program. For such problems stepwise enhancement provides a more natural

\(^{1}\text{In this thesis the word 'enhancement' is used in this restricted sense, as opposed to the broad meaning associated with it in software engineering.}\)
way of developing programs.

1.2.2 Types of enhancements

The thesis identifies three types of primitive enhancements. They are modulation, extension, and mutation.

When developing a program we often need to change only part of a program. To change that part a distinction must be made between it and the rest of the program. This distinction is made by creating a new procedure from that piece of code. Creating a new procedure from existing code or making a procedure inline is called modulation.

Changes that involve addition of extra arguments and goals to the simple program are called extensions. It is typically performed to introduce some extra computation around the control flow provided by an existing program without altering its flow of control.

In contrast if a change alters the underlying flow of control of the original program it is called a mutation. Changes that introduce clauses or introduce goals that introduce a new recursive branch are examples. Though deletion of clauses or goals also change the control flow they are not considered mutation.

Enhancements to a program are carried out by applying a sequence of these primitive enhancements.

1.2.3 Program development tools

An algorithm for composing programs that result from enhancing a common program is presented. The algorithm uses the knowledge that these programs
have the same control flow or skeleton. It composes the two programs by merging the enhancements performed for them around the same skeleton.

The thesis also presents a collection of commonly used enhancements that are partially independent of domain. These are called programming techniques. The application of these enhancements may be decomposed in two steps. The first step involves repeating domain-independent patterns of changes to a program. The component of change that depends on the domain is captured using schemas. The second step involves instantiating the schema variables to predicates performing the specific computation.

1.2.4 The workbench

The workbench provides a collection of tools to support the development of programs using stepwise enhancement. It contains mechanical support to compose programs and also for enhancing programs.

It also provides support to automatically incorporate programming techniques into a program. The domain-independent programming knowledge for a technique is represented as a meta-interpreter. To incorporate a program with a specific technique it may be compiled using that interpreter. The compilation is performed by partially evaluating the interpreter representing a technique with respect to the program.

The partial evaluator used to incorporate techniques can in fact be used as an independent tool for meta-programming. Other program transformation tools may also be built and added to the toolkit.

To make full use of the power of these tools one may integrate them with other environmental support. Some initial ideas towards this end are sketched
1.3 Outline

The thesis assumes a basic familiarity with logic programming and Prolog. The related concepts have been discussed in depth in text books [21, 81] and other literature and hence are not repeated here.

In Chapter 2 the method of stepwise enhancement is presented. Its use is demonstrated by means of developing a parser. Besides easing the development of some types of programs the method also brings other benefit. In particular it makes it possible to reuse common enhancements between programs written for entirely different purposes.

Identifying a skeleton program that provides the primary control flow of the final program is the first step in stepwise enhancement. Chapter 3 gives examples of skeleton programs. These are classified in five categories to help the reader abstract the notion of skeleton programs.

Chapter 4 identifies and classifies three elementary enhancements. They are modulation, extension, and mutation. Programs are developed by applying a sequence of these enhancements.

A partial evaluator for “full” Prolog is developed in Chapter 5. Its development is of interest for two reasons. First, it is developed by enhancing a Prolog meta-interpreter and, second, it forms the basis for the enhancement tool presented in the next chapter.

In contrast to Chapter 4 that classified elementary enhancement, Chapter 6 classifies some enhancements performed as programming techniques. These techniques are conventions used to propagate information across various levels
in execution of a program. A tool to automatically incorporate these techniques into a program is also developed.

The method of stepwise enhancements allows the development of a program to be forked into independent branches. Each branch introducing an independent behavior on top of the skeleton program. An algorithm for composing these programs into one is presented in Chapter 7.

Chapter 8 contains a step-by-step development of a Prolog tracer using the method of stepwise enhancement.
Chapter 2

Stepwise Enhancement

Programming usually requires an untangling of threads of logic for intertwining subproblems. Stepwise enhancement is motivated from the observation that there are situations when these subproblems form separate threads that have to be woven around a common skeleton. They are so strongly coupled with the skeleton that they cannot exist independently.

Stepwise enhancement is a method for developing programs that require several intertwined threads of logic around the same skeleton program. The method suggests developing the program by first building the skeleton and then carefully weaving one thread at a time around it. The weaving of a thread of logic around the control provided by a skeleton program is called an enhancement. An alternate method is proposed for conditions when these threads of logic depend on the skeleton but are mutually independent. Instead of weaving all the threads around the same skeleton the threads may be untangled and woven around separate copies of the skeleton. This results into several enhancements of the same skeleton program. The final program is derived by interweaving these independent enhancements around the same skeleton program. The interweaving is performed by composition.

The difference between stepwise enhancement and other methods may be highlighted from the difference in their underlying philosophy. The methods proposed by [5, 28, 43] are driven by the desire to have a rudimentary system
up and running as soon as possible. This is usually the result of a 'quick and dirty' implementation. These methods emphasize that at all stages during the development one has a system that behaves partially like the final program. In contrast the initial program in stepwise enhancement and all the intermediate programs solve some part of the final problem. However they may not be treated as a prototype of the final program.

This difference in the philosophy between stepwise enhancement and other methods indicate that the two approaches are non-competing. In a practical environment for developing programs a combination of the two approaches may be used to break the complexity of the programming task. This issue and a detailed comparison with related methods are discussed later in Section 2.4.

2.1 A Programming Example

Wirth opens his landmark paper [100] with the statement "programming is usually taught by examples". This still seems to be the best way of describing programming principles and techniques. This section illustrates the use of stepwise enhancement by using it to develop a parser.

The programs are developed using definite clause grammars (DCG). A working knowledge of DCG's and various issues involved in the writing of a parser would be helpful in understanding the examples and appreciating the issues raised. The reader may refer to [1] for knowledge about parsing and to [23, 84] and [66] for discussion on parsing and compiling using DCGs.
2.1.1 A parser

A parser takes a stream of tokens and checks if the incoming sequence can be recognized by the grammar of the language. Besides analysing the syntax it also performs the following tasks:

- create the derivation tree from the input stream. A derivation tree is more convenient for code optimization and generation.
- create a symbol table of all variables declared in the program. The symbol table maintains type and scope information.
- check for consistency of declaration and use of variables
  - Each variable declared must be declared only once
  - Each variable used must be declared

The source language recognized by the parser developed here is PL, a procedural language. Its grammar in BNF notation is given in Figure 2.1. The language is very simple. It consists of declaration, assignment, conditional, and iteration statements. Arithmetic expressions are omitted from the discussion for the sake of brevity.

2.1.2 Successive enhancements

The construction of a derivation tree as well as a symbol table requires parsing the input program. The two structures are created by the semantics actions triggered on reducing a grammar rule. Instead of parsing the program twice to construct the two structures the two tasks may be performed simultaneously.
\[<\text{program}> ::= \text{begin} \ <\text{decl\_list}> \ <\text{stmt\_list}> \ \text{end}\]

\[<\text{decl\_list}> ::= \text{declare} \ <\text{id}>\]
\[<\text{decl\_list}> ::= \text{declare} \ <\text{id}> \ <\text{decl\_list}>\]

\[<\text{stmt\_list}> ::= <\text{stmt}>\]
\[<\text{stmt\_list}> ::= <\text{stmt}> \ <\text{stmt\_list}>\]

\[<\text{stmt}> ::= <\text{id}> ::= <\text{exp}>\]
\[<\text{stmt}> ::= \text{if} <\text{cond}> \ \text{then} \ <\text{stmt\_list}> \ \text{else} \ <\text{stmt\_list}> \ \text{fi}\]
\[<\text{stmt}> ::= \text{while} <\text{cond}> \ \text{do} \ <\text{stmt\_list}> \ \text{od}\]

Figure 2.1: BNF for language PL

The semantic actions corresponding to each grammar rule would therefore consist of a collection of actions corresponding to the two activities.

When developing the parser by stepwise refinement there would be a refinement step corresponding to the processing of each grammar rule. Simultaneous to performing the parsing activity, at each refinement step, one has to make decisions for constructing the two structures. Each step would therefore concern about more than one issue at a time.

It may be noted that semantic actions required for constructing a symbol table do not depend on those required for creating a derivation tree, and versa. This knowledge may be used to break the construction of the parser into the following steps:

- develop a syntax analyzer for the language. Figure 2.2,
- enhance it to construct the symbol table. Figure 2.5,
- enhance it to construct the derivation tree. Figure 2.6.

The parser can therefore be developed by a series of successive enhancements. This is graphically represented by the chain at the left hand side in Figure 2.3.
The first node in the chain represents the skeletal program. Each intermediate node in the chain represents intermediate programs after an enhancement step. Application of enhancements causes a transition in the program development process. This is represented by the arcs. The last node in the chain represents the final program.

When a program performs several activities around the same flow of control it is convenient to develop it by a sequence of enhancements. This sequence can be planned by performing the following exercise:

- identify the flow of control and separate it from the various activities built around it
- create the dependence relation between the activities
- if the dependence relation of a set of activities form a cycle cluster them into one activity
- The dependence relation defines a partial order on the new set of activities. Perform a topological sort of these activities on this relation.

The program may now be developed by a sequence of successive enhancements. The first step in this sequence is to build a program that encodes the flow of control. This provides the skeleton for applying the sequence of enhancements. This sequence is derived from the last step above. Enhancements for an activity that depends on other activities are performed later in the sequence.

This method eases the programming process by allowing one to concentrate on the development of one activity at a time. This is despite the fact that in
the final program code for all the activities coexist. It uses independence of activity to decompose the development process when this independence can not be used to decompose the program.

2.1.3 Parallel enhancement

It was noted earlier that the construction of the symbol table and the derivation tree are mutually independent. In the previous section the parser could also have been developed by swapping the order of the two enhancement activities.

The enhancement steps described above attempt at separating these two activities. At the first glance it may seem that the attempt is successful, in practice it is not completely true. Enhancements require making changes to a program. In the development chain the $N^{th}$ enhancement is performed after $N - 1$ enhancements have been performed. This requires manipulating a program resulting from the collective changes performed by all the enhancements so far. This meddling with the code of another activity implicitly requires one to be concerned about it. Applying successive enhancements buys separation of concerns only from the forthcoming changes.

A better approach to develop the parser would be:

- develop a syntax analyzer for the language, Figure 2.2
- enhance the syntax analyzer to construct the derivation tree, Figure 2.4
- enhance the syntax analyzer to construct a symbol table from the declarations in a program and propagate it to the statement part, Figure 2.5
- compose derivation tree constructor and symbol table constructor to produce a parser, Figure 2.6
program ←
    [begin],
    decl_list,
    stmt_list,
    [end].

decl_list ←
    [declare], id.
decl_list ←
    [declare], id, decl_list.

stmt_list ← stmt.
stmt_list ← stmt, stmt_list.

stmt ← id, [:=], exp.
stmt ←
    [if], cond,
    [then], stmt_list,
    [else], stmt_list,
    [fi].
stmt ←
    [while], cond,
    [do], stmt_list,
    [od].

Figure 2.2: Syntax analyser for PL in DCG

A syntax analyser for a language takes a string of tokens and checks if the string belongs to the language. The program above is a syntax analyzer for the language PL given in Figure 2.1. It is written in definite clause grammar (DCG) syntax. Each DCG rule can be directly translated to a Prolog clause thereby generating a syntax analyser for PL in Prolog. (Similar to the translation of yacc rules to ‘C’ [42]). DCG, its use for parsing and compiling, and translation to Prolog is very well described in [23, 66] and [84].

A syntax analyser provides the skeleton for building most language processing applications. It may be enhanced to build a parser, a static semantic analyser, or even an interpreter.
This development activity may be represented by the directed graph in the middle of Figure 2.3. The graph has one start node and one destination node. The start node represents the program for the skeletal subproblem and the end node the desired final program. The nodes (denoted as \( \infty \)) always have in-degree \( > 1 \). They are nodes where programs are merged. The process of merging programs is called composition; it is discussed in Chapter 7. The in-degree of all other nodes (except the start node) is one. These nodes represent various intermediate enhancements. The arcs, as before, simply denote transitions between stages of program development.

This process of programming has provision for performing multiple yet independent enhancements of the same program in parallel. It opens up the possibility where an enhancement may be performed only on a program that contains details relevant for the purpose of the enhancement. The development process can be planned by altering the steps given for successive enhancements. Instead of performing a topological sort the dependence relationship may be used to create a graph as follows:
program(Tree) ←
  [begin],
  decl_list,
  stmt_list(Tree),
  [end].

decl_list ←
  [declare], id.
decl_list ←
  [declare], id, decl_list.

stmt_list(S) ← stmt(S).
stmt_list((S;S1)) ← stmt(S), stmt_list(S1).

stmt(assign(Id,Exp)) ← id(Id), [:=], exp(Exp).
stmt(if(Cond,Then,Else)) ←
  [if], cond(Cond),
  [then], stmt_list(Then),
  [else], stmt_list(Else),
  [fi].
stmt(while(Cond,Do)) ←
  [while], cond(Cond),
  [do], stmt_list(Do),
  [od].

Figure 2.4: Enhancement to create derivation tree

The DCG program above is enhanced from the syntax analyser for PL given in Figure 2.2. It is enhanced to create the derivation tree corresponding to the derivations performed to reduce the string. Since the derivation tree is eventually used for code generations only the tree of derivation for program statements and not the declaration is created.

The derivation tree is represented as a Prolog term. Each statement is represented with a suitable structure that encodes all the relevant information. The variable Id is bound to the identifier name accepted during the reduction. The variable Exp bound to the derivation tree resulting from reducing an expression.
program(Env) ←
    [begin],
    decl_list(Env),
    stmt_list(Env),
    [end].

decl_list(Env) ←
    [declare], id(Id),
    {init_env(Id, Env}).
decl_list(Env) ←
    [declare], id(Id), decl_list(Env2)
    {add_to_env(Id, Env2, Env}).

stmt_list(Env) ← stmt(Env).
stmt_list(Env) ← stmt(Env), stmt_list(Env).

stmt(Env) ← id(Env), [:=], exp(Env).
stmt(Env) ←
    [if], cond(Env),
    [then], stmt_list(Env),
    [else], stmt_list(Env),
    [fi].
stmt(Env) ←
    [while], cond,
    [do], stmt_list(Env),
    [od].

Figure 2.5: Enhancement for symbol table construction

The DCG program above is enhanced from the syntax analyser for PL given in Figure 2.2. It is enhanced to construct a table of symbols (variables) appearing in a program. This table is constructed when parsing the declaration section. This information is propagated to the statement section. Appropriate support predicates to help in the construction of the symbol table and validation of constraints have also been introduced.

The predicates \texttt{init.env/2} and \texttt{add.to.env/3} help in creating the symbol table as well as validate that:

\begin{itemize}
  \item each variable declared must be declared only once.
  \item the validation of the second constraint, namely
        \begin{itemize}
          \item each variable used must be declared
        \end{itemize}
  \end{itemize}

is performed during the processing of arithmetic and logic expression. It is not shown here.
program(Tree, Env) ←
[begin],
decLlist(Env),
stmtList(Tree, Env),
[end].

decLlist(Env) ←
[declare], id(Id),
{init_env(Id, Env)}.
decLlist(Env) ←
[declare], id(Id), decLlist(Env2)
{add_to_env(Id, Env2, Env)}. 

stmtList(S, Env) ← stmt(S, Env).
stmtList((S;S1), Env) ← stmt(S, Env). stmtList(S1, Env).

stmt(assign(Id,Exp), Env) ← id(Id, Env). [=], exp(Exp, Env).
stmt(if(Cond,Then,Else), Env) ←
[if], cond(Cond, Env),
[then], stmtList(Then, Env),
[else]. stmtList(Else, Env),
[fi].
stmt(while(Cond,Do), Env) ←
[while], cond(Cond, Env),
[do], stmtList(Do, Env),
[od].

Figure 2.6: A parser for PL

This DCG program above is a parser for the language PL given in
Figure 2.1. It checks if the incoming string of tokens belongs to this
language. If so, it constructs Env, a table of its symbols, and Tree a
structure representing the derivations made to accept the string.
This program is created by composing the programs in Figure 2.4
and 2.5. These programs in turn are enhanced from that in Figure 2.2.
The behavior of the whole can be understood by understanding the
behavior of its parts.
• The graph has \( N + 2 \) nodes where \( N \) is the number of activities. Each activity is associated to a unique node. The two extra nodes are called start and end nodes respectively.

• An edge is drawn from node \( A \) to node \( B \) if activity \( A \) depends on activity \( B \).

• An edge is drawn from a node to the start node if its activity does not depend on any other activity.

• An edge is drawn from the end node to an activity if there is no other activity depending on it.

The directed graph representing the programming steps may be developed by simply reversing the direction of the edges in the above graph. In this graph nodes with in-degree \( > 1 \) represent positions where programs are composed. Similarly nodes with out-degree \( > 1 \) represent parallel enhancements.

2.1.4 Almost independent enhancements

If two enhancements are mutually independent they can be performed in parallel. But complete independence of enhancements is too ideal a condition. In practice some enhancements may be "almost" independent. In other words, while these enhancements may be conceived of as introducing independent behavior, they actually interact though at a very limited number of places. One approach for developing such a program may be to temporarily ignore the dependency and proceed with the development as though the enhancements are independent. The inter-dependency between the enhancements may be taken care of after composing the corresponding programs.
Such a situation arises when writing a compiler. The derivation tree returned by the parser is eventually used during code generation. It may be noticed that the tree constructed by the parser of Figure 2.4 retains the symbolic names of variables. But to generate code one needs the memory address of variables, not their symbolic names. Allocation of memory addresses to variables is performed using the symbol table. The symbolic name kept in the derivation tree can be used to access the memory address from the symbol table.

This method of using the actual symbolic name to match a variable with its memory address works well because PL is not a block structured language. In a block structured language the same symbol may be used to represent different variables. The ambiguity is resolved using scope rules. For such a language the matching of a symbol in the derivation tree with that in the symbol table may not be sufficient. The match has to be guided by some context information. This can become rather complex if the two structures are completely independent.

A simple solution is to modify the derivation tree structure. Instead of keeping the symbolic name one may keep a "pointer" into the symbol table to the entry for a variable after resolving the scope. This tying of the derivation tree to the symbol table introduces a dependency between the two activities. The task of constructing a derivation tree is no longer independent of the task of constructing the symbol table.

A little exercise reveals that the dependency between the two tasks is limited to the grammar rules that process arithmetic and logical expression. The remaining rules still constitute a large part of the program. The above solution
proposes that a large part of the enhancements for constructing a derivation tree may still be performed by ignoring any knowledge of the symbol table. The derivation tree so constructed may be tied to the symbol table at a later stage after composing the two programs.

While this may sound a bit cumbersome this process actually has its rewards. For one thing it makes the decision of tying otherwise independent tasks explicit. This ensures that one would recognize these situations and question the design decisions before creating links between different behaviors.

2.2 Method

The method used in the previous section to develop a parser has also been used for developing several Prolog programs. Amongst other programs it has been found helpful in the development of a partial evaluation system, Chapter 5, a Prolog tracer, Chapter 8, and an explanation shell for Prolog [101]. These are large and complex real life programs. Developing these programs by stepwise enhancement tremendously reduces the complexity of the programming exercise.

The steps required in developing a program by stepwise enhancement may be summarized as follows:

- Analyse the problem to be solved.
- Identify the subproblem that may provide the control for the whole program. Call it the skeletal problem.
- Implement a program for the skeletal problem. This is the skeleton program.
This program itself may be developed using a suitable programming method possibly stepwise enhancement itself.

- If the skeletal problem is equivalent to the whole problem then terminate and use some other method for development.

- Identify various independent activities of the problem built around the skeleton such that the set of activities ‘cover’ the whole problem.

- Enhance the skeleton program into a set of independent programs that implement these activities.

- If there are more than one enhancements compose them into one program.

The key to applying stepwise enhancement is in identifying the program that provides the primary flow of control for a problem and decomposing the problem into a set of independent activities. The intent behind abstracting out the central problem, like other divide-and-conquer techniques, is to divide the primary problem to one that is intellectually manageable. What is intellectually manageable depends on individuals and their experiences.

Given a problem there are no formal directives that may guide one in decomposing the problem as skeleton and enhancements. This is also true of methods suggesting the decomposition into modules and procedures or as interpreters and knowledge bases. The natural decomposition of a problem is arrived at from an individual’s experience with programming. Most often it is the experience with solving problems similar to the one in hand that helps in identifying the correct subproblems. In other words, the decomposition of
the problem becomes clearer after repeated implementation of same or similar problems.

The method of abstracting from experience with one problem and applying it to another is also used to teach the use programming methods. Various programming methods are taught by giving examples exhibiting their use in decomposing problems and implementing the corresponding programs. The programmer is expected to abstract the method from these examples and apply them in similar or dissimilar circumstances. Examples of several skeleton programs are given in the next chapter in the same spirit.

2.3 Advantages

2.3.1 Reuse by sharing enhancements

*Reusing skeleton:* Developing programs by enhancement also makes possible a novel way of reusing code. A skeleton program once developed could potentially be used as a starting point for other programs. This in fact happens with language processors. Analysing the syntax of the incoming data string is an important activity for these programs. The same syntax analyzer can be used for this purpose in all the programs that process a particular language. Their specific processing activity may be built by enhancing the syntax analyzer.

This can be demonstrated by looking at the development of a static-sematic analyzer. This is a program that performs simple context dependent semantic validation of a program without actually executing it. A static semantic analyzer for our example language *PL* is developed by Reps [68]. It

---

¹This is not a coincidence. The language *PL* is taken from [68] and the parser example was triggered from his static-semantic analyzer.
looks for the following anomalies in the static program:

- inconsistency in declaration and use of variables,
- use of variables before initialization, and
- violation of assertions

These anomalies can all be detected during the syntax analysis of an input program. The static-semantic analyzer may therefore be built by enhancing the syntax analyzer of Figure 2.2. Further, the processing required to perform the required validations for an anomaly is independent of those required for the others. The syntax analyzer may therefore be enhanced in three different directions, one for each anomaly. The final program may then be developed by composing the three enhancements. These steps are represented in Figure 2.7.

Reusing enhancements: A more interesting avenue for reuse opens up when two or more programs are developed by multiple parallel enhancements of the same skeleton program. If one or more enhancements performed for one program are also required by the other program they may simply be reused. This is the case between the parser and the static-semantic analyzer. The test
for consistency in declaration and use of variables performed by the static-
semantic analyzer is the same as that performed by the parser. These tests are
performed by the enhancement that constructs a symbol table. The symbol
table construction program of the parser can therefore also be used by the
static-semantic analyzer. Such a reuse of code is not possible with conventional
programming techniques.

The actual details about the other two enhancements required for static-
semantic analyzer are not of interest here. If required one may find them in [68].
The analyzer there is developed using attribute grammar. The grammar rules
can trivially be translated to definite clause grammar rules [26].

The method used by Reps in [70] to demonstrate how semantic informa-
tion may be encoded using attribute grammar is also interesting. He first gives
the context-free grammar for the language. This itself is an attribute gram-
mar program. He then enhances this grammar in three independent steps to
demonstrate:

How to -

- propagate symbol table information.
- detect uninitialized variables in a program. and
- encode semantic information for verification of correctness.

That the enhancements are performed independently to demonstrate one
problem at a time, in my view, is not a coincidence. It is done because it makes
the task of communicating the various ideas simpler. This reason is not stated
explicitly in [70]. It can be implied from the ease and flow of discussion. This
implicit use of stepwise enhancement for discussing a program only strengthens
the utility of this method in program development. One can say that if a
method reduces the complexity in the discussion of a program then it would also reduce the complexity of its development.

2.3.2 Ease of extending capabilities

Developing programs by enhancement and composition not only eases the initial development of a program. In certain situations it also eases its later extension. This happens when the extension to be performed can be done by program enhancement.

Say for instance one wants to extend the parser with the capability to perform static-semantic analysis. One approach for doing this is to modify the program in Figure 2.6 to perform all the analyses it does not already perform. Alternatively one may not make modifications for all the new features at the same time, instead they may be added one at a time. This is given by the schema on the left hand side in Figure 2.8. Using conventional techniques one of these two alternatives would be used.

Based on the observation made in the previous subsection one can say that: 1) the computations required to perform the various analyses are mutually independent and 2) they are built around a syntax analyzer and do not depend on the other enhancements performed to derive the parser. In this scenario stepwise enhancement offers yet another alternative. Rather than extending the parser by enhancing it directly the extended capabilities may be developed independently by enhancing the syntax analyzer. These enhancements may then be merged into the parser by composition. The schematic diagram in the middle of Figure 2.8 abstracts this approach.
2.4 Related Works

Most incremental methods of programming can be summarized using the following extract taken from Basili and Turner [5]

“This technique begins with a simple initial implementation of a properly chosen (skeletal) subproject which is followed by the gradual enhancement of successive implementations in order to build full implementation.”

It is suitable for describing the method given by Dromey for “systematic program development” [28] and that given by Feather for “constructing specifications” [30]. It is also suitable for describing program development by stepwise enhancement.
This similarity between abstract descriptions sometimes makes one to jump to the conclusion that stepwise enhancement is identical to these methods. It is however fascinating to note that despite the rather strong similarities in the abstract description the method is actually different. The key to the differences lie in the interpretation of the terms skeleton program and enhancement. The confusion arises because these terms are elicited by examples and are not formally defined. A comparison of the examples is however good enough to bring out the differences in the spirit of the work.

Other than these methods stepwise refinement also bears a resemblance with Jackson Structured Programming method [17]. A comparison with JSP is also performed.

2.4.1 Incremental construction of specifications

Just like programs, specifications too are written incrementally. This incremental method for constructing specifications given by Feather [30] is paraphrased below.

A specification is developed incrementally by starting with the specification of some very simple idealized version of the task. This simple specification is incrementally “elaborated” toward a final specification. Each elaboration step adds some more details about the system, adds new constraints, or relaxes existing ones.

While elaborations, in general, depend on particulars of the task being specified, there are some elaborations whose high-level nature can be expressed somewhat independent of the task. These are called “idiomatic elaborations”. The intended change from these elaborations can
be described fairly concisely. The execution of these elaborations, using conventional editors, may require repetition of individual changes distributed across the specification. The Knowledge Based Specification Assistant (KBSA) provides a collection of high-level editing primitives to perform these idiomatic elaborations as unit operations.

In the course of construction a specification may undergo a sequence of elaborations. An elaboration that depends on changes introduced by another elaboration is performed later in this sequence. However elaborations that are "independent" may either be performed in an arbitrary order or, still better, in "parallel". Incremental construction can be divided into separate paths, one for each independent elaboration; each path pursued in parallel. These parallel elaborations may be "combined" at a later stage.

One may notice that the above description has a ring similar to stepwise enhancement. If a phrase-for-phrase substitution is performed on the above text according to the correspondence given in Figure 2.9 the resulting text almost paraphrases stepwise enhancement. Besides, in a declarative style of programming a logic program is like a specification. It says what the activity is rather than how it is to be performed. This blurs the difference between a specification and a program and therefore between the two methods.

An initial specification in Feather’s method describes an approximation of the actual task or a system with idealized version of the task. For instance, consider the specification for a patient monitoring system that monitors many patient, an example used by [30]. A simple starting specification for such a system could be one that specifies just one patient. Though an approximation
specification $\leftrightarrow$ logic programs
initial specification $\leftrightarrow$ skeleton program
elaboration $\leftrightarrow$ (enhancement + refinement)
idiomatic elaboration $\leftrightarrow$ programming techniques
parallel elaborations $\leftrightarrow$ corresponding programs
combine $\leftrightarrow$ compose
high-level editing $\leftrightarrow$ incorporating techniques

Figure 2.9: Correspondence between terminology with Feather's method

this specification can still be used for situations where there is actually only one patient.

In contrast we suggest writing a program that provides the skeletal control flow of the whole program. This program may not approximate the final program; it only provides the foundation on which the final program may be built. For example, a syntax analyzer provides the skeleton for a parser. It is obvious that for no subset of a language can the syntax analyzer itself be taken as the parser.

The difference between initial specification and skeleton program leads to the difference between elaborations and enhancements. The former are defined in terms of functional changes in a specification whereas the latter are structural changes. A change in structure implicitly changes the functionality. It is therefore not surprising that a large subset of "idiomatic elaborations" identified in [43] are described by their effect on the structure of a specification. Some examples are: exchange parameters of a construct; replace a predicate with some other predicate.
2.4.2 Inductive Stepwise Refinement

I call the incremental method of programming proposed separately by Basili and Turner [5] and Dromey [28] as "inductive stepwise refinement". Their methods are similar to stepwise enhancement in that they propose to develop programs incrementally. The main difference comes in identifying a subproblem for starting the program development and the notion of enhancement. Basili and Turner [5] give the following definition for a skeletal subproblem:

"A skeletal subset is one that contains a good sampling of the key aspects of the problem, that is simple enough to understand and implement easily, and whose implementation would make a usable and useful product available to the user."

"... the skeletal compiler to be initially implemented can be specified by choosing a skeletal language, $L_0$, for $L$. The language $L_0$ may be a slightly modified sublanguage of $L$ with a grammar $G_0$ that is essentially a subgrammar of $G$.”

In essence a skeletal program as defined by Basili and Turner is functionally identical to the final program with respect to some skeletal subset of its inputs.

Dromey [28] captures these ideas more formally using state model semantics of programs:

"The semantics of a program may be conveniently interpreted in terms of mapping between sets of initial states and sets of final states. ... Using the state model, one systematic way to compose a program in a stepwise manner is by a sequence of refinements, each of which enlarges the set of initial states for which the postcondition"
is established until finally a mechanism has been composed that
will establish the postcondition for the given precondition.

A program is therefore developed inductively by incrementally expanding
the set of inputs it is expected to work on. On the surface this is similar to
proving a theorem inductively. But in practice it is not equally elegant. To
enhance a program for a larger set of inputs one has to explicitly modify the
original program. That modification to a program questions the inheritance of
correct behavior is obvious and expected. A more important question is the
viability of design decisions made so far in accommodating the expansion in the
set of initial states.

This can be best explained by an example. Consider for instance the prob-
lem of writing a parser for a rather complex language such as C. Developing
it using inductive stepwise refinement one may start with writing a parser for
a subset of the language. This parser may then be enhanced by expanding the
subset of the language constructs it parses. This approach is used by Basili
and Turner in the example of developing of a compiler used to demonstrate
their method [5].

Experience with writing parsers show that expanding it to process new
language constructs can not be done by simply adding more code. If the parser
is written using yacc [42] one needs to ensure that the grammar does not have
shift-reduce and reduce-reduce conflicts. Addition of new grammar rules may
introduce new conflicts. Resolving these conflicts may require reorganizing the
grammar rules from the previous subset of the language too. This has a ripple
effect on the semantic actions - actions performed after reducing a grammar
rule. A reorganization of grammar rules violates the assumption these actions
are based on. These actions therefore have to be redesigned. This implies that in developing a parser using inductive stepwise refinement some activities may be designed more than once. The cost of developing a program by such an incremental method could therefore be higher than that due to straight development.

In contrast stepwise enhancement recommends that the task of writing a parser may be decomposed into that of first writing its grammar rules and then enhancing these rules with relevant semantic actions. When developing a parser using yacc one may first write the grammar rules for the whole language, validate these rules to be free of conflict, and then introduce the necessary semantic actions.

In practice a combination of stepwise enhancement and inductive stepwise refinement methods may be used to tackle the complexity of the programming task without increasing the cost of program development. The inductive approach may be used in the development of the skeleton itself. It may also be used in carrying out an enhancement incrementally. However, an enhancement step should only be performed after the program it enhances has been implemented in full.

2.4.3 Jackson Structured Programming

Jackson Structured Programming (JSP) is a method of program design developed by Michael Jackson in the early seventies [17]. Its founding principle is that:

The structure of a program should be based on the structure of the underlying problem.
The method advocates writing a program by first building the structure of a problem and finishing with the program. It emphasizes the correctness of a program structure to achieve correct and maintainable programs. JSP consists of the following basic steps [17]:

1. Draw structure diagrams of all the data objects of the program

2. Merge these data structure diagrams into a single structure diagram, the program structure diagram

3. Make a list of executable operations from the programming language used

4. Allocate the operations one-by-one into the program structure

5. Convert the diagrammatic representation into a program

This method is part of the Jackson System Development (JSD) approach which aims to address most of the software lifecycle. It has been found very effective in the data processing domain.

There are several similarities between JSP and stepwise enhancement. The program structure diagram corresponds to the skeleton program. This correspondence is obvious when processing data that can be represented by context-free grammar. Its grammar leads to the program structure diagram and also the skeleton program. In both case this first step captures the primary control flow of the program.

It may be noted that in the running example of this chapter JSP may implicitly or explicitly be used to develop the syntax analyser. Experience shows that the grammar of a language defines the structure of its syntax
analysers. It helps to first write the grammar and use it as a guideline to write a syntax analyser. This is an explicit use of JSP. On the other hand when one uses yacc or DCG this step is implicitly done by these parser generators.

The derivation of program structure from the data structure is based on the assumption that the processing of individual data elements has local effects. This is not always the case. Consider for example the effect of `!` in Prolog. According to syntax `!` enjoys the same status as any other atomic goal. However its semantics is a lot different. The effect of executing an atomic goal can be characterized only in terms of its success, failure, and the variables it binds. Its effect on the execution of other goals is a consequence of these and is local to the clause that contains the goal. In contrast a `!` directly effects the execution of other goals in a clause and the clauses following it. The definite clause syntax of Prolog provides the program structure for meta-interpreters of pure Prolog. This program structure is not adequate to handle cut because of the non-local effect not captured syntactically. It may however be noted that the correct meta-interpreter for "full" Prolog forms the skeleton program for most meta-programs that process "full" Prolog.

There is also a correspondence between allocation of executable operations and enhancement. Enhancement may be interpreted as the process of allocating executable operations around a skeleton program. In contrast to JSP, stepwise enhancement allows several different enhancements to a skeleton program; each enhancement applying a set of logically related executable operations. This leads to several different enhanced programs that may be merged at a later stage.
The other difference is in the object created during the development steps. The first two JSP steps create a data structure diagram which is later used as a guideline for constructing programs. Stepwise enhancement creates a program at each step.
Chapter 3

Skeleton Programs

Identifying the skeleton program is the first step in developing logic programs by stepwise enhancement. The skeleton provides the primary control flow of the program. What may constitute a good skeleton program is not always intuitive. A lot of programming time may be spent in identifying the control flow of a program. However explicitly identifying the skeleton may speed up the later programming task. The following remarks by O'Keefe [62] about his experience with writing a Prolog tokenizer highlights this point:

... I spent several days trying to write a tokenizer and failing. ... But once I got the "finite state" idea, it really did take me a single evening to write the first draft of the tokenizer ...

The idea referred to above is the recognition that a Prolog tokenizer is a finite state machine. Having recognised that, a tokenizer may be written using the following steps:

- Draw the diagram of finite-state machine that accepts a Prolog token.
- Transliterate it to a Prolog program that does the same work.
- Enhance this program to collect the list of characters consumed by the finite state machine in accepting a token.
The finite-state machine and therefore the transliterated program abstracts the flow of control of the tokenizer; this is its skeleton. The complexity of the programming exercise revolves around identifying and implementing this flow of control.

Stepwise enhancement urges a programmer to explicitly identify the skeleton of a program rather than run into it by accident. This is important even if the problem is not glamorous enough to require multiple parallel enhancements. In several programming problems, including the tokenizer, the final program is derived by just a single enhancement of the skeleton program. All the same developing the skeleton is such an important exercise that it is best developed alone – with as little distraction as possible.

Analyzing a problem to understand the essential structure of its program is difficult. This chapter gives examples of several skeleton programs to provide an insight into identifying this structure. The skeleton programs are classified in five families corresponding to the nature of the problem defining the flow of control.

- data structure traversers,
- syntax analyzers,
- meta-interpreters,
- clichés of control flow,
- algorithms.

These families are not mutually exclusive in that some skeleton programs may belong to more than one family. This classification is also not claimed to be complete. There may be skeleton programs that do not belong to any of the families above. This classification is only intended to help abstract the notion
list([],).
list([X|Xs]) :- list(Xs).

Figure 3.1: Skeleton for list traversal programs

of skeleton programs.

3.1 Data Structures Traversers

Since processing of the data structure invariably requires traversing it, programs traversing data structures form an important family of skeleton programs.

The program in Figure 3.1 traverses a list. It forms the skeleton of many programs requiring complete traversal of a list. It may be noted that this program can be derived directly from the definition of a list. In fact, from a declarative view, this Prolog program itself defines a list.

It may generally be said that a Prolog program that defines a data structure also traverses it. For instance, the following Prolog program defines a binary tree.

tree(nil).

tree(t(L,V,R)) :-
    tree(L), tree(R).

It also forms the skeleton for programs operating on trees. The clause and goal selection order of Prolog associates an order with the traversal. The above program performs a left-to-right depth-first traversal of a tree. If a different order is required a different program may be chosen. The following program performs a right-to-left depth-first traversal of a tree.
tree(nil).

\[ \text{tree}(t(L,V,R)) :- \]
\[ \text{tree}(R), \text{tree}(L). \]

However it is as good a definition of a binary tree as the previous program.

The list or tree data structures defined above are specific types of Prolog terms. The skeleton programs for other types of terms may be written in a similar manner. In practice, there may not always be a predefined set of terms associated to a data structure. For instance, programs performing static analysis take other Prolog programs as data. The set of terms constituting the program being operated upon as data can not be predefined. The following program traverses an arbitrary Prolog term and is therefore a suitable skeleton for such tasks.

\[ \text{term}(\text{Var}) :- \text{var}(\text{Var}). \]

\[ \text{term}(\text{Term}) :- \]
\[ \text{functor}(\text{Term}, _\text{Functor}, \text{Arity}). \]
\[ \text{term}_x(\text{Arity}, \text{Term}). \]

\[ \text{term}_x(0, _\text{Term}). \]
\[ \text{term}_x(N\text{thArg}, \text{Term}) :- \]
\[ N\text{thArg} > 0, \]
\[ \text{arg}(N\text{thArg}, \text{Term}, A), \]
\[ \text{term}(A), \]
\[ N1 \text{ is } N\text{thArg} -1, \]
\[ \text{term}_x(N1, \text{Term}). \]
3.2 Syntax Analyzers

The choice of data structures such as lists or trees constrains the internal structures used to represent data. Very often further restrictions may need to be imposed on the contents of these structures. A list, for instance, may be used to represent a sequence of characters (or string) acceptable by some language. In such a case a program that traverses a list would not provide a suitable skeleton for computations that are sensitive to the syntax of the language. Instead a syntax analyzer – a program that accepts only those strings that belong to a given language – would be an appropriate skeleton.

The need for analyzing the syntax of a sequence is very common in computing. It usually arises when processing data acquired from sources external to the program. The data is validated for syntactic correctness and then translated to some internal structures suitable for further processing. This was done in the running example of the previous chapter.

As another example consider the task of processing a non-empty string containing characters from the English alphabet. The following grammar defines the class of such strings:

<alphas> ::= <is_alpha> <alphas>
<alphas> ::= <is_alpha>

<is_alpha> ::= 'A' .. 'Z' | 'a' .. 'z'

Using lists to represent strings, the above grammar may be translated into the following program.

alphas(In, Out) :-
\[
is\_alpha(\text{In}, \text{Mid}), \\
\text{alphas}(\text{Mid}, \text{Out}). \\
\text{alphas}(\text{In}, \text{Out}) :- \\
is\_alpha(\text{In}, \text{Out}). \\
\text{is\_alpha}([\text{C}|\text{Out}], \text{Out}) :- \\
\text{alpha\_range}(\text{C}).
\]

The \text{In} argument of a clause represents the incoming list of characters. On successful execution of a clause some suffix of this list is accepted. The remaining list is bound to the \text{Out} argument and returned. The predicate \text{alpha\_range/1} is defined as follows:

\[
\text{alpha\_range}(\text{C}) :- \text{var}(\text{C}), !, \text{fail}. \\
\text{alpha\_range}(\text{C}) :- \\
\quad \text{C} \geq 65, \% \geq 'A' \\
\quad \text{Z} \leq 90, \% \leq 'Z' \\
\text{alpha\_range}(\text{C}) :- \\
\quad \text{C} \geq 97, \% \geq 'a' \\
\quad \text{Z} \leq 122, \% \leq 'z'
\]

The query \text{alphas(L, [ ])} would terminate successfully if \text{L} is a list of characters that correspond to a string generated by the above grammar; it will fail otherwise. Similar programs may be used as skeletons for all activities that require syntactic validation of strings with respect to a given grammar.

The above syntax analyzer exhibits non-deterministic behavior. A character accepted by the \text{is\_alpha/2} goal of the first clause may be rejected because
its consecutive goal fails. This character may then be accepted again by the
next clause. The program relies on the memory resident property of lists. An
element of a list is not lost after it is accessed once. Besides a program may
move down a list as it traverses it; however its position in the list is restored
on backtracking.

One may choose an alternate representation for strings. Instead of using
lists a string may be represented using an input stream. Streams are data
objects that interface to external devices. An input stream provides an ab-
straction of an input device; it may be connected to devices such as files and
keyboard. Streams have the property that only one character may be read
from it at a time. Once a character is read the stream is positioned at the next
character. A stream can not be reverted to a previous state, hence a character
once read can not be read again.

A different skeleton is required for language processors that process se-
quences of characters from an input stream. The Prolog tokenizer referred to
in the introduction of this chapter is such a language processor. The non-
determinism inherent in the processing is countered by rewriting the grammar
and by allowing the analyzer to look ahead into the stream. The following
grammar generates the same language as the grammar given before.

\[
\begin{align*}
<\text{alpha}> & ::= <\text{is}\_\alpha> <\text{alpha}_x> \\
<\text{alpha}_x> & ::= <\text{is}\_\alpha> <\text{alpha}_x> \\
<\text{alpha}_x> &
\end{align*}
\]

It however performs a deterministic parse using a left-to-right scan of the
stream with one character look ahead. The following program implements the
corresponding syntax analyzer.
alphas(InC, OutC) :-  
is_alpha(InC, MidC), !.
alpha_x(MidC, OutC).
alpha_x(InC, OutC) :-  
is_alpha(InC, MidC), !,
alpha_x(MidC, OutC).
alpha_x(InC, InC).

is_alpha(InC, OutC) :-  
alpha_range(InC),
get0(OutC).

The predicate alpha_range/1 is the same as defined before, and get0/1 reads one character from the standard input stream. The InC variable in all the clauses carries the character looked ahead before invoking the clause. At the end of successful execution of a clause a string starting with this character is assumed to be accepted from the stream. The character following this string is looked ahead and returned as OutC. The query:

?- get0(C), alphas(C, Co).

accepts from the standard input stream a sequence of characters from the English alphabet.

Similar programs may be used as a skeleton for language processors that accept the input from an external device and process it. The processing, as in the case of a tokenizer, may simply involve converting the sequence of characters into some internal representation.
It may be observed that when representing a string by list one need not perform the translation from BNF to Prolog as demonstrated above. One may instead write the syntax analyzer in DCG [66]. The translation of a DCG program to its Prolog equivalent is performed automatically by most systems using a translation scheme similar to the one used above. The syntax analyzer written in DCG then acts as the skeleton program for developing programs requiring syntax analysis.

### 3.3 Meta-interpreters

Meta-programming is a programming technique wherein an algorithm may be decomposed into a logic component and a control component. The logic component specifies the knowledge to be used in solving a problem. The control component determines the inference mechanism by means of which the knowledge may be used. The decomposition of a program into logic and control components (also referred to as object program and meta-interpreter, respectively) provides a nice abstraction mechanism.

This decomposition also provides the programmer with greater flexibility in programming. One may now experiment with different inference mechanisms or their extensions by replacing one meta-interpreter for another. Conversely, the same control mechanism may be used for different applications represented by different object programs.

The abstraction and flexibility provided by meta-programming is conducive for building expert systems. An expert system may be built as "a uniform collection of meta-level programs built around a core of expert knowledge" [81]. The meta-level programs usually implement an execution strategy and per-
solve(true).
solve((G1,G2)) :-
    solve(G1),
    solve(G2).
solve(A) :-
    sys(A), call(A).
solve(H) :-
    unfold_clause(H, B),
    solve(B).

Figure 3.2: Vanilla meta-interpreter

form some additional computation. The execution strategy considers issues such as the order in which literals and rules are selected, the mechanism used for 'matching', creation of choice-points, etc. The additional computation typically retains information from the execution of the object program for later processing. This may be information such as the proof-tree of the object program's execution.

The execution strategy of an interpreter concerns with its flow of control and is therefore its skeleton. Several meta-interpreters of an expert system may have the same execution strategy but differ in the computation they perform. These meta-interpreters would have the same skeleton. It would therefore be to build these interpreters by enhancing the appropriate skeleton interpreter.

For example Prolog interpreters written in Prolog have been studied extensively in the literature. They have been written at several levels of abstraction. The most widely used Prolog meta-interpreters are at the clause reduction level. They make explicit the choice of clause being used to reduce a goal, and the choice of literal to reduce the resolvent. Unification and backtracking are handled implicitly, relying on the behavior of Prolog.
The program in Figure 3.2 is the most commonly used form of a Prolog meta-interpreter. We call this meta-interpreter *vanilla*. The vanilla interpreter interprets a subset of Prolog. It implements pure Prolog plus system predicates. The system predicates may be for arithmetic predicates, `functor/3` and `arg/3`, and the meta-logical predicate `clause/2`. The four clauses in vanilla are intended to be mutually exclusive. Thus `true`, conjunction, and user-defined predicates are not system predicates. A table of `sys/1` facts may specify the system predicates.

Most Prolog meta-interpreters that do not alter the clause and goal selection order of Prolog are enhancements of the vanilla meta-interpreter. One may write interpreters at still lower levels of abstraction by explicitly implementing unification, backtracking and clause selection. The essential difference between levels is the amount of computational detail made accessible.

Cut is not part of the subset interpreted by the above interpreter. Cut and constructs depending on cut can be handled by other interpreters. For example, O'Keefe gives an interpreter for handling if-then-else in [61]. The Appendix B contains a Prolog meta-interpreter for “full” Prolog. This meta-interpreter provides a useful skeleton for building programs such as the partial evaluator (see Chapter 5) and the Prolog tracer (see Chapter 8). A layered meta-interpreter that forms the skeleton for systems that reason about Prolog's computation is given in [85]. This meta-interpreter has been used to build a system that gives explanation for success as well as failures of “full” Prolog.

Use of meta-interpreters for interpreters in other circumstances may be found in the text book [84]. For instance, it provides an interpreter for a non-deterministic finite-state automaton and various enhancements of the vanilla
meta-interpreter. Pereira and Shieber [66] develop an interpreter of definite clause grammars (DCG) for use in natural language analysis. This interpreter provides a skeleton for enhancements to the DCG computation model.

3.4 Clichés of Control Flow

Every programming language fosters its own characteristic ways for structuring the flow of control in programs. These ways are centered around the model of computation and the abstraction of control provided by the language. Prolog’s non-deterministic model of computation has also led to the development of program structures that express different patterns of control. Program structures such as failure-driven loop, generate and test, or if-then-else have become clichés in a Prolog programmer’s vocabulary. Some of these program structures such as if-then-else have even been included as primitives of the language.

These structures of control flow provide skeletons that harness the non-determinism in Prolog’s computation. The following is an example of such a program structure:

```
choice_point.
choice_point :- choice_point.
```

This predicate creates an indefinite number of choice-points, i.e. it always succeeds on backtracking. A goal `choice_point` when used in conjunction with another goal like `(choice_point, p(X))` causes the predicate `p(X)` to be reinvoked after failures. In “pure” Prolog this structure has little value as 1) proving `p(X)` once is the same as proving it multiple times, and 2) the computation never terminates. However, when used with the ‘!’ side-effecting, and extra-logical
predicates it provides the skeleton for failure-driven loops. These loops are
driven by the failure of a predicate and are usually terminated by a '!' .

Another use of the choice_point/0 skeleton is demonstrated in Section 8.7.3.
There it is enhanced into the predicate check_to_stop_retrying/1 to support the
retry command of a Prolog tracer. This command causes a goal previously
executed to be reinvoked. The ability of choice_point to create a choice point
is used to reinvoke the selected goal, see Section 8.7.3 for details.

Generate-and-test is another common program structure in nondeterminisitic programming. Such program structures typically have a conjunction of
two goals. The first goal generates a set of candidate solutions to a problem
one at a time. The second goal tests the candidates for being acceptable as
a solution for the problem. The following Prolog program gives a skeleton for
generate-and-test programs.

\[
\text{find}(X) \leftarrow \text{generate}(X), \text{test}(X).
\]

For a query such as \( \text{?- find}(X) \) the predicate generate(X) will return some \( X \)
which is tested by the predicate test(X). If the test is successful the value of
\( X \) is returned as a solution. On the contrary, if the test fails the execution
backtracks to generate(X) which then generates another candidate for testing.
This continues until either the tester succeeds or the generator is exhausted of
alternative candidates.

The use of '!' to get the effect of if-then-else is another commonly used
program structure. The program

\[
p(X, Y) \leftarrow \text{if}(X), !, \text{then}(X, Y).
\]
\[
p(X, Y) \leftarrow \text{else}(X, Y).
\]
demonstrates its use. The predicate if$(X)$ tests whether $X$ satisfies some property. If it does then the selection of the next clause is inhibited by a `!'. Hence only the action due to then$(X, Y)$ is performed. The goal else$(X, Y)$ is executed only when the if test fails.

It may not be possible to list the entire collection of such program structures or clichés. Some of the other popular programming clichés representing different flow of control in Prolog are: always-succeed, catchall-fail, catchall-true, cut-fail, find-all-solutions, negation as failure, succeed-at-most-once, and test-existence.

### 3.5 Algorithms

Algorithms provide skeletons that are more strongly related to the problem domain and not as much to its representation. Thus algorithms for sorting, graph traversals, numerical computations, pattern-matching, etc. may be treated as skeletons.

In the simplest case the enhancement specialize the skeleton for a selected data representation. The algorithm to compute the transitive closure of a relation is one such example. The following program computes the transitive closure of a relation $r(X, Y)$.

\[
\text{transitive\_closure}(X, Y) :- r(X, Y).
\]

\[
\text{transitive\_closure}(X, Y) :-
\]

\[
\text{r}(X, Z),
\]

\[
\text{transitive\_closure}(Z, Y).
\]

In Chapter 4 this skeleton is enhanced to the program in Figure 4.1. The
enhanced program tests the connectivity in a graph as transitive closure of edges between pairs of nodes.

An algorithm may also provide a skeleton to piggyback additional computations for a more complex algorithm. This relationship may be seen between Euclid's algorithm and Extended Euclid's algorithm given on pages 2 and 14 respectively of [46]. Euclid's algorithm computes the greatest common divisor of two numbers. The program gcd/3 of Figure 3.3 encodes this algorithm. Given two positive integers \( M \) and \( N \) the goal \( gcd(M, N, GCD) \) returns \( GCD \) the largest positive integer that divides both \( M \) and \( N \).

The Extended Euclid's algorithm computes the greatest common divisor \( GCD \) of two positive integers \( M \) and \( N \). Besides it also computes two integers \( A \) and \( B \) such that \( A \cdot M + B \cdot N = GCD \). The program in Figure 3.4 is a Prolog translation of the corresponding algorithm given Knuth in [46]. That this program is an enhancement of the one in Figure 3.3 can be seen by inspecting the two programs. Not surprisingly, the same relationship may be observed between the algorithms as stated in [46].

### 3.6 Background

The programs, programming concepts, clichés, and algorithms presented in this chapter have appeared in the literature in one form or the other. This chapter looks at these programs as skeletons on which other complex programs may be built. The reader is advised to refer to some textbook on Prolog for understanding the programs and data structures. Most of these programs are also given in the textbook *The Art of Prolog* [81].

Further details on writing parsers using Prolog may be found in [23] and [66].
\text{gcd}(M, N, \text{GCD}) :-
\quad \text{Rem} \text{ is } M \mod N,
\quad \text{gcd}(\text{Rem}, M, N, \text{GCD}).
\text{gcd}(0, M, N, N).
\text{gcd}(\text{Rem}, M, N, \text{GCD}) :-
\quad \text{Rem} > 0,
\quad \text{gcd}(N, \text{Rem}, \text{Gcd}).

\textbf{Figure 3.3: Euclid's algorithm}

\text{gcd}(M, N, \text{GCD, A, B}) :-
\quad \text{gcd\_ex}(M, N, \text{GCD, 0, 1, A, 1, 0, B}).

\text{gcd\_ex}(M, N, \text{GCD, A, AP, FinalA, B, BP, FinalB}) :-
\quad \text{Rem} \text{ is } M \mod N,
\quad \text{Quot} \text{ is } M \div N,
\quad \text{gcd\_ex}(\text{Rem, Quot, M, N, GCD, A, AP, FinalA, B, BP, FinalB}).
\text{gcd\_ex}(0, \text{Quot, M, N, N, A, AP, A, B, BP, B}).
\text{gcd\_ex}(\text{Rem, Quot, M, N, GCD, A, AP, FinalA, B, BP, FinalB}) :-
\quad \text{Rem} > 0,
\quad \text{NewA is } \text{AP - Quot } \times \text{A},
\quad \text{NewB is } \text{BP - Quot } \times \text{B},
\quad \text{gcd\_ex}(N, \text{Rem, GCD, NewA, A, FinalA, NewB, B, FinalB}).

\textbf{Figure 3.4: Extended Euclid's algorithm}
The tutorial [62] gives a good account about writing a tokenizer. These documents do not develop the parsers or tokenizers as enhancements of skeleton programs that simply performs syntax analysis of its input. It would be interesting to rewrite their programs using stepwise enhancement.

Brna et. al. [14] have summarized a set of 14 programming clichés collected from textbooks and by interviewing Prolog experts. The clichés cited at the end of Section 3.4 are a subset taken from this paper. This subset collects only those clichés that relate to the control flow of a program. Brna et. al. called their set of clichés 'programming techniques'. This thesis redefines this phrase to refer to clichés for conventions of propagating information across a program, see Chapter 6.

The Euclid's algorithm and Extended Euclid's algorithm are taken from Knuth's classic [46].
Chapter 4

Types of Enhancements

A program, when developed systematically using stepwise enhancement, goes through a sequence of enhancement steps. At each step some computation is added around the skeleton provided by the program from the previous step. Enhancements only add computation around the flow of control provided by a skeleton program. They do not alter the control itself, at least not explicitly. The alteration, if any, may be implied due to the added computation.

This chapter classifies a set of three elementary enhancements: *modulation*, *extension*, and *mutation*. These enhancements monotonically add computation to a program. Since syntactic alterations to the computations of a skeleton explicitly alter its flow of control they are not considered enhancements. Complex enhancements can be represented as a sequence of these elementary enhancements.

The classification is performed to explore the possibility of providing mechanical support for systematic and incremental development of programs by stepwise enhancement. The program of Figure 4.1 is used as a running example to demonstrate the various enhancements.
4.1 Background definitions

4.1.1 The Anatomy of a Clause

This chapter, and in fact the thesis, concerns with manipulating Prolog programs. A Prolog program consists of a set of clauses. A clause forms the smallest unit for most manipulation operations. It has the following syntax:

\[ h :\ldots b_n. \]

This is essentially a Horn Clause, i.e. it contains at most one positive literal and is read as “\( h \) if \( b_1 \) and \( b_2 \) and \ldots and \( b_n \)”. A clause has the following parts defined with respect to the above clause:

**Head**: The literal ‘\( h \)’ is called the head. This is the only positive literal of the Horn Clause.

**Neck**: The structure “\( :\ldots \)” is its neck.

**Body**: The sequence of literals “\( b_1, b_2, \ldots, b_n \)” is its body.

A Horn Clause consisting of only a positive literal is called a **fact**. Its clause does not have a neck or a body. It may however be translated to a functionally equivalent clause by adding a neck and a body consisting of the literal **true**. In this thesis, unless otherwise specified, all facts are assumed to have been translated in this way.

The above description only defines terms not defined in Prolog or logic programming text. A reader is referred [52] for more details on the syntax and semantics of Prolog.
4.1.2 Program Transformation Rules

The following transformations are used in discussion of enhancements.

Unfold:

The clause \( C \) whose body contains the goal \( G \) which is unifiable with the heads of clause \( C'_1, \ldots, C'_n \), is replaced by the set of clauses \( \{C_i\} \), where \( C_i \) is a resolvent of \( C \) with \( C'_i \).

Folding:

An instance of the body of clause \( C' \) within the body of \( C \) is replaced by the head of \( C' \), giving a new clause \( C'' \) which replaces \( C \) in the program.

Abstraction:

A sequence of goals in the body of a clause of a given program are replaced by a new predicate and a new clause consisting of the new predicate at its head and the goals at its body are added to the program.

Alternatively a group of clauses of a procedure may be extracted to create a new procedure by renaming the predicate at its head. In place of these clause a link clause may be inserted with a call to this new predicate in it body.

The unfold and fold transformations are defined in Tamaki and Sato [87]. The abstraction transformation is taken from Scherlis [75] and adapted for logic programming. These transformations preserve the denotational and operational meaning of the programs.
connected(Start, End) :- edge(Start, End).
connected(Start, End) :- edge(Start, Hop), connected(Hop, End).
edge(cleveland, ny). edge(ny, houston). edge(sf, la).
edge(cleveland, la). edge(ny, sf). edge(sf, cleveland).

Figure 4.1: Check graph connectivity

A directed graph can be captured by a logic program as a collection of facts representing its edges. A fact edge(Node$_j$, Node$_k$) is present in the program if there is an edge from Node$_j$ to Node$_k$ in the graph. The relation connected(Node$_k$, Node$_i$) is true if there is a connection from Node$_k$ to Node$_i$.

4.2 Program Modulants

When developing a program we often need to change only part of a program. To change that part a distinction must be made between it and the rest of the program. This distinction is made by creating a new procedure from that piece of code. Creating a new procedure from existing code or making a procedure inline is called modulation.

**Definition 1 Modulation** ($\mathcal{P} \simeq \mathcal{M}$)

Program $\mathcal{M}$ is a modulant of program $\mathcal{P}$, and vice-versa, if they can be derived from one another by a sequence of unfold and abstraction transformations.

These transformations preserve meaning hence $\mathcal{M}$ and $\mathcal{P}$ have the same denotational and operational meaning.

As an example, the two clauses of Figure 4.1 may be abstracted to form a new procedure:

connected(Start, End) :- connected_x(Start, End).
connected(Start, End) :- connected.x(Start, End).
connected.x(Start, End) :- edge(Start, End).
connected.x(Start, End) :- edge(Start, Hop), connected.x(Hop, End).

Figure 4.2: Modulated program for checking connectivity

connected.x(Start, End) :- edge(Start, End).
connected.x(Start, End) :- edge(Start, Hop), connected(Hop, End).

If the goal connected.x(Start, End) in the first clause is unfolded we get back to the initial program of Figure 4.1. These two programs are modulants of each other.

A program may be modulated for several reasons. A new procedure may be created to identify some piece of code as a logically independent unit. Modulation may also be performed to identify some important logical state in the program. In the program above, the neck of connected/2 corresponds to a state reached in the program just before making any hop across the graph.

Modulations are performed as a precursor to making enhancements where the logically independent task or state is of special significance. For instance, the above program may be modulated further by unfolding the goal connected.x(Hop, End) in the second clause of the new procedure to get the program in Figure 4.2. The position just before (after) the goal connected.x/2 is reached immediately before (after) checking if a given pair of nodes are connected. Making an independent procedure helps distinguishing these conditions from the rest of the program. Having made the distinction one may enhance (see next section) the modulant version with new behavior. This particular modulant is useful for enhancements that call for some processing before
or after checking the connectivity between nodes.

4.3 Program Extension

A program $E$ is an extension of a program $P$ if $E$ is created by introducing computation around the control flow provided by $P$. Typically $E$ will 'do something extra' such as compute a value for an argument. An extension relationship is denoted using the expression $P \triangleright E$ where procedure $E$ is extended from procedure $P$.

Let $E$ be a procedure defining predicate $e$ with arity $m$ and $P$ be a procedure defining predicate $p$ with arity $n$ then

**Definition 2 Extension** $(P \triangleright E)$

$E$ is said to be an extension of $P$, or $P \triangleright E$, if $m \geq n$ and $E$ can be generated from $P$ by applying the following modifications on clauses of $P$. The modifications should be applied in the sequence stated.

- substitute all occurrences of predicate name $p$ with $e$.

- add $m - n$ arguments to all the $e/n$ structure appearing as the head or a goal in the clauses obtained above.

- reorder the arguments of all heads and goals with structure $e/n$ in the transformed clauses from above. The same permutation should be applied for all the reorderings.

- add some extra goals to the body of the clauses from above. The goals added should not introduce recursion involving a $e/n$ predicate.
The procedure $\mathcal{P}$ is called the \textit{parent} procedure and $\mathcal{E}$ its \textit{child}. The following program is an extension of the program in Figure 4.1.

$$
\text{path(Start, End, [Start, End])} :\text{- edge(Start, End).}
$$

$$
\text{path(Start, End, [Start|Rest])} :\text{- edge(Start, Hop), path(Hop, End, Rest).}
$$

The extension performed constructs the list of nodes on the path connecting a pair of nodes.

\textit{Symbolic Variants:} If $\mathcal{P}$ can be enhanced to $\mathcal{Q}$ and $\mathcal{Q}$ can be enhanced to $\mathcal{P}$, then $\mathcal{P}$ and $\mathcal{Q}$ are symbolic variants, or $\mathcal{P} \equiv \mathcal{Q}$.

\textit{Correspondence of program components:} If $\mathcal{P} \sqsupseteq \mathcal{E}$ then for every clause in $\mathcal{P}$, the parent program, there corresponds a clause in $\mathcal{E}$, the child program. We use the terms \textit{parent clause} and \textit{derived clause} appropriately. The derived clause is also referred to as an \textit{inherited clause}.

For every goal in the body of a parent clause there corresponds a goal in the body of the corresponding derived clause. The goals in the derived clause that correspond to goals in the parent clause are called \textit{inherited goals}. The other goals are called \textit{non-inherited} goals. An inherited goal may undergo modification during extension. The changes would entail renaming of predicates, addition of new arguments, and reordering of arguments. The mapping between the predicate name and arguments of inherited goal with the goal it corresponds to in the parent clause may be captured by using extension relationships.

The extension relationship for the above program is:

$$
\text{connected(Start, End)} \sqsupseteq \text{path(Start, End, Path).}
$$
This relation specifies that \( \text{path/3} \) is enhanced from \( \text{connected/2} \). The argument correspondence is represented by using the same variable at the corresponding positions.

**Sibling procedures:** If \( \mathcal{E} \) and \( \mathcal{F} \) are children of the same procedure \( \mathcal{P} \) then \( \mathcal{E} \) and \( \mathcal{F} \) are siblings.

The following procedure \( \text{safe(Start, End, Unsafe)} \) is an extension of the procedure \( \text{connected(Start, End)} \).

\[
\begin{align*}
\text{safe}(&\text{Start, End, Unsafe}) \leftarrow \text{edge}(&\text{Start, End}). \\
\text{safe}(&\text{Start, End, Unsafe}) \leftarrow \\
&\text{edge}(&\text{Start, Hop}), \text{member}(\text{Hop, Unsafe}), \text{safe}(\text{Hop, End, Unsafe}).
\end{align*}
\]

with the extension relationship:

\[
\text{connected(Start, End)} \supseteq \text{safe(Start, End, Unsafe)}.
\]

The procedure \( \text{safe/3} \) and \( \text{path/3} \) are siblings because they are both enhanced from \( \text{connected/3} \).

The clauses of two sibling procedures that are enhanced from the same clause of the parent procedure are called corresponding clauses or sibling clauses.

Goals of two corresponding clauses that are inherited from the same goal of the parent clause are referred to as the corresponding goals. Arguments of corresponding goals that are inherited from the same argument of their common parent are called corresponding arguments.

Sibling procedures are similar in structure. i.e.

- they have the same number of clauses.
- the clauses have one-one correspondence due to extension from the same clause, and

- the corresponding clauses have similar recursive structure.

The knowledge of structural similarity of programs is exploited to compose programs in Chapter 7.

### 4.4 Program Mutants

Addition of computation to a program that alters the underlying flow of control of the original program is called a mutation.

**Definition 3** Mutation \((P \rightarrow M)\)

A procedure \(M\) is a mutant of procedure \(P\) if \(M\) is derived from \(P\) by

- adding one or more new clauses, and/or

- adding goals that introduce a new recursive branch in the procedure.

The following program is a mutant of the program \(path/3\) given in the previous section.

\[
\begin{align*}
\text{path2(Start, End, [Start,End])} & \leftarrow \text{edge(Start, End)}. \\
\text{path2(Start, End, [Start|Rest])} & \leftarrow \text{edge(Start, Hop)}, \text{path2(Hop, End, Rest)}. \\
\text{path2(Start, End, no_path)} & \leftarrow .
\end{align*}
\]

This program is a fail-safe version of the path-finding program. Instead of failing when a path is not found, it returns a \(no\_path\).

When committing logic programs to the Prolog execution model

- reordering of clauses
• reordering of goals
also affect the control flow of the program. These modifications too may be considered as creating mutants. The effect of such a change can be seen from the following examples.

The following program is created by swapping the order of clauses of the program in Figure 4.1. The change in the flow of control is visible from how this program reacts to a query connected(Y, cleveland). It does not terminate whether or not there is an edge in the graph converging into ‘cleveland’.

connected(Start, End) :- edge(Start, Hop), connected(Hop, End).
connected(Start, End) :- edge(Start, End).

Similarly changing the order of goals in the 2\textsuperscript{nd} clause of the program in Figure 4.1 gives the program.

connected(Start, End) :- edge(Start, End).
connected(Start, End) :- connected(Hop, End), edge(Start, Hop).

This reverses the order in which the graph is traversed. Given a query connected(X, Y) the initial program checks for connectivity by traversing the graph from node \(X\) to node \(Y\). The modified program traverses it from node \(Y\) to node \(X\).

The flow of control of a program may also be altered by changes other than those stated above. The deletion of a goal or a clause of a program alters its control. For example: note the difference in the control of the following program in comparison to the program connected/2 of Figure 4.1. It is created by deleting the goal edge(Start, Hop) from the latter program.
simply_connected(Start, End) :-
    simply_connected(Start, End, [Start]).

simply_connected(Start, End, _Visited) :- edge(Start, End).
simply_connected(Start, End, Visited) :-
    edge(Start, Hop),
    not member(Hop, Visited),
    simply_connected(Hop, End, [Hop|Visited]).

Figure 4.3: Check connectivity without cycles

connected(Start, End) :- edge(Start, End).
connected(Start, End) :- connected(Hop, End).

Similarly any modification to the existing goals or the atom at the head of a clause also changes a program's flow. These modifications in a sense correct bugs by rectifying the errors of previously made decisions. Such modifications do not fall in the purview of enhancements and are not considered mutations.

4.5 Multiple Enhancements

Modulation, extension, and mutation are elementary enhancements. An enhancement to a program, in practice, may be broken into a sequence of elementary enhancements.

For instance, the program in Figure 4.3 is created by a sequence of modulations and extensions of the program in Figure 4.1. Its development may be traced as follows:

Figure 4.1 $\simeq$ Figure 4.2 $\supset$ Figure 4.3
Similarly the path2/3 program given in the previous section can be created by the following sequence of enhancements:

\[ \text{connected/2} \supset \text{path/2} \rightarrow \text{path2/3} \]

where \text{connected/2} is the program from Figure 4.1 and \text{path/2} the example program used in one of the previous subsections to demonstrate extensions. An alternate sequence of enhancements to create \text{path2/3} from \text{connected/2} could be:

\[ \text{connected/2} \rightarrow \text{connected/2} \supset \text{path2/3} \]

Here the first \text{connected/2} refers to the program in Figure 4.1. The second \text{connected/2} is the result of adding an extra clause to give following program.

\[
\begin{align*}
\text{connected(Start, End) :- edge(Start, End).} \\
\text{connected(Start, End) :- edge(Start, Hop), connected(Hop, End).} \\
\text{connected(_,Start, _End).}
\end{align*}
\]

This program is then enhanced to the program \text{path2/3}.

### 4.6 Related Works

#### 4.6.1 Program slices

A program slice as defined by Weiser is a program that can be “constructed from an original program by deleting statements” such that it “exactly reproduces a projection of the original program’s behavior” [97]. It may be noted that if procedure \( E \) is an extension of procedure \( P \) then \( P \) can be created from \( E \) by deleting some goals and arguments besides reordering remaining
arguments and renaming the principal predicates. Ignoring the reordering of
arguments and renaming of predicates, the 'trace' of procedure $P$ would be a
projection of the trace of $E$. Procedure $P$ is therefore a slice of procedure $E$.

The converse may not always hold, that is if $P$ is a slice of $E$ it is not
necessary that $E$ is an extension of $P$. This is because slicing a program may
result in the loss of a clause or a recursive goal. The program and its slice may
therefore not have the same control flow.

Weiser reports that experienced programmers use program slices when de-
bugging an unfamiliar program. They slice it to comprehend the interaction
of scattered pieces of logically related code [98]. Our observation may be in-
terpreted as saying that program slices are useful in the actual development of
a program. Multiple enhancements represent multiple slices of the same pro-
gram. The process of composition discussed in Chapter 7 merges these slices
into one program.

### 4.6.2 Object-oriented programming

The extension operation creates an inheritance relationship between programs.
If program $E$ is extended from $P$ then $E$ inherits the behaviors of $P$. For
example a parser developed by extending a syntax analyzer inherits from latter
the property of analyzing syntax. In other words one may say that a parser is a
syntax analyzer. Similarly a Prolog tracer is a Prolog interpreter, see Chapter 8.

This inheritance of relationship is analogous to inheritance of properties
between objects and classes in object-oriented programming paradigm. The
analogy may be illustrated by means of an example. Let us say that $X$ is an
instance of the class elephants. Let the class elephants be defined in terms of
the class mammals. In the object-oriented terminology one says that \( X \) is an \textit{elephant} and that \( X \) is a \textit{mammal}. The object \( X \) inherits properties from the class elephants and also from the class mammals.

The analogy above does not imply that the two concepts are the same or even similar. An object in object-oriented programming is a dynamic entity created during execution of a program. It is created by instantiating a class definition. A class is a static element defined by some piece of code. It defines a new data type, in other words a family of objects. On the contrary, we define inheritance relationship between programs itself. As opposed to instantiation, extension is an operation performed in the process of programming not during execution. It is performed on programs and results in programs. The operation is usually carried out by a programmer and not the machine.

One may wish to study the implication of meta-programming and automatic programming systems on the above analogy. Automatic programming systems aim at generating programs automatically. This blurs the difference between a machine and a programmer, at least as far as programming goes. The distinction between data and program is blurred by meta-programming and so is the notion of static and dynamic elements. This leads to a blurring of the distinction outlined in the above discussion. The only difference that still stands is between the operation used to create an object or a program. In one case it is instantiation and another it is extension. But automatic programming systems are still far from reality and hence this is may be treated just as an intellectual exercise.

One may refer [11, 24, 37] or other literature for details on object-oriented programming.
4.6.3 Enhancements as in software engineering

The word 'enhancement' is used in the software engineering paradigm to denote modifications to a 'frozen' system to accommodate features that were initially overlooked. The oversight may be due to incomplete specification, improper understanding of the specification, or simply lack of insight and experience in implementation. According to Balzer enhancement is the dominant maintenance activity in the life-cycle of 'large' software systems [3]. He therefore says that "we should recognize that enhancement, not initial development, is the central software activity". In this respect one improvement of Boehm's spiral model of software process is the consideration of enhancement as mainstream software activity rather than a "second class citizen" [10].

Enhancement a macro-level activity: In principle the meaning associated to the word enhancement by the researchers in software engineering and this thesis is same. It denotes a change in structure or behavior of a program. There is however one big difference. In software engineering it is studied at a macro-level as an activity in the life of a software system.

In contrast in stepwise enhancement it is a micro-level activity. Enhancements are performed during the initial development of a system even when it is not frozen. Enhancements that are part of the initial programming process are planned; they do not appear as an after-thought. Further the enhancement activity may be even localized to the development of an individual component or subcomponents of a system; it may not necessarily span the entire system.

Potentially harmful effects of repeated enhancement: An implication of developing programs by repeated enhancement is that the initial development of a program too becomes an evolutionary activity. The final program evolves
through successive enhancements of some initial program where the evolution steps are controlled. This leads one to the realm of Lehman's second law of software evolution [51]. This law states that:

"As a program is continuously changed, its complexity, which reflects deteriorating structure, increases unless work is done to maintain or reduce it."

It is our hypothesis that a deterioration of structure happens when enhancements alter the flow of control of a program. Classifying enhancements as modulation, extension, and mutation on the basis of their effect on the flow of control turns out to be useful. Modulation and mutation alter the structure of the program and hence the flow of control. Repeated modulation and mutation could most likely lead to a spaghetti code. On the other hand extensions do not alter the flow of control in a program and hence repeated extensions cannot deteriorate the structure of a program. It is advisable therefore that when performing a sequence of enhancements one first performs the modulations, then mutations, and then the extensions. Modulation and mutation manipulate the program skeletons of the program. Once the structure has been determined extensions may be used to add meat to it without altering it. This ordering of enhancements would ensure that the eventual program does not deteriorate.

Meaning and scope of enhancement: In software engineering, any change made to a 'frozen' system for purposes other than fixing implementation bugs is considered an enhancement. According to Balzer "the set of all possible enhancement fall in two categories - those that change the domain model and those that do not" [2]. The enhancements that alter domain models he calls
structural enhancements. Enhancements that change the membership of an enumerated set is an example of structural enhancement. These are similar to the enhancements performed by Dromey [28] to expand the set of initial states, see Chapter 2. The type of enhancements we identify do not belong to the category of structural enhancements.

Balzer does not expand further on what he means by functional enhancements [2, 3]. It is therefore hard to make any comparisons.
Chapter 5

Prolog Partial Evaluator

Partial evaluation is now a well researched subject in the logic and functional programming research community. It is of interest to the Prolog world because it offers a practical solution to reduce the meta-level overhead in meta-programming. This is crucial in making meta-programming a viable method of programming for production quality programs.

Partial evaluation forms the basis for enhancement tools developed in the next section. These tools use meta-interpreters to represent the required programming knowledge and use partial evaluation to transmit this knowledge to the program to be enhanced.

While several researchers have reported partial evaluation of complete Prolog, there is no document available that provides guidance in developing a partial evaluator for Prolog. The solutions announced in the literature have mostly presented the abstract details without giving the implementation details. Since the problem and its solutions are now well understood this chapter looks at it from an engineering perspective. It describes the architecture of ProMiX a partial evaluation system for “full” Prolog. The system is decomposed into a set of components on the basis of the tasks performed during partial evaluation. This makes the development and maintenance of such a system simple. Such a decomposition of a partial evaluation system has not been done before.
Looking back it is now interesting to note that the development of the system subconsciously followed the method of stepwise enhancement. The important first step is the realization that a Prolog partial evaluator can be developed by enhancing a Prolog meta-interpreter, (compare programs in Figures 3.2 and 5.5). Once this is realized the complexity of the programming exercise changes dramatically. It also has an effect on the structure of the whole program. This can be seen by comparing the partial evaluators presented by Takeuchi and Furukawa [86] and Venken [93]. Venken enhances his partial evaluator from a meta-interpreter. As a result he comes up with a code that is crisp and easy to understand. On the other hand the partial evaluator given by Takeuchi and Furukawa [86] seems to have been developed using an ad hoc approach. As a result the flow of control is not visible from the programs structure. The code is therefore harder to understand.

A brief overview of the basic concepts of partial evaluation relevant for our needs is presented in the next section. For more details one may refer to the collection of papers in [9].

5.1 The Principle of Partial Evaluation

The notion of partial evaluation is very simple. In the context of functional programming languages it is formulated as follows: Suppose there is a function $F$ with the parameters $x_1, x_2, \ldots, x_n$. If the values of some of its parameters are known, say $x_1 = a_1, \ldots, x_k = a_k$, where $a_1, \ldots, a_k$ are constants then a specialized version $F'$ of $F$ may be generated such that

$$F(a_1, \ldots, a_k, x_{k+1}, \ldots, x_n) = F'(x_{k+1}, \ldots, x_n)$$
for all values of $x_j$, $j = k + 1, \ldots, n$. In the recursive function theory this is known as the Kleene's S-m-n theorem.

In the context of a logic programming language with pure Horn clauses partial evaluation may be stated as follows. Given a program $P$ and a goal $G$. The result of partially evaluating $P$ with respect to the goal $G$ is the program $P'$ such that for any substitution $\theta$ $G.\theta$ has the same (correct and computed) answers with respect to $P$ and $P'$. The intention is to produce a $P'$ on which $G.\theta$ runs more efficiently than on $P$.

Program $P'$, the result of partial evaluation of $P$ with respect to a goal $G$ is called the residue. It is generated by constructing the "partial" search trees for $P$ and suitably chosen atoms as goals, and then extracting $P'$ from the definitions associated with the leaves of these trees.

The interesting results in partially evaluating interpreters written in Prolog are summarized as follows.

Let $I$ be an interpreter for some language $L$ in Prolog and $P$ be a program in language $L$. Let $I(P)(D)$ denote the interpretation of $P$ by $I$ with some data $D$. The result of specializing the interpreter $I$ with respect to $P$ is a Prolog program $P'$ such that $P'(D)$ has the same (correct and computed) answers as $I(P)(D)$.

The generation of $P'$ from $I(P)$ is called "specialization of the interpreter $I$" since $P'$ is Prolog program specialized to perform some aspects of the interpreter $I$. It is also called "compiling the program $P$" because $P$ a program in language $L$ is translated to an 'equivalent' Prolog program. In the example that follows the interpreter and the program are both written in Prolog, hence the compilation effect is not very obvious.
count(true,0).
count((GoalA,GoalB), Count) :-
    count(GoalA, CountA),
    count(GoalB, CountB),
    Count is CountA + CountB.
count(Goal,0) :-
    sys(Goal),
    call(Goal).
count(Head, Count) :-
    unfold_clause(Head, Body),
    count(Body, CountBody),
    Count is CountBody+1.

Figure 5.1: count - a Prolog interpreter to count reductions

5.2 Example

Consider the Prolog program count in Figure 5.1. It performs a depth-first left-to-right execution of pure Prolog programs. The relation count(Goal, Count) gives Count the number of reductions made to resolve a Prolog goal Goal. It is a variant of the four clause meta-interpreter commonly found in literature. Modifications have been made to keep it simple, self-applicable, and executable on contemporary Prolog systems without errors.

Now consider the program should_take of Figure 5.2. It is a fragment of a simple medical expert system taken from [86]. Patient information is provided to this expert system as a set of facts complains_of/2.

The following is a sample session with this expert system and the count interpreter.

| ?- should_take(sue, Y).  
| Y = lomotil ? ; |
should_take(Person, Medicine) :-
    complains_of(Person, Symptom),
    suppresses(Medicine, Symptom).

suppresses(aspirin, pain).
complains_of(john, pain).
suppresses(lomotil, diarrhea).
complains_of(sue, diarrhea).

Figure 5.2: should_take - a rudimentary medical expert system

no

l ?- count(should_take(sue, Y), C).
C = 3, Y = lomotil ? ;
no

l ?- count(count(should_take(sue, Y), C1), C2).
C1 = 3, C2 = 6, Y = lomotil ? ;
no

The expert system makes 3 reductions to solve the goal should_take(sue, Y).
The interpreter itself requires 6 reductions to count these reductions. This is
twice the number of reductions made by the expert system. If queries for
counting reductions of should_take are made routinely, but with varying parameters, it may be worthwhile partially evaluating the execution of
\text{count}(should\_take(X, Y), C) and evaluating only the residue during actual operation.

The residue obtained from partial evaluation varies depending upon when the unfolding is stopped. When partially evaluating \text{count}(should\_take(X, Y).
count(should_take(Person, Medicine), Count) :-
    count(complains.of(Person, Symptom), Count0),
    count(suppresses(Medicine, Symptom), Count1),
    ICount is Count0 + Count1,
    Count is ICount + 1.

count(suppresses(aspirin, pain), 1).
count(suppresses(lomotil, diarrhea), 1).
count(complains.of(john, pain), 1).
count(complains.of(sue, diarrhea), 1).

Figure 5.3: Result of compiling count with respect to should.take

C) the goal may be exhaustively evaluated giving the residue:

    count(should_take(sue, lomotil), 6).
    count(should_take(john, aspirin), 6).

or the unfolding may be terminated using the strategy discussed later in Section 5.9 giving the residue in Figure 5.3.

This program may be transformed further to the one in Figure 5.4 by replacing all count(Goal, Count) atoms using the following rewrite rules:

    count(suppresses(A, B), C) → suppresses(A, B, C).
    count(complains.of(A, B), C) → complains.of(A, B, C).
    count(should.take(A, B), C) → should.take(A, B, C).

The programs in Figure 5.3 and 5.4 are equivalent in the sense that if Goal → NewGoal holds and Goal succeeds for the program in Figure 5.3 then NewGoal succeeds for the one in Figure 5.4 and returns the same answers, and vice versa. The answers returned by:

    | ?- should.take(sue, Y, C).
    | C = 3, Y = lomotil ? ;


should\_take(Person, Medicine, Count) :-
  complains\_of(Person, Symptom, CountA).
  suppresses(Medicine, Symptom, CountB),
  ICount is CountA + CountB,
  Count is ICount + 1.

  suppresses(aspirin, pain, 1).
  complains\_of(john, pain, 1).
  suppresses(lomotil, diarrhea, 1).
  complains\_of(sue, diarrhea, 1).

Figure 5.4: Result of transforming residue

no

are equivalent to the answers for count(should\_take(sue, Y), C), as shown before.

The program of Figure 5.4 is however more efficient than the one it is transformed from. The reasons are discussed in the next section.

5.3 Issues in Partial Evaluation

It is our intent to develop a system that is specifically suitable for partially evaluating interpreters written in Prolog with respect to a given object program. The process effectively compiles object programs interpreted by the interpreter into Prolog. This is also called specializing interpreters. The example above demonstrates the various stages in a typical specialization application.

The problem of developing such a system may be decomposed into the following subproblems.

- How to generate residues for a goal?
- How to control unfolding?
• How to collect residues to get a correct and compiled program?
• How to rewrite the residues to achieve greater efficiency?

These subproblems provide the basis for decomposing a partial evaluation system. Though the questions are suitable for any general purpose partial evaluation system, the corresponding software components in ProMiX are engineered for the purpose of specializing interpreters. The following text discusses these issues.

5.3.1 Generating residue

The clause in Figure 5.3

\[
\text{count(should\_take(Person,Symptom), Count) :-}
\]
\[
\text{count(complains\_of(Person,Symptom), Count0),}
\]
\[
\text{count(suppress(Medicine,Symptom), Count1),}
\]
\[
\text{ICount is Count0 + Count1,}
\]
\[
\text{Count is ICount + 1}
\]

is a residue for the goal

\[
\text{count(should\_take(Person,Symptom), Count).}
\]

Creation of residues is an issue central to the partial evaluation problem. The complete program in Figure 5.3 is a collection of residue clauses.

5.3.2 Controlling unfolding

As was illustrated in the example above, the result of partial evaluation is not unique. On one extreme a goal may be completely evaluated, on the other it
may be left as it is. The latter case is redundant as it doesn't achieve anything. Whereas complete evaluation of any goal for a pure logic program returns a set of all the true relations for that goal, assuming the evaluation terminates successfully.

Due to the functional and extra-logical predicates in practical Prolog programs it may not always be possible to exhaustively evaluate a goal with incomplete inputs. Besides, partial input may also lead to infinite computation for recursive programs or a combinatorial explosion of residues.

The termination of partial evaluation may be guaranteed by terminating the unfolding process before entering an infinite loop or on detection of one. A general criterion that detects whether any computation would terminate can never be correct. Such a criterion would solve the halting problem for Prolog programs, and hence for Turing machines. It is however possible to develop a specialized set of criteria specific to an interpreter.

Control of unfolding is also important in the presence of predicates with side-effects, for example write/1, and assert/1. Such predicates should be detected and suspended during partial evaluation. They can only be evaluated at runtime.

5.3.3 Collecting residue

The residue from partial evaluation of the goal

\[ \text{count(should\_take(Person\_Symptom), Count)} \]

given the program in Figure 5.1 and 5.2 consists only of the first clause of Figure 5.3. The goals such as \( \text{count(complains\_of(Person, Symptom), Count0)} \) retained in the residue still require the interpreter and the object program
for further evaluation. To remove the need for interpretation completely these
goals may be partially evaluated too. The program consisting of the collection
of such residues can potentially be executed without the interpreter or the
object program.

We say 'potentially' because there are several situations where an inter-
preter goal cannot be partially evaluated and would need the interpreter for
evaluation at runtime. Besides the residue, definition of support procedures
used by the interpreter but not evaluated during specialization may also be
needed by the compiled program.

5.3.4 Rewriting atoms

The program in Figure 5.3 is a result of partially evaluating the interpreter goal
count(Goal, Count) with Goal instantiated to different procedures of the object
program. As it is the result of partial evaluation of count/2 goals all the clauses
of the resultant program have count/2 as their head atom. Such programs
where all clauses of the residue belong to just one procedure is common while
specializing interpreters.

The program in Figure 5.4 is a translation of the above program. The
translation defines a mapping between the atoms of the two programs such
that the computed answers of two mapped goals is the same. Rewriting splits
the second program into small procedures. This buys an extra level of indexing
in the transformed program. The translated program is therefore more efficient
than the original program.

The issue of translating programs as from Figure 5.3 to Figure 5.4 is not
necessary for partial evaluation. But the additional efficiency achieved is im-
portant when specializing interpreters. It also makes the residue more readable which may be helpful if the residue is to be processed by a human being.

5.4 System Architecture

The ProMiX system consists of three major subcomponents. They are decomposed around the subproblems stated in the previous section and have the following responsibilities.

Kernel: generate and rewrite residue for one goal.

Driver: provide user interface, drive the kernel, and collect residue.

Knowledge Base: maintain unfolding and rewrite rules.

The kernel is responsible for generating the residue for one interpreter goal. It is driven by the driver for partially evaluating different goals the collection of whose residues form the final program. The task of generating a residual clause requires collecting the subgoals whose evaluation may be suspended and translating the residue goals into more efficient forms. These two activities may be further separated to be performed by separate program components: mix and rewrite. Mix generates the residue in terms of the interpreter goal. Rewrite translates these interpreter goals to a more efficient form.

5.5 Mix

Most research in partial evaluation of Prolog has focussed on developing mix. This component forms the core of any partial evaluator as the correctness of the transformation rests largely on the functional equivalence of the residue.
% mix(Goal, Residue)
mix(true, true) :-!,
mix((GoalA,GoalB), (MixA,MixB)) :-!,
    mix(GoalA, MixA),
    mix(GoalB, MixB).
mix(Head, Mix) :-
    should_unfold(Head) !,
    unfold_clause(Head, Body),
    mix(Body, Mix).
mix(Goal, Goal).

Figure 5.5: Mix kernel for pure Prolog

generated with respect to the original goal. The task involved in doing so is not specific to specialization of interpreters. Thus mix is a generic component suitable for any partial evaluator.

When the input consists of “pure” Prolog mix can be derived from a meta-interpreter for “pure” Prolog. Figure 3.2 contains the interpreter from which the partial evaluator of Figure 5.5 is derived. The derivation from solve/1 to mix/2 is obvious.

In the relation mix(Goal, Residue), Residue is a sequence of goals suspended during the evaluation of Goal. A goal X is unfolded if the rule should_unfold(X) succeeds, otherwise it is suspended. The criterion for unfolding rules are developed in Section 5.9.

This would be sufficient if Prolog were truly logical, but it has side-effect predicates, extra-logical predicates, and the cut. These lead to several interesting problems that should be tackled by the partial evaluator.

The relation between an interpreter and mix of “pure” Prolog may be extrapolated to that for “full” Prolog. The mix for “full” Prolog (except cut)
in Appendix C.1.1 follows from the interpreter for "full" Prolog (except cut) of Appendix B. However, the extrapolation is not intuitive and needs explanation.

The extensions from "pure" to "full" Prolog involves adding 1) control constructs, 2) extra-logical predicates, and 3) side-effecting predicates. The problems associated with partially evaluating programs in the presence of side-effecting and and extra-logical predicates have been studied by Venken [93, 94]. He classifies the problems as

- *backward unification*, and
- *multiple clauses* problems.

The problems are described here along with their solutions.

The control constructs and predicates added to "full" Prolog may be classified as

- *cut*
- *all solutions predicates*
- *additional control constructs*.

Partial evaluation of these is discussed in reverse order. The discussion is deferred until after *backward unification* and *multiple clauses* since the code developed to solve these two problems is also useful in processing the additional constructs and all solutions predicates. Other than the control constructs there are several system primitives in Prolog. These primitives may be specialized by explicitly providing their definition in Prolog. This is discussed under the heading:

- *reifying system primitives*
Backward Unification

A goal is unfolded by replacing it with the body of a clause whose head unifies with it. The bindings due to this unification propagate backward and forward through a program. Forward propagation is useful during partial evaluation as it helps narrow the options for predicates evaluated later. Backward propagation is not helpful at compile time. It instead leads to problems if the bindings are propagated to variables of the preceding extra-logical or side-effects predicates.

For instance, unfolding only $\text{ilike}(X)$ in the query

? - fruit(X), write('Hello World'), ilike(X), ...

with respect to the program

fruit(orange). fruit(apple). ilike(apple).

would result in

? - fruit(apple), write('Hello World'). ...

The run-time behavior of the two queries are not equivalent. The first one writes "Hello World" twice, whereas the second does so only once.

The problem arises because backward unification propagates the value of $X$ to the goal $\text{fruit}(X)$. In the original query the predicate $\text{fruit}(X)$ works as a 'generator'; it generates multiple solutions on backtracking. It is converted to a 'tester' due to backward unification; it succeeds only once. This change in behavior affects the number of times $\text{write}/1$ is performed since it appears after $\text{fruit}/1$. 
A correct transformation for the above query would be:

?- fruit(X), write('Hello World'), X = apple, ... 

where unifications due to unfolding ilike(X) have been converted into runtime unifications. It may be noted however that occurrences of X after the goal ilike(X) may still be unified to apple without affecting the query.

A similar problem arises with extra-logical predicates, such as var/1. A goal var(X) whose argument X is not bound during partial evaluation may not be unfolded because when the complete data is given X may actually be bound. Such a goal should thus be left in the residue. The variable X should also be guarded against possible binding due to backward propagation of values.

For example, the unfolding of ilike(X) from the previous example in the clause

p(X):- var(X), ilike(X), ...

would give:

p(apple) :- var(apple), ... 

This clause would always fail. A correct result would be

p(X) :- var(X), X = apple, ...

The above behavior may be reflected in the miz code of Figure 5.5 by including the following as its second clause.

mix((X,Y), (MixX, MixY)) :-
    (side_effect(X); extra_logical(X));!,
    mix(X, MixX),
    mix_copy(Y, MixY).
where mix_copy/2 generates the residue for a copy of Y. The residue is preceded by a sequence of unification goals for binding the variables in Y to its copy. The unifications are performed at runtime.

mix_copy(X, (Unifs, MixX)) :-
    copy(X, CopyX),
    variables_in(X, XVars),
    variables_in(CopyX, CopyVars),
    mix(CopyX, MixX),
    generate_unifications(XVars, CopyVars, Unifs).

The predicate CopyX is a term identical to X but for variable renaming; it does not use variables that have already appeared in the program. XVars and CopyVars are the lists of variables in X and CopyX, respectively, such that the ith element in XVars is a variable that is replaced by the variable at the ith position in CopyVars. The binding of variables due to partial evaluation of CopyX is reflected in the list CopyVars. The bindings in this list along with XVars are used to generate a sequence of unifications between variables in XVars with their corresponding element in CopyVars. This sequence becomes part of the residue and precedes the residue from CopyX.

The code for generate_unifications/2 is as follows:

generate_unifications([], [], true).

generate_unifications([Var|XVars], [CopyVar|CVars], TUnifs) :-
    var(CopyVar), !,
    Var = CopyVar,
generate_unifications(XVars, CVars, TUnifs).
generate_unifications([Var|XVars], [CopyVar|CVars], Unifs) :-
generate_unifications(XVars, CVars, TUnifs),
merge_unifications((Var = CopyVar), TUnifs).
merge_unifications(Unif, true, Unifs) :-!.
merge_unifications(Unif, Unifs, (Unif, Unifs)).

The code for variables_in/2 and copy/2 is in the Appendix C.1.4.

The relations side_effect(X) and extra_logical(X) guide the partial evaluation. They are provided as a set of declarations, like:

side_effect(write(_)).
side_effect(assert(_)).
side_effect(X) :- var(X).
extra_logical(var(X)) :- var(X).
extraLogical(nonvar(X)) :- var(X).

A variable goal $X$ that is not instantiated at compile time is classified as having side_effect since it could potentially be bound to a side-effecting predicate at runtime.

The above properties are not limited to system primitives. They may be propagated to user-defined procedures that use these system predicates. It is for this reason that in the mix/2 clause above a goal is partially evaluated even though it is extra-logical or generates side-effect.

The declaration of system primitives with side-effect or extra-logical behavior and the rules to propagate these properties to user-define predicates
are maintained by the knowledge base module of the system. It is described in Section 5.8.

Multiple Clauses

Another interesting problem arises when unfolding goals that have multiple residues. Normally separate residue clauses (or goals) may be generated for every residue of the predicate. For instance, consider the program

\[ \text{vegetable}(X) ::= P, \text{fruit}(X). \]
\[ \quad \text{fruit(orange). fruit(apple)}. \]

where \( P \) is some sequence of predicates. Unfolding \( \text{fruit}(X) \) in \( \text{vegetable}/1 \) we get:

\[ \quad \text{vegetable}(\text{orange}) ::= P'. \]
\[ \quad \text{vegetable}(\text{apple}) ::= P''. \]

where \( P' \) and \( P'' \) are derived by applying the corresponding substitutions for \( X \) to \( P \).

Now if \( P \) causes a side-effect, the query

\[ ?- \text{vegetable}(X). \]

would perform the side-effect twice in the residual program, and only once in the original program. On the other hand the extra-logical predicates contained in \( P \) may use the variable \( X \). The unfolding of \( \text{fruit}(X) \) propagates the bindings to \( X \) backwards and thereby splitting the original clause in two. In the presence of extra-logical or side-effecting predicates in \( P \) this raises the question of correctness of the transformation. In the absence of such predicates in
P correctness is not a problem; there is however another problem. Splitting of the original clause may sometimes degrade the performance even though it preserves the behavior. This happens when P has predicates that perform some expensive operation. The residual program generated due to splitting the clause would perform such an operation multiple times thereby becoming less efficient.

A solution to all these problems is to expand the multiple residue as a sequence of disjunctions in the same clause, such as:

\[ \text{vegetable}(X) :\!\!: P, (X = \text{orange}; X = \text{apple}). \]

Here \((X = \text{orange}; X = \text{apple})\) represents the ‘bag of all residues’ from partially evaluating the goal \(\text{fruit}(X)\). This behavior can be accomplished by adding to the kernel of Figure 5.5 the clause:

\[ \text{mix}((X,Y), (\text{MixX}, \text{MixY})) :\!\!: \]
\[ (\text{side_effect}(X); \text{extra_logical}(X)), !, \]
\[ \text{mix}(X, \text{MixX}), \]
\[ \text{mix_bag}(Y, \text{MixY}). \]

in place of the clause added to solve the backward unification problem. The procedure \(\text{mix_bag}(Y, \text{MixY})\) computes \(\text{MixY}\) a ‘bag of all the residues’ of \(Y\). The ‘bag’ is represented as a sequence of disjunctions with the elements of this sequence corresponding to the residues for the goal \(Y\).

A first attempt at defining \(\text{mix_bag}/2\) is:

\[ \text{mix_bag}(\text{Goals}, \text{Residue}) :\!\!: \]
findall(R, mix(Goals, R), AllR),

bag_from_list(AllR, Residue).

The findall/3 returns a 'list' of all residues for Goal. This list is then converted to a 'bag'. An empty list translates to a bag with just fail.

The above clause demonstrates the important idea behind mix_bag/2; it is however incorrect. The all solution predicate findall/3 works on a copy of the goal. It returns the solution in terms of a fresh set of variables. Hence, the solutions returned by findall/3 do not use the variables in Goals. This makes it impossible to relate the solutions in the residue with the variables used in the original program.

With this definition of mix_bag/2 partially evaluating the previous example would give:

vegetable(X) :- P, (.142 = orange; .342 = apple).

where .142 and .342 are arbitrary variables. In a correct residue these variables should be replaced by X. This may be done by unifying these variables with X. Note only the variables generated corresponding to the variable X from the original goal should be unified with X. This should be done for each solution returned by findall/3. The unifications generated in the bag, such as .142=orange, are not performed at compile time. Doing that would propagate values across different branches of the disjunction which is undesirable.

This logic is incorporated by the following definition of mix_bag/2.

mix_bag(Goals, Residue) :-

variables_in(Goals, Vars),

findall((R,Vars), (nonvar(Vars), mix(Goals, R)), AllR),
generate_unifications_for_bag(AllR, Vars, ResidueList),
bag_from_list(Residue, ResidueList).

The findall/3 goal returns a list of pairs. The first element of each pair is a residue from partially evaluating Goals. The residue uses a fresh set of variables in place of variables used in the goal. The second element returns the bindings made in the residue to these fresh variables. The bindings are generated by passing Vars the list of variables in the Goals along with the mix/2 goal. When variables in Goals are renamed the renaming is reflected in this list. The predicate nonvar/2 is used only to carry the list of variables; it always succeeds.

The predicate generate_unifications_for_bag/3 uses the Vars and the list of bindings for variables in the residue to generate a sequence of unifications for all the solutions returned by findall/3.

Control Constructs

"Full" Prolog consists of several control predicates besides conjunction. Some of those are not/1, ;/2 (disjunction), and \rightarrow/2 (if-then-else). The kernel of Figure 5.5 partially evaluates only goals connected by conjunctions. For a practical system the other predicates need to be taken care of as well.

The problems in partially evaluating these control predicates are similar to that in partially evaluating conjunctive goals containing side-effect predicates and that in generating multiple residues. Their residue cannot be split in clauses. Also the bindings due to computing the residues should not be propagated to the rest of the clause unless it is committed to be on the path of evaluation. The latter is possible when there is only one such path and only
one residue. In conforming with the spirit of partial evaluation whenever such an opportunity arises it should be availed.

The \textit{mix} in Appendix C.1.1 partially evaluates all these predicates. It is written with the above principles in mind. An underlying uniformity in evaluating these predicates is that all the residue of their subgoals should be computed at the same time. That explains the use of \textit{mix\_bag/2} throughout.

When a goal is completely evaluable the result of its evaluation may be used for making control decisions. For instance, in \( A \rightarrow B; C \) if \( A \) is completely evaluable the success or failure of its evaluation may be used for committing to \( B \) or \( C \), respectively, during partial evaluation. Otherwise \( B \) and \( C \) may be partially evaluated but not committed to.

The bag of residues is completely evaluable if it contains only one residue and that, besides the sequence of unifications generated internally, is either \textit{true} or \textit{fail}. Such a residue may be completely evaluated. If the bag contains only one residue but it is neither \textit{true} nor \textit{fail} then it is not fully evaluable. However, because there is only one residue the unifications generated in the bag may be performed at compile time. When a bag has multiple residues this cannot be done.

The predicate \textit{mix\_bag\_eval/2} encodes this logic to evaluate the residue as much as possible.

\[\text{mix\_bag\_eval(Goal, Residue)} : -
\]

\[\text{mix\_bag(Goal, MixGoal),}
\]

\[\text{determine\_bag(MixGoal, Residue).}
\]

\[\text{determine\_bag(fail, _Residue)} : - !, \text{fail.}
\]

\[\text{determine\_bag((Unifs,Residue), Residue)} : - !, \text{call(Unifs).}
\]
determinate_bag(Residue, Residue).

All Solutions Predicates

The findall/3 predicate used in mix_bag/3 provides another interesting case for partial evaluation. The residue for partially evaluating

?- findall(X, fruit(X), Y).

with respect to the program

fruit(orange). fruit(apple).

should be:

?- Y = [orange, apple].

Here the findall/3 goal has been completely evaluated. However, partially evaluating

?- findall((X,Y), (fruit(X). p(Y)), Z).

when p/2 cannot be evaluated at compile time, the residue be:

?- findall((X,Y), (X = orange. p(Y); X = apple, p(Y)), Z).

Such a result can be obtained by performing

mix_bag(Y, MixY), Residue = findall(X, MixY, Z).

for:

mix(findall(X,Y,Z), Residue)
and evaluating the *Residue* if all the disjunctions in the bag \( \text{Mix}Y \) are evaluable at compile time.

Generally speaking a set predicate cannot be fully evaluated at compile time. This is because the set of solutions returned at compile time may be further constrained if the variables of the partially evaluated goal are bound at runtime.

The treatment for *bagof/3* follows similarly by replacing *findall/3* for the respective predicate. There is a subtle problem. The unfolding of \( Y \) introduces new variables in the residue \( \text{Mix}Y \). These variables do not change the set of solutions returned by *findall*(\( X, Y, Z \)) as it is existentially quantified over all variables in \( Y \). In case of *bagof/3* the solution should be explicitly quantified over the new variables.

The correct behavior of:

\[
mix(\text{bagof}(X,Y,Z), \text{Residue})
\]

would be performed by the goals

\[
\begin{align*}
\text{mix\_bag}(Y, \text{Mix}Y), \\
\text{extra\_variables}(\text{Mix}Y, Y, \text{Vars}), \\
\text{Residue} = \text{bagof}(X, \text{Vars}\uparrow\text{Mix}Y, Z).
\end{align*}
\]

Where *extra\_variables/3* gives the variables \( Vars \) that are in \( \text{Mix}Y \) but not in \( Y \).

The code in the Appendix C.1.1 reflects this logic. The treatment of *setof/3* is analogous to that of *bagof/3*. 
Cut

The cut in Prolog is perhaps the most widely misused control construct. In most cases it may be done away in favor of the if-then-else construct [61]. But there are cases, such as a cut in disjunction, when it can be replaced. To partially evaluate “full” Prolog it is necessary for the partial evaluator to handle cut.

The problem in handling cut arises due to its non-local behavior. A clause containing a cut cannot be simply unfolded in the body of a procedure calling it. For instance, unfolding \( p(X) \) in

\[
f(X) \leftarrow \ldots, p(X).
\]

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

\[
p(1) \leftarrow !.\]

\[
p(X) \leftarrow \text{write}(X).
\]

leaves the residue

\[
f(X) \leftarrow \ldots, (X=1,!; \text{write}(X)).
\]

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

The programs are obviously not identical in behavior.

Venken [93] introduced the predicates \( \text{mark}(.) \) and \( !(.\) for handling cut. Using the method he proposes the result of unfolding \( p/1 \) in the above program would be:

\[
f(X) \leftarrow \ldots, \text{mark}(1), (X=1;!((1; \text{write}(X)).
\]

\[
f(X) \leftarrow q(X).
\]
where \( \text{mark}(1) \) delimits the scope of \( !/(1) \). Execution of \( !/(1) \) cuts only the choice-point up to \( \text{mark}(1) \) therefore preserving the program's behavior.

As \( \text{mark}/1 \) and \( !/1 \) (or their equivalent predicates) are not supported by most Prolog systems this is not a practical solution.

When partially evaluating a program we need to bother only about \(!\) for which the goals preceding it are not fully evaluable at compile time. In an interpreter cuts are used for committing decisions made on the structure or property of the string being interpreted. In our experience such usage could be attributed to almost all the cuts in most interpreters. The evaluation of these cuts depends only on the object program and not its data. Since during specialization of an interpreter the object program is specified, these cuts could be compiled away.

For the cuts that remain we collect the goals in the scope of the cut and create a single clause procedure out of it. Variables that are common to the scope and the rest of the clause are chosen as arguments. For the above example this strategy returns the original program as residue.

The code for handling cut is not presented as it is too complex to explain and tends to be messy.

**Reifying System Primitives**

Consider partially evaluating the query

\[
?- f(X) = f(Y).
\]

A naive but incorrect partial evaluation would simply succeed after unifying variables \( X \) and \( Y \). A correct but unoptimal result would be achieved by
leaving the query as it is. Where as the correct and optimal result from partially evaluating the above query would be:

?- X = Y.

The above result cannot be derived if the partial evaluator depends on the system supported definition of =/2. This is because the built-in definition of =/2 is a unit operations and cannot be decomposed by the partial evaluator. The result can instead be achieved by simulating the behavior of the primitive =/2 by a Prolog predicate say unify/2, and partially evaluating the query as a call to this predicate.

Making implicit behavior of a system explicit is called *reification*. System primitives are treated as unit operations by a naive partial evaluator because they cannot be decomposed. Using the built-in support, system goals arising in the residue may either be evaluated during partial evaluation or left in the residue. To generate efficient residues these primitive may be reified so that they may be evaluated as far as possible.

This can of course be done for only those primitives that can be written in Prolog by using some ‘lesser’ features of Prolog. Built-in predicates of Prolog that have side-effects cannot be decomposed in this way. There are times when a predicate written using side-effecting predicates can be rewritten using extra-logical features of Prolog. A case in point is the predicate *copy/2*. This predicate is usually defined using *assert/1 and retract/1* to make copies of variables [84].

\[
\text{copy}(X, Y) :- \text{asserta}('\$copy'(X)), \text{retract}('\$copy'(Y)).
\]

Now partially evaluating the query
?- copy(f(X), Y), p(Y).

with the above definition of copy/2 one gets:

?- asserta('$copy'(f(X))), retract('$copy'(Y)). p(Y).

This is no more efficient than the initial query. Knowing the behavior of copy/2 the preferred residue would be:

?- copy(X, Z), p(f(Z)).

This can generate more efficient residue because the partially instantiation of the arguments of predicate p/1 can be used to partially evaluate it further. This result is not achievable using the definition of copy/2 just presented, but can be achieved if the alternate definition of copy/2 given in the Appendix C.1.4 is used. The latter definition is more complex to write, but is more suitable for the purpose of partial evaluation.

5.6 Rewrite

The rewrite module provides support to translate residue clauses by replacing the 'left over' interpreter goals with new goals as shown in the example in Section 5.2. This activity may not be necessary for general partial evaluation applications.

The code for this module is presented in Appendix C.1.5. It is easy to understand and hence is not explained piece by piece. The translations performed by rewrite are purely syntactic in nature. Its salient features are

a) it translates “full” Prolog,
b) it uses memoing techniques to remember rewrite relations, and

c) its rewrite rules are easily modifiable to suit different applications.

Most of rewrite's code is general enough to be used for syntactic translation of Prolog programs elsewhere. A discussion of its overall logic follows.

**Rewrite One Predicate**

The procedure rewrite_rule/2 translates an interpreter goal. Given an interpreter goal IGoal with meta-argument Obj, rewrite_rule(IGoal, Pred) translates it to the term Pred. This term's principal functor is the same as the principal functor of Obj. Its arguments are generated by concatenating the arguments of Obj with the arguments, except the meta-argument, of IGoal.

In the interpreter of Figure 5.1 the first argument of count/2 is its meta-argument. The following query shows the translation of a count/2 goal taken from the residue in Figure 5.3.

```prolog
?- rewrite_rule(count(should_take(X,Y),C),Z).
Z = should_take(X, Y, C) ?
```

Only variables relevant to the discussion are shown in the result.

An interpreter goal can be rewritten only if its meta-argument is instantiated. When the meta-argument is a variable the interpreter goal is returned without modification. Example:

```prolog
?- rewrite_rule(count(X,Y), Z).
Z = count(X, Y) ?
```
An interpreter goal is also retained in the residue if there is no residue-procedure corresponding to it.\footnote{Residue Procedure should be defined in Section 5.2} This happens when the definition for its meta-argument is inaccessible during partial evaluation.

The definition of a system primitive or a compiled predicate is not accessible. It is also not accessible if the predicate gets defined at runtime. This happens when an object program defines and alters predicates dynamically using assert and retract. Similar condition arises when the definition of an object program procedure is not available during partial evaluation and is provided only during execution.

In all these cases the interpreter goal is retained in the translated program; the interpreter is not compiled away completely. The translated program requires the interpreter during execution to execute the retained interpreter goals.

All decisions made by the procedure rewrite_rule/2 are easily alterable as they are implemented using separate procedures. The choice of name for a translated procedure, or the order of its arguments, or the decision on when to retain an interpreter goal can be affected by performing changes local to these procedures. This flexibility is achieved at the cost of repeating certain operations across these procedures.

**Memoing Rewrite Relations**

Interpreter goals are rewritten by making extensive use of system primitives to split and create structures. These primitive operations are rather expensive in time. The rewrite module uses a memoing technique to remember the translations performed by rewrite_rule/2.
The procedure `expand_predicate/2` provides the memoing support. The predicate `rewrite_rule/2` is called through this procedure. When a translation is requested `expand_predicate/2` first checks if the requested goal has been translated before. It calls `rewrite_rule/2` only if the check fails. The operand and result of `rewrite_rule/2` are asserted as '#$$rewrite'/2 relations for later use.

```
expand_predicate(Goal, RGoal) :-
  check_rewrite_relation(Goal, RGoal).

expand_predicate(Goal, RGoal) :-
  is_interpreter_goal(Goal),
  rewrite_rule(Goal, RGoal),
  store_rewrite_relation(Goal, RGoal).
```

The procedures `check_rewrite_relation/2` and `store_rewrite_relation/2` implement the memoing capability. The test `is_interpreter(Goal)` is performed inside `expand_predicate/2`, rather than in its caller. This keeps its calling procedure general purpose and usable for translations elsewhere.

In addition to memoing another trick is used to reduce the number of calls to `rewrite_rule/2` and the number of relations asserted.

Before applying `rewrite_rule/2` the interpreter goal is translated to a template. The template is a term with the same functor and arity as the initial goal but has all the arguments, except the meta-argument, uninstantiated. The meta-argument is replaced by a term with the functor and arity of its counter-part in the original goal but with variable subterms. All the variables used in the template are new and different.

The conversion of an interpreter goal to its template is done by the procedure `create_goal_template/2`. The following query demonstrates its use:
?- create_goal_template(count(should_take(sue,lomotil),3), Z).

Z = count(should_take(_101, _102), _103) ?

Using this predicate expand Predicate/2 is modified to translate the template of an interpreter goal instead of the interpreter goal itself.

expand Predicate(Goal, RGoal) :-
    check_rewrite_relation(Goal, RGoal).

expand Predicate(Goal, RGoal) :-
    is_interpreter_goal(Goal),
    create_goal_template(Goal, GMetaGoal),
    rewrite_rule(GMetaGoal, RGoal),
    store_rewrite_relation(GMetaGoal, RGoal),
    GMetaGoal = Goal.

When there are more than one interpreter goals with the same template the new definition is more efficient. The overheads introduced in creating a template is paid off by reducing calls to rewrite_rule/2.

"Full" Prolog

The expand Predicate/2 and rewrite_rule/2 procedures translate only one predicate at a time. A whole clause or a sequence of goals is translated by the procedure rewrite/2. This procedure does with expand Predicate/2 what Prolog does with expand_term/2, but with a difference. Only terms treated in the program as Prolog goals are translated using expand Predicate/2.

Terms appearing in a conjunctive sequence, in the head and body of a clause, and as arguments of meta-predicates have the potential to become
Prolog goals. A conjunctive sequence and a clause are explicitly decomposed by the procedure rewrite/2 to access their subterms. However, the traversal of various meta-predicate is not coded explicitly in rewrite/2. Instead it is guided by meta-knowledge about the usage of a meta-predicate's arguments provided by the relation meta_property(Type, Predicate, Property):

\[
\text{meta_property}(\text{meta}, \rightarrow(\_,\_), \rightarrow(+,+)).
\]

\[
\text{meta_property}(\text{set}, \text{bagof}(\_,\_,\_), \text{bagof}(-,+,+)).
\]

The relation gives the Property of the arguments of a Predicate. The term Property is of the same functor and arity as the term Predicate. There is a either a ‘+’ or a ‘-’ at the argument positions of Property. A ‘+’ indicates position for a Prolog goal or a meta-argument, a ‘-’ otherwise.

The relation also associates a Type to a meta-predicate Predicate. This is a provision to classify the meta-predicates into different groups, for example set and meta. This classification is not useful for the sake of translating predicate names. It may be used by other procedures that test the property of a predicate and choose different course of actions for predicates of different type. The partial evaluator kernel itself is an example of such a procedure.

A meta_property/3 fact is written for every meta-predicate. The rewrite/2 procedure uses the meta_property/3 declarations to selectively translate meta-arguments. The Appendix C.3 contains similar declarations for disjunction, not, call, and all solutions predicates.

The 'meta' property of a predicate propagates to procedures using it. Predicates whose arguments are passed to a meta-predicate as meta-arguments are also meta-predicates. This is similar to the propagation of side-effect property discussed in Section 5.5 and may be treated in a similar fashion.
5.7 Driver

The \textit{miz/2} predicate, as one may recall, is a general purpose partial evaluator. It has no restriction on the nature or the intended usage of its data except that they be Prolog programs. The input to \textit{miz} is a goal and its output a sequence of goals suspended while partially evaluating the goal. The evaluation is performed with respect to program definitions stored in the Prolog database. The suspension of goals is controlled by rules provided separately.

The important issues for using \textit{miz} as a practical tool for specializing interpreters are related to:

- creating goals to partially evaluate,
- creating clauses from residues,
- driving \textit{miz} to compile a program, and
- providing an easy and flexible user interface.

These issues are not at the heart of the partial evaluation problem; but are important for the utility and generality of the tool. They are discussed in the following text.

Create Goals for Partial Evaluation

The clauses in Figure 5.4 are the result of partially evaluating the following three interpreter goals:

\begin{align*}
\text{count(should\_take(X,Y), Z). or} \\
\text{count(complains(X,Y), Z). or} \\
\text{count(suppresses(X,Y), Z).}
\end{align*}
These goals specialize the interpreter count/2 of Figure 5.1 with respect to the program in Figure 5.2. The meta-argument of each goal corresponds to a procedure of the Prolog program in Figure 5.2.

Given IntGoal - the skeleton for an interpreter goal, IntArg - the variable at its meta-argument position, and Rhead - a “rule head”, the following clause creates an interpreter goal for partial evaluation.

```
create_goal_to_peval(IntGoal, IntArg, Rhead, PevalGoal) :-
copy((IntArg,IntGoal), (Rhead,PevalGoal)).
```

This plugs the term Rhead into the meta-argument position of the copy of the interpreter goal. The interpreter goal is copied to protect the skeleton goal from being instantiated because the same skeleton term may be used for creating other interpreter goals.

A “rule head” is a term used by an interpreter for unfolding “rules” of the language it interprets. For Prolog interpreters “rules” correspond to clauses and “rule head” to the head of clause. Generally, the syntactic positioning of such an atom in a rule would depend on the definition of the language being interpreted. This information is provided as a set of declarations, example:

```
% Declarations for Prolog

rule_head((X:-Y), T) :-!, generalize_term(X, T).
rule_head(X, T) :- generalize_term(X, T).

% Declarations for DCG

rule_head((X -> Y), X) :- generalize_term(X, T).
```

The predicate generalize_term(X, T) creates a term T with the same functor and arity as X but with uninstantiated arguments:
generalize_term(Term, GeneralTerm) :-
  functor(Term, Functor, Arity),
  functor(GeneralTerm, Functor, Arity).

The arguments of the meta-argument carry inputs and outputs of the object program. Since the intent is to specialize an interpreter the arguments of the meta-argument may not be instantiated.

The interpreter goals required to specialize a given interpreter can be generated from the object program using the predicates defined above.

Create Clause from Residue

The unfolding rules of Section 5.9 are designed to stop unfolding interpreter goals that have rule head terms as meta-arguments. This is done to achieve a correspondence between object program statements and residual clauses. With this strategy mix/2 returns the interpreter goals created above without unfolding.

To compile a rule the interpreter should be executed so that it unfolds the rule head once. This unfolding is performed by drive_mix/2 before calling mix/2.

\[\text{drive_mix(MetaGoal, (MetaGoal :- MixedBody)) :-}
\]
\[
\text{unfold_clause(MetaGoal, Body),}
\]
\[
\text{mix(Body, MixedBody).}
\]

The goal clause/2 does not directly unfold an object program rule. Instead it unfolds an interpreter clause, since MetaGoal is an interpreter goal. The object program rule is unfolded due to unfolding the interpreter clause that unfolds
a rule. This procedure \textit{drive.mix/2} creates a residue clause with the goal at the head and its residue as the body.

The body of the residual clause requires some further processing. Interpreter goals left as residue in the clause should be rewritten using \textit{rewrite/2} discussed in Section 5.6. Some processing is also required to remove extraneous \textit{true} goals generated as residues during various stages in the partial evaluation. Besides the body also contains a sequence of nested conjunction. The conjunctions are "flattened".

This postprocessing of residue and 'rewriting' of its residual interpreter goals is performed by \textit{peval/2}.

\begin{verbatim}
peval(MetaGoal, ResidueClause) :-
    drive.mix(MetaGoal, MixedClause),
    flatten.conjunction(MixedClause, FlattenedClause),
    rewrite(FlattenedClause, ResidueClause).
\end{verbatim}

The code for \textit{flatten.conjunction/2} is in Appendix C.1.4.

\textbf{Drive mix to Compile Program}

With the components developed in the previous sections compiling a whole program is a routine exercise. The procedure \textit{peval.program/4} takes the interpreter goal, position of the meta-argument, and the list of rule heads as input. It creates an interpreter goal for every rule head in the list and collects all the residual clauses in a list.

The clauses collected are the compiled result. It is likely however that this program is not complete, that is it does not have all the procedures required
to execute. The residue from partial evaluation may inherit goals from the interpreter that perform semantic actions. Unlike the residue in Figure 5.4 these goals may not always be system primitives. To execute the compiled program the definitions of these goals would be required. These may be provided by including the complete interpreter with the compiled program. The interpreter is also required for executing interpreter goals that were not 'rewritten' but were retained in the residue as is.

The residue may also inherit Prolog goals from the object program. This happens when a language allows Prolog goals in its rule. DCG is one such language. Its interpreter uses a call/1 to pass the Prolog goals to the Prolog engine. If these goals are not all system primitives it means that the object program actually has two distinct parts. One written in the language being interpreted (language part) and the other in Prolog (Prolog part). The Prolog part should be included in the compiled result for completeness.

The matter becomes complex when the interpreter being specialized itself interprets a subset of Prolog. A Prolog interpreter like count interprets only conjunction and user-defined goals. It uses the Prolog engine to evaluate meta-logical predicates like disjunction, if-then-else, and all solution predicates. These predicates may have user-defined goals as meta-arguments. Thus a user-defined goal may be evaluated by the interpreter as well as the Prolog engine. For instance, the object program used for specializing count belongs to both the language part and Prolog part. Thus it should be included with the compiled result.

The possibility of retaining interpreter goals creates a similar complexity for any language. The goal retained may have a variable meta-argument. At
runtime this may get instantiated to a rule head that belongs to the language part of the object program compiled.

Should the object part then be included with the compiled result? No, instead the rewrite relations collected by \texttt{rewrite} can be used to translate interpreter goals at runtime. A new clause may be added at the beginning of the interpreter which traps calls to rule heads that have already been compiled and maps these calls to the corresponding compiled procedure. For the \texttt{count/2} interpreter this clause would be:

\begin{verbatim}
count(Goal, C) :-
    '\#\$\$rewrite'(count(Goal,C), CompiledGoal), !,
    call(CompiledGoal).
\end{verbatim}

Now the rewrite relations should be included with the compiled result.

The final result of compilation is the collection of the following components:

- residues,
- Prolog part of the object program,
- interpreter modified with the extra clause, and
- the rewrite relations.

The components other than the residues may not always be required. The decision to include a component in whole or part requires further processing of the residue. The task involved is equivalent to finding dead code in a program. This is an independent problem in itself and may be handled separately.

Care is also required to avoid conflict between the names of predicates generated in the residues with the predicates of the other components composing the compiled program.
User Interface

The user interface provides the support for handling the input and output of a system. The capabilities of a system may be demonstrated without a good interface, but to translate those capabilities to a useful tool the user interface becomes an important issue in itself. Its design depends on the conditions the tool would be used in and the support provided from the environment. Unlike the other modules there can be no general purpose user interface suitable for all conditions.

The inputs required by the partial evaluator are:
- the interpreter
- the object program split in language and Prolog part, and
- meta-knowledge about the interpreter and the language.

The meta-knowledge guides the unfolding, rewrite rules, and extraction of rule head from the object programs.

The output consists of the components of compiled results discussed above.

The input interface of ProMiX has the following features:
- input is given in files.
- a collection of files may be classified as system
- "similar" interpreters can share meta-knowledge.

Providing inputs through files makes the task of specifying the individual input programs easier. Only the file(s) containing the interpreter, its meta-knowledge, and the object program have to be specified.

The meta-knowledge provides the information to guide the compilation. The information is related to the interpreter, the language it interprets, and the interpretation technique. Interpreters that interpret the same language in
the same way may share this information. This is done by classifying similar interpreters in the same class. The meta-knowledge is associated to the class.

When a program is distributed over several files it would be easier to group these files as system. The name of the system identifies the collection of files. This is useful for real-life programs. Such a system definition can be used by other program manipulation or analysis tools. In our environment it is used by a cross referencer, static analyzer, and a profiler. It is also used by utilities that extend Prolog’s capability of consulting files to consulting systems.

The sequence of events for compiling a program are:

- load all the input files in the database
- extract all the rule heads in the object program
- get the interpreter goal and position of meta-argument
- call peval\_program/2 with the appropriate parameters

The interpreter files should be loaded such that its information can be accessed by clause/2. The object program should be loaded such that it can be accessed by the predicates used for unfolding the rule heads. In the case of count/2 it is also clause/2.

To port this system to Prolog implementations that do not support clause/2 or limits the use of clause/2 some changes are required. This problem may be solved by loading the interpreter as a set of facts instead of a Prolog program.

Example:

```prolog
program(count(true,0), true).
program(count(A,B),C),
    (count(A,CA),
     count(B,CB),
```


C is CA + CB).

program(count(X,0),
        (not processed_above(X),
         sys(X),
         call(X)).

program(count(H,C),
        (predicate_property(H, interpreted),
         program(H,B),
         count(B,CB).
        C is CB+1).

The Prolog part in the object program should be treated similarly. Now by redefining the unfold_clause(X, Y) procedure as:

    unfold_clause(X, Y) :- program(X, Y).

the system would be ready to specialize interpreters given in the above form.

The modified system described above is portable across various Prolog systems. We prefer to use the version with clause/2 because it enables partial evaluation of programs while they are actually being used. Programs in the above form are not directly evaluable by the Prolog engine.

The output interface of ProMiX provides three options, the compiled result may be:

- displayed on the screen.
- written to a file, or
- bound to a variable.

The first option is good for testing and demonstrating the result of partial evaluation. For compiling file(s) the second option suits better. The compiled
result is put in a file and may be used again later. The third option enables postprocessing of the compiled result. The capabilities of the system may be extended by writing 'filters' that take this result and process it further. Removal of dead code or asserting the compiled program into Prolog database are examples of such processing.

5.8 Knowledge Base

The mix and rewrite modules of ProMiX are general purpose in that the knowledge about properties of predicates whether system defined or user-defined is not hard coded. They use predicates external to these modules to associate a property to a predicate. The criteria for controlling unfolding too is maintained externally as a set of rules. Similarly, knowledge about any specific interpreter is not built into the driver; it is provided to it by separate declarations and annotations.

The collection of such predicates, rules, and declarations, constitute the knowledge base module. This module customizes ProMiX to a particular application. In a way it is the variant component of the system. Its content varies upon the interpreter, the language it interprets, and to some extent the Prolog implementations. The other modules remain unchanged across applications.

The knowledge provided by this module may be classified in the following categories

- Property of Predicates
- Declarations for Unfolding
- Rewrite Rules
- Declarations for Driver
Most of these issues have been discussed along with the discussion of the other modules. The following text only compiles together the pieces spread across the previous sections.

5.8.1 Property of predicates

Naive partial evaluation of predicates with *side-effecting* or *extra-logical* behavior leads to the 'backward propagation' and 'multiple clauses' problems discussed in Section 5.5 and 5.5. The *mix* module is guided by predicates in the knowledge base that declare the system defined and user defined predicates that exhibit such properties. For example:

```prolog
side_effect(write(X)).
extra_logical(var(X)).
```

It is easy to write an exhaustive set of declarations for system defined primitives for a particular Prolog implementation. A user defined predicate can cause a side-effect either if it uses a side-effecting system predicate or another user defined predicate displaying side-effect. This property can therefore be extracted by statically analysing the program before partial evaluation. At present these predicates have to be declared too.

The *extra-logical* property, just like side-effect behavior, propagates too. A *user-defined* predicate using an extra-logical system primitive also displays extra-logical behavior. Given a set of system defined primitives that are extra-logical the user-defined extra-logical predicates may be determined by static analysis.

The *rewrite* module too depends on external knowledge. It needs to know which arguments of a meta-logical predicate, like *bagof/3*, can eventually be
used as predicate. This knowledge is provided by the meta_property/3 relations discussed in Section 5.6.

5.8.2 Declarations for unfolding

To unfold or not to unfold a goal is an important question for partial evaluation. As observed before there is no automatic way to detect whether unfolding a goal would lead to termination of partial evaluation. Some additional information specific to the program being partially evaluated is required to make this decision. Besides more than one residues for the same goal generated due to different unfolding decisions.

The decision to unfold or not is made by the mix module via the predicate should_unfold/1. It passes the Goal to this predicate to test if it may be unfolded. A set of rules, like:

should_unfold(Goal) ← True if Goal may be unfolded

are provided by the user that perform the required test. These rules may be application specific and can use knowledge about the program that is otherwise hard to extract automatically.

A set of guidelines for developing unfolding criteria for specializing interpreters are given in Section 5.9.

5.8.3 Rewrite rules

The rewrite module is responsible for translating the interpreter goals in the residue. To do that it should generate names for new predicates. If the residue is to be used by a programmer it would help if the names given to the new predicate make sense and are also unique. While the latter condition is easy
to satisfy by a machine, it is impossible to mechanically generate names that makes sense too.

Most researchers have implicitly used the principal functor of the object goal in the residual goal as the name of the translated predicate. This is the default scheme used by ProMiX. Considering the fact that the initial object program may at times be included with the compiled result, this scheme may violate the uniqueness requirement. Besides one may just wish to use a different name since the old functor name may not be suitable for the compiled predicate.

This can be done by providing new rules to rewrite predicates to overwrite the default definition. The predicate:

\[ \text{expand\_predicate(PredIn, PredOut)} \]

provides this information. It says that a predicate \( \text{PredIn} \) in the residue may be translated to \( \text{PredOut} \).

### 5.8.4 Declarations for driver

The \textit{driver} module drives the predicate \texttt{mix/2} to specialize interpreters. It creates interpreter goals for \texttt{mix/2} to partially evaluate such that the collection of their residues is the compiled result. To construct an interpreter goal one needs three things: which goals are interpreter goals, which argument is its meta-argument, and which object program atoms are used to compile the program. The first two questions are answered by a declaration with the following signature:

\[ \text{meta\_goal(Goal, MetaArg)}. \]
Here *Goal* is an interpreter goal. *Meta.Arg* is a variable which also occurs in meta-argument position of *Goal*.

Since an interpreter may be used to compile several programs, for ease of reference one may prefer to associate a name with an interpreter rather than referring to it by a goal. The mapping between interpreter name and the interpreter goal is declared by:

```
flavor(Flavor, IntGoal).
```

where *Flavor* is a name associated to an interpreter and *IntGoal* is its interpreter goal. Such declarations are also useful when specializing layered interpreters [83]. In these interpreters the task of interpretation is divided between more than one predicates arranged as a cascade. The processing of a predicate at the lower level is controlled by that at a higher level. A predicate processes the input string and invokes a predicate at a lower level with appropriate substring. This predicate chooses its future course of action depending on the results returned by the lower level predicate. An object program goal may therefore be interpreted by more than one interpreter predicate. In order to specialize layered interpreters all the interpreter predicates should be specialized with respect to the object program goals. In such a case the same flavor name may be associated to all the interpreter goals by appropriate *flavor/2* declaration. Corresponding *meta.goal/2* declaration for each of the interpreter goals would also be required.

The information about an interpreter provided above remains static with respect to the interpreter; it does not change for every object program. What changes with every object program are the set of atomic symbols used to create interpreter goals for compiling a program. Instead of having the user state all
the object program goals, in ProMiX the object program is input via a set of files. It is assumed that the object program consists of a set of Prolog terms called rules. A definite clause characteristic is also assumed. That is one atom of a rule is used to select that rule. Beyond that the driver does not assume the syntax of the rule and relies on a rule.head/1 declaration to extract the head atom from the rule. This declaration has the following signature:

\[
\text{rule.head}(R, H).
\]

where \( H \) is the atom used to select the rule \( R \). ProMiX uses this rule to extract all the head atoms in the program files. These atoms, it assumes, can potentially become meta-arguments of an interpreter goal.

### 5.9 How to Control Unfolding

This section gives tips on how to control unfolding when specializing interpreters. An interpreter can be viewed as performing two tasks: parsing and execution. Interpreters are specialized by partially evaluating the parsing activity while leaving the execution component as residue. We give a procedure for identifying goals that participate in the parsing process and present rules for unfolding these goals. This procedure may potentially be mechanized, thereby leading to automated compilation of the object program by specializing interpreters.

#### 5.9.1 Working of an interpreter

An interpreter performs two functions. First, it parses the incoming term to verify that it belongs to the language it interprets. Second, it simulates the
operational behaviour associated to any well-formed string of the language being interpreted. It does so by mapping the meaning associated to these strings to operations in the language in which the interpreter is written. We call these operations *semantic actions*.

The interpreters that we study interpret strings that are valid Prolog terms. We call these strings *rules*. They parse a rule by unifying it with skeleton of structures acceptable by the language, extracting its subterms, and parsing them recursively. Parsing terminates when an atomic symbol of the language is reached, though the interpretation may still continue. The semantic actions corresponding to an atomic symbol may *chain* the interpretation of other strings. We call such symbols *non-terminal*.

Not all atomic symbols trigger chaining. Symbols that do not chain interpretation we call *terminal*. The interpreters use *checking predicate* to test if the incoming symbols is terminal. In meta-interpreters cited in [82], system defined Prolog goals are terminal symbols. The test is performed by the checking predicate *sys/1*.

As these interpreters are written in Prolog the semantic actions are Prolog goals. These goals may be classified in two categories: system defined goals and user defined goals. Goals that are primitive builtins of the Prolog environment we call system defined goals. Predicates whose definition is given by the user are user defined goals.

### 5.9.2 An observation

From experience with specializing interpreters it may be observed that:
• The input to the partial evaluator is the interpreter. When the predicate \( miz(Goal, Residue) \) of Figure 5.5 is invoked, \( Goal \) would be instantiated to some goal in the body of the interpreter being specialized. Therefore, goals unfolded by a partial evaluator come from the body of the interpreter.

Object program rules are unfolded as a result of semantic actions that chain interpretation.

• The result of specializing interpreters is a rule-by-rule translation of the object program into Prolog. In [66, 86] every grammar rule is translated into an equivalent Prolog clause. Similarly in [33, 82, 86] every clause of a program is translated into another clause (or clauses) with enhanced features inherited from the meta-interpreter used for translation.

This is akin to compiling the object program into Prolog, where compilation of one rule of the object program is independent of other rules.

From these observations we derive the following guidelines for our unfolding criteria:

• The 'unfolding knowledge' required to do a source-to-source translation of an object program depends only on the interpreter being specialized, it is independent of the object program itself.

• Interpreter goals that lead to unfolding of other object program rules should not be unfolded.
5.9.3 Unfold criteria

The overhead of interpretation lies mainly in parsing the object program at run-time. Assume that an interpreter is partially evaluated such that all actions related to parsing a program are completely executed. The residue after parsing a rule of the object program would consist of semantic actions associated with parsing it and processing its atomic goals. Hence the residue for a rule would have the same execution behaviour as that associated to interpreting the rule, but would be free of the latter's parsing overhead.

Goals in the residue that are sufficiently instantiated for execution may be further (partially) executed during specialization. As these goals do not incur any interpretation overhead we consider their pre-execution analogous to compile-time optimizations in traditional compilation approaches.

When specializing interpreters our primary interest is to remove the parsing overhead. Optimizations by pre-executing semantic actions, though important, are secondary. We classify our unfolding criteria on the basis of the effects they produce.

Removing Parsing Overhead

Parsing is performed by matching the input structures to constructors of the language interpreted, then decomposing this structure to parse its components recursively. In practice we can extract the set of structures decomposed by an interpreter by scanning the interpreter. We can also mark a set of procedures that participate in parsing. Goals corresponding to this set of procedures we call meta-goals and arguments that carry the input term meta-arguments. Our first criterion for unfolding is that a meta-goal may be unfolded only if its
meta-argument is bound to a structure that is decomposed by the interpreter.

The criterion when translated to Prolog takes the form:

```prolog
should_unfold(Goal) :-
    meta_goal(Goal, MetaArg), structure_decomposed(MetaArg).
```

The predicates `meta_goal/2` and `structure_decomposed/1` provide information about the interpreter. The goal `meta_goal(Goal,MetaArg)` says `Goal` is a meta-goal with meta-argument `MetaArg`, and `structure_decomposed(MetaArg)` checks if `MetaArg` belongs to the set of structures decomposed by the interpreter being specialized. By unfolding only those instances of a meta-goal that carry a structure that is decomposed by the interpreter, we implicitly avoid the unfolding of clauses that process atomic goals.

For the vanilla meta-interpreter this knowledge would be:

```prolog
meta_goal(solve(Goal, _Cf), Goal).
structure_decomposed(true).
structure_decomposed([_GoalA, _GoalB]).
```

In an environment where there are several meta-interpreters, as that envisioned by [82], one may classify interpreters on the language they process. All interpreters processing the same language decompose the same set of structures. The `structure_decomposed` information can therefore be associated to a class of interpreters rather than an individual interpreter.

The next step after recognizing the meta-goals, meta-arguments, and the structures decomposed is to identify the imperative goals. These are goals that are mutually recursive with the meta-goals but are not meta-goals themselves.
that is, goals that appear in the recursion path from a meta-goal to a meta-goal. An imperative goal carries the meta-arguments for the recursive calls but does not itself process these arguments. These goals should be unfolded so as to lead the execution to the meta-goals. Hence our second rule of unfolding.

should_unfold(Goal) :- imperative(Goal).

The interpreters cited in partial evaluation experiences of other researchers [33, 82, 86] do not have imperative goals. Example of such interpreters may be found in Prolog text book [84].

Pre-execution of Atomic Symbols

The above rules refrain from unfolding clauses that process atomic symbols that may chain interpretation to other rules. But as noted before not all atomic symbols chain interpretation. It may thus be worthwhile to unfold the processing of those atomic symbols that terminate interpretation. Thus our third rule that unfolds such goals.

should_unfold(Goal) :-
    meta_goal(Goal, MetaArg), terminal_atom(MetaArg).

The goal terminal_atom(MetaArg) tests if MetaArg is a terminal atom. The test may be performed by the same predicates as that used by the interpreter for identifying the terminal symbols. This knowledge about the interpreter can be extracted easily by scanning it and can be customized for a class of interpreters. For example:

    terminal_atom(Term) :- system(Term).
is an extract from a Prolog meta-interpreter that uses system/1 to check if a goal is a system defined primitive.

Compile-time Optimizations

To pre-execute semantic actions that are sufficiently instantiated we fall back to an annotation scheme similar to that of [86]. We classify these goals as export, immediately evaluable, and partially evaluable. An export goal becomes a residue, whereas an immediately evaluable goal is evaluated completely using the underlying Prolog engine. Partially evaluable goals, as the name suggests, continue to be unfolded.

Annotating goals needs partial evaluation expertise or some trial-and-error experimentation. We have some useful hints that may make this task less tedious.

- A goal must fall in one of the three categories: export, immediately evaluable or partially evaluable. Thus goals for only two categories may be annotated, the third category being treated as default. Like Takeuchi & Furukawa [86], we assume a goal to be partially evaluable unless otherwise annotated. The fourth unfold rule thus is:

  should_unfold(Goal) :-
  not export(Goal), not immediate(Goal).

- The goals in question now do not participate in parsing. So as a first try one may mark all such goals as export and perform the compilation. Then later looking at the goals in the residue one may perform suitable annotations to get the most efficient form of compiled result.
• System defined goals, such as the arithmetic predicates, may either be completely evaluated or exported. The definition of these predicates is inaccessible, so they can not be unfolded.

Being builtins, these predicates are independent of the program being partially evaluated. We suggest these goals be treated separately by the partial evaluator. A database of executable/1 rules may be maintained as a global information about the system. These rules test if a system goal is sufficiently instantiated for execution. For instance:

\[
\text{executable(X is Y + Z)} \leftarrow \text{nonvar(Y)}, \text{nonvar(Z)}. \\
\text{executable(append(X, Y, Z)) \leftarrow complete_list(X)}. \\
\]

The predicate \textit{complete_list(X)} verifies if \(X\) is a complete list, as defined in [84].

• The partial evaluator of Figure 5.5 may be augmented by adding the following clauses.

\[
\text{mix(Goal, true)} : - \\
\text{immediate(Goal), !,} \\
\text{call(Goal).} \\
\text{mix(Goal, true)} : - \\
\text{executable(Goal), !,} \\
\text{call(Goal).} \\
\]

These clauses provide the support necessary for the classification of goals given above.
5.10 Background

The theory of partial evaluation originated from mathematics. It is based on Kleene's S-m-n Theorem [45] for recursive function languages presented in 1952. The concept was introduced to computer science by Futamura [35] in 1971. In this paper Futamura recognized the relation between interpretation and compilation by the means of partial evaluation. He projected the use of partial evaluation as a compiler, to develop compilers, and compiler-compilers. These projections are now called Futamura projections.

Other researchers who actively pursued partial evaluation in the 70's were A. Haraldsson from Sweden and A.P. Ershov from Russia. Haraldsson studied the problem in the context of Lisp while Ershov studied an Algol-like language. The sparks from their work triggered a fire of research activity on partial evaluation in the 80's. The concept has now been studied by researchers around the world. It is beyond the scope of this paper to survey all the efforts. Sestoft and Zamulin [78] have compiled a rather extensive bibliography of over 250 papers related to this subject. A less elaborate bibliography is published in [77]. The Workshop on Partial Evaluation and Mixed Computation held in Denmark in August 1987 and organized by D. Bjørner, A.P. Ershov, and N. Jones constitutes a landmark event for this subject. Papers presented during this workshop with relevance to logic programming appear in [9], other papers appear in [8]. The proceedings of Meta88: a workshop on meta-programming and logic programming [53] contains some more recent papers on partial evaluation.

Partial evaluation was introduced to logic programming by Komorowski [47]. Venken [93] was probably the first to report a partial evaluator for Prolog. In
this and a later paper [94] he raised the various problems that arise due to 'impurities' in Prolog. Several researchers have since reported partial evaluators for "full" Prolog [9]. It has been explored as a tool to generate efficient Prolog programs (in terms of logical inferences) by transforming semantically equivalent Prolog programs. Recently Lloyd and Sheperdson [54] have attempted at developing the formal foundation for partial evaluation.

The first Futamura projection [35] states how partial evaluation of an interpreter with respect to an object program effectively compiles the object program. This has been the guiding principle for a large segment of the logic programming community interested in partial evaluation. Gallagher, in 1983, observed that this property of partial evaluation can be used to reduce run-time overhead of meta-interpreters. His observations appeared later in 1986. Meanwhile Takeuchi & Furukawa independently came to the same conclusion in a 1985 ICOT technical report published later as [86]. Both the papers observe that partial evaluation of an interpreter with respect to an object program has the effect of compiling the object program into a new object program. The new program inherits the functionality of the meta-interpreter but not its overhead. Sterling & Beer [82] particularized this technique for expert systems. Pereira & Shieber [66] used this method to compile DCG programs to Prolog.

The use of partial evaluation as a compiler provides a very good optimization tool for Prolog meta-programs. Takeuchi & Furukawa [86] and Sterling & Beer [82] reported up to a factor of 40 improvement in execution time of compiled program over its interpreted counterpart. It is felt, therefore, that partial evaluation can bridge the dilemma between writing programs for machine or writing programs for humans. It promises to make meta-programming
a feasible technique for developing production quality expert systems.

The euphoria generated by these initial results raised the expectations of the Prolog community. Several researchers are now disillusioned since partial evaluation does not deliver the wonders that they thought it would [63, 92]. In our belief the conflict arises because the term "meta-programming overhead" has never been defined informally or formally. In our experience partial evaluation of interpreters removes only the "parsing activity" associated with executing the specific object program. The "semantic actions" are left for evaluation at run-time (see Section 5.9). If an interpreter spends most of its time in performing these semantic actions, the specialized program would not do any better.

Partial evaluation is just a sequence of carefully guided unfold transformations. To do any better transformations other than unfold are necessary. Owen [63] has proposed the use of fold transformations in the partial evaluator. The proposal has merit since it is known that the fold transformation is necessary to change the algorithmic complexity of any program. However, we do not advocate its inclusion in the partial evaluator. If required it should be performed on the residue by a separate 'filter'. In fact a generalized scheme of 'filtering' programs through transformers needs be worked out. This would ensure simplicity of code of individual transformers and also provide flexibility of juxtaposing more transformers at will.

Alternate transformation methods that generate results similar to those from partial evaluation of interpreters have been reported too. Neumann [60] derives a translator from an interpreter. The translator is then used to transform an object program. Louis & Vauclair [55] write a translator directly
instead of writing a meta-interpreter. It is our hypothesis that one should be able to generate the translators used by Louis & Vanclair [55] by partially evaluating the partial evaluator with respect to an interpreter. Similarly the translator-generator used by Neumann [60] should be derivable by partially evaluating the partial evaluator with itself. This hypothesis follows directly from Futamura's second and third projection.

Self-applicability is the most glamorous part of the Futamura projections. So far only a self-applicable partial evaluator for only "pure" Prolog has been reported by Fujita & Furukawa [33]. A self-applicable partial evaluator for "full" Prolog is required to test the hypothesis above. ProMiX provides a sound foundation for such an experiment. On the same note, a self-applicable partial evaluator for functional programming language was reported by Jones et. al. [44]

Curiously, the word mix is a name commonly used for partial evaluators. Its root can be traced to Ershov [29]. Ershov calls partial evaluation mixed computation. Presumably Jones et. al. [44] derived the name mix for their partial evaluator from it.
Chapter 6

Enhancement Tools

This chapter takes a closer look at programs developed by enhancing other programs. It splits the effect of enhancements into domain-independent and domain-dependent parts. A comparison of domain-independent changes across different programs reveals certain patterns. These patterns can neither be captured by algorithms nor by using program schema. It is argued that these patterns are not a coincidence but are a result of consciously applying certain programming techniques. These techniques, unlike algorithms, play a supporting role in a program. In a language such as Prolog, they provide conventions to propagate results of computations in a program. They can be abstracted out from the context of the specific computation performed.

This chapter classifies three types of programming techniques commonly used in Prolog programs and develops a prototype support for applying these techniques automatically to programs. This gives a very powerful mechanism to incorporate structured changes to programs in contrast to the mechanisms supported by current editors.

6.1 Domain-Independent Enhancements

Consider, for instance, the problem of counting the number of reductions required for resolving a Prolog goal. The program count/2 of Figure 5.1 solves this problem. It interprets a Prolog program and simultaneously counts the
number of reductions made in doing so. The program may be viewed as being modified from the Prolog meta-interpreter solve/1 of Figure 3.2. In a similar way the program length/3 of Figure 6.1 is modified from the program connected/2 of Figure 4.1. The latter program checks for the connectivity of a graph. It was used for the running examples in the previous chapter. The enhanced program length/3 checks for connectivity and also returns the length of the path between the two nodes.

A comparison of count/2 with solve/1 and length/3 with connected/2 reveals that though the modifications seem dissimilar the following comments hold for the two cases:

- the head predicates and the recursive goals are renamed,
- an argument is added to the predicate.
- the argument is used to 'return' the result of 'some' computation,
- the computation is performed at the end of a clause,
- the computation depends on the newly added variables and constants,
- the computation added succeeds once and only once, and
- the modification does not alter the state of previous variables.

This list of similarities is independent of the actual functionality of the initial program and also of the computation introduced due to the modification. The pattern is common not just to these two programs cited but a variety of programs. In fact it also captures the enhancements made in Chapter 2 to construct the derivation tree of Figure 2.4 from the syntax analyser of Figure 2.2.

A still more complex pattern may be seen in the changes required to generate programs count/2,3 and length/3,4 of Figure 6.2 from the programs solve/1
length(Start, End, 1) :-
    edge(Start, End).
length(Start, End, L) :-
    edge(Start, Hop),
    length(Hop, End, L1),
    L is L1+1.

Figure 6.1: Programs 'propagating context up'

length(Start, End, L) :-
    length(Start, End, 1, L).
length(Start, End, L, L).  
length(Start, End, Li, Lo) :-
    Lm is Li+1,
    edge(Start, Hop),
    length(Hop, End, Lm, Lo).

count(Goal, Count) :-
    count(Goal, 0, Count).

count(true, Ci, Ci).

count((G1,G2), Ci, Co) :-
    count(G1, Ci, Cm),
    count(G2, Cm, Co).

count(H, Ci, Co) :-
    Cm is Ci+1,
    unfold_clause(H, B),
    count(B, Cm, Co).

Figure 6.2: Programs using 'accumulators'
and connected/2 of Figures 3.2 and 4.1, respectively. A close analysis shows the following similarities:

- two new arguments are introduced,
- one argument is used for input,
- the other argument is used for output.
- in a clause handling the base case these arguments are unified.
- for recursive clauses some local variables are added,
- output of one recursive goal is propagated to another,
- the output from the last recursive goal is the output of the clause,
- the input to the first recursive goal is a function of the input argument and some constant.

This list of similarities too is independent of the structure of the initial programs or the computation introduced by the enhancement. This pattern is also common to a variety of programs. In fact it bears a strong resemblance to several programs given by Sterling and Shapiro [84] to demonstrate 'the use of accumulators'.

6.2 Programming Techniques

Prolog programs from text books, papers, and programs were studied to analyse the programming techniques used to propagate results of (partial) computations across different levels in (recursive) programs. The techniques commonly used may be classified in three types.

- propagate context up,
- propagate context down, and
- accumulator technique.
These techniques are implicitly used by most programmers, but there has been no reported attempt at characterizing their properties. The accumulator technique is the most explicitly stated technique in the literature. Sterling and Shapiro [84] introduce the use of 'accumulators' for 'bottom-up construction of lists'. Brna et. al. [14] use the names 'constructing-datastructures-in-the-clause-head' and 'constructing-datastructures-in-the-clause-body' to differentiate the position of computation. Sterling and Beer [82] use the phrases 'structural enhancement' and 'contextual enhancement' to identify different types of changes performed to interpreters. These correspond to propagating context up and propagating context down respectively.

A study of usage of these techniques indicate two common features.

- One or more variables of a procedure may be associated to a technique.

  These variables I call context. Not all procedures in the program may 'participate' in the technique.

- Computation added in the technique may use variables inside the context and also outside it. But the computation not considered part of the technique may use only variables outside the context.

The restriction in the second point imposes a functional view where output variables, if any, in the context are given as a function of input variables in context as well as variables outside it. It does not restrict variables outside the context from being used in the output mode. It only restricts them from being dependent on variables in the context.
6.2.1 Propagate context up

This technique uses context only in output mode. It is used to propagate information computed at deeper levels in an execution tree outside. The depth may be due to recursion or due to invocation of subprocedures.

The programs count/2 of Figure 5.1 and length/2 of Figure 6.1 demonstrate the propagation of context upwards in the execution. The third argument of program length/3 and the second argument of count/2 is used in the output mode. The result of computations from lower levels in the execution tree is propagated upwards. The program length/3 computes the length of a path by incrementing the length received from lower levels and propagating the new value up. The program count/2 uses the same technique for counting reductions.

6.2.2 Propagate context down

In this technique the context is used only in input mode. It propagates information down in an execution tree. New context may be generated using information from the incoming context and the information accessible within the scope of the computation.

The second argument of the program connected/2 and length/3 exemplify downward propagation of context.

6.2.3 Accumulator technique

This technique ‘intertwines’ the two context propagation techniques. It uses some context variables for input and others for output. The result from partial computation are propagated down the execution tree till the base case of
recursion is reached. At this point the incoming context is used to compute the outgoing context and the result propagated up.

The programs in Figure 6.2 demonstrate the use of accumulator technique. The third argument of length/4 is the partial length (corresponding to the initial segment of a the path). This value is propagated down. When the recursion terminates the length of the prefix of the end of the list is propagated up. Counting the number of reductions by using accumulator technique is more tricky. The second argument of count/3 is the input context. It corresponds to a partial count from the execution thus far. For conjunctive goals the count returned from left goal becomes partial count for the right goal.

The input and output context are collectively called accumulators. This technique is used in computing finite differences [64].

6.3 A Technique Interpreter

Program schemas [56] are conventionally used to abstract similarities between the control flows of programs. Since programs using the same technique may not have the same control, program schemas cannot capture their similarities. This may be seen by comparing the control flow of programs in the examples just cited.

A technique can be abstracted by extending a Prolog meta-interpreter with one or two extra arguments representing input and output context vectors. It is also interspersed with abstract predicates to be invoked at various important stages during the interpretation of an object program.

The program schema of Figure 6.3 encodes the programming knowledge pertinent to accumulator technique in a Prolog meta-interpreter. Abstract
acc(\text{Goal}, \hat{Z}) :-
    \theta(\hat{Y}),
    acc(\text{Goal}, \hat{Y}, \hat{Z}).

acc(\text{Head}, \hat{X}, \hat{Z}) :-
    \text{unfold\_clause(Head, Body)},
    \alpha(\text{Head}, \hat{X}, \hat{Y}),
    acc\_body(\text{Body}, \hat{Y}, \hat{Z}).

acc\_body(\text{',}'(\text{And,Then}), \hat{X}, \hat{Z}) :-
    !,
    acc\_body(\text{And}, \hat{X}, \hat{Y}_1),
    acc\_body(\text{Then}, \hat{Y}_1, \hat{Y}_2),
    \beta(\text{And,Then}, \hat{Y}_2, \hat{Z}).

acc\_body(\text{Goal}, \hat{X}, \hat{Z}) :-
    \delta(\text{Goal}),!
    call(\text{Goal}),
    \gamma(\text{Goal}, \hat{X}, \hat{Z}).

acc\_body(\text{Goal}, \hat{X}, \hat{Y}) :-
    acc(\text{Goal}, \hat{X}, \hat{Y}).

\delta(\text{Goal}, \hat{Y}, \hat{Z}).

Figure 6.3: An accumulator technique interpreter schema

Predicates are represented using Greek symbols such as \(\alpha, \beta\), etc. The symbols \(\hat{X}, \hat{Y}\), and \(\hat{Z}\) represent a vectors of input variables, local, and output abstract variables, respectively. The symbols are subscripted if a further subgrouping is required.

The technique interpreter is enhanced from the Prolog interpreter \texttt{solve/1} of Figure 3.2. The enhancements involved require adding two arguments to represent the input and output context. For example, in the predicate \texttt{acc/3} the second argument carries the input context and the third argument the output context.

Abstract predicates are used for performing computation required by the technique. Their execution is interleaved with the execution of the object pro-
gram. The predicate \( \theta \) initializes \( \bar{Y} \) the initial input accumulator for \( acc/3 \). It is associated as the input context for the object goal used to invoke the interpreter. The predicates \( \alpha, \beta, \gamma, \) and \( \delta \) have the following signature:

\[
\tau'( \text{ObjectInfo. } \bar{X}, \bar{Z} )
\]

where \( \bar{X} \) is input accumulator and \( \bar{Z} \) the output accumulator. Value for \( \bar{Z} \) is computed using \( \bar{X} \) and \( \text{ObjectInfo} \). The latter variable provides the object program environment in which the computation takes place. The predicates \( \gamma \) and \( \delta \) use the state of the object program goal interpreted immediately before calling them. The predicate \( \alpha \) abstracts the computation performed immediately after the \( neck \) of a clause. It uses information in the head of the clause for computation. Computations that use info from more than one goal in the clause body is abstracted by \( \beta \).

Collectively \( \alpha, \beta, \gamma, \) and \( \delta \) provide a mechanism to compute a new value of the accumulator after the \( neck \) of a clause and after every goal in a clause.

The abstract predicate \( \vartheta \) is different. While the other predicates abstract the computation on accumulators, \( \vartheta \) abstracts the interpretation of the object program itself. Typically \( \vartheta \) may be instantiated to test if its argument is a 'system predicate'. In general it provides an 'escape mechanism' to object program goals that do not participate in performing any computation on the accumulators. For instance, the goal \( clause/2 \) in the interpreter of Figure 3.2. The predicate \( \vartheta \) provides an effect analogous to the use of '{' and '}' in DCGs [66].

The propagate-context-up and propagate-context-down techniques are simpler than the accumulator technique. These techniques use the context to propagate information only in one direction. Interpreters for these techniques are very similar to that for the accumulator technique. They both use only one
abstract variable. They differ with each other in the operands used for abstract computation. When propagating context up the abstract predicates use context returned by subgoals. In contrast when propagating context down the abstract predicates operate on the incoming context and propagate the newly computed context down to the subgoals.

An Observation: It may be interesting to note that the interpreter for accumulator technique itself uses accumulator technique.

6.4 Incorporating Techniques in Programs

A technique interpreter schema, in essence, encodes the programming knowledge corresponding to a technique. This interpreter cannot be instantiated directly to get a meaningful program. Instead the technique it encodes may be incorporated into a program by partially evaluating it with a base program. Thus a technique interpreter encoding the use of accumulators when specialized with respect to programs enhances it with the encoded technique. Instead of partially evaluating an instance, the technique interpreter schema itself may be specialized.

It may be observed that the technique interpreter of Figure 6.3 reflects the computation model of Prolog when interpreting an object program. The behavior of the object program whether evaluated directly by Prolog or by the technique interpreter remains the same. Therefore, the result from compiling a program with respect to a technique interpreter retains the functionality of the program. The computations performed by the abstract predicates are over and above the interpretation task. The compiled program would perform these computations too. The result therefore is a Prolog program the same as the
\[ \tau(\text{Start, End, } \bar{Z}) : - \\
\theta(\bar{Y}), \\
\tau(\text{Start, End, } \bar{Y}, \bar{Z}). \]

\[ \tau(\text{Start, End, } \bar{X}, \bar{Z}) : - \\
\alpha_1(\text{Start, End, } \bar{X}, \bar{Y}), \\
\text{edge}(\text{Start, End}), \\
\delta_1(\text{Start, End, } \bar{Y}, \bar{Z}). \]

\[ \tau(\text{Start, End, } \bar{X}, \bar{Z}) : - \\
\alpha_2(\text{Start, End, } \bar{X}, \bar{Y}_1), \\
\text{edge}(\text{Start, Hop}), \\
\tau(\text{Hop, End, } \bar{Y}_1, \bar{Y}_2), \\
\delta_2(\text{Hop, End, } \bar{Y}_2, \bar{Z}). \]

Figure 6.4: Accumulator specialized for list handling

initial program with abstract computation for a technique induced in it.

The specialization of the interpreter schema also generates a schema. It consists of predicates and variables from the object program and along with abstract predicates and variables inherited from the interpreter. The schema in Figure 6.4 is the result of partially evaluating the goal \( \text{acc}(\text{connected}(S,E), \bar{Y}) \). given the accumulator technique interpreter of Figure 6.3 and the program \text{connected/2} of Figure 4.1. The specialization was done using ProMiX, a Prolog partial evaluation system described in the previous chapter.

The specialized result in Figure 6.4 was derived after rewriting the atom \( \text{acc}(\text{connected}(A,B), \bar{Z}) \) by \( \tau(A, B, \bar{Z}) \) and the atom \( \text{acc}(\text{connected}(A,B), \bar{X}, \bar{Z}) \) by \( \tau(A, B, \bar{X}, \bar{Z}) \) in the actual residue. The predicate name \( \tau \) in Figure 6.4 was arbitrarily chosen. It is renamed during instantiation. Abstract predicates in the residue are translated in a similar way too. These predicates are renamed (numbered) to counter the loss of information due to translation.
6.5 Generating Programs from Schema

The schema resulting from specializing a technique interpreter may be used to generate a family of programs. These programs use the same technique but may perform different computations on the context.

Instantiating a schema means associating some interpretation to the abstract predicates and functions and domain to the abstract variables in the schema. This in turn depends on the domain of application and the functionality desired. This cannot be automated without further knowledge about the desired applications.

An ideal scenario would be where this information is inferred directly from some incremental specification of the technique relative to the initial program. The structure and feasibility of such a scenario is in itself an independent topic of research best pursued in the domain of generating programs from specifications.

An immediate solution is to seek the definition of the abstract predicates directly from the user. This definition may be given as standard Prolog procedures. For instance consider the following definitions:

\[ \theta(0). \]
\[ \alpha_1(\_, \_, C, C). \]
\[ \alpha_2(\_, Ci, Co) :- \]
\[ \text{Co is Ci+1.} \]
\[ \delta_1(\_, \_, C, C). \]
\[ \delta_2(\_, \_, C, C). \]

The schema of Figure 6.4 along with these abstract predicates computes the length of a list. If the abstracted predicates are unfolded and the predicate \( r \) renamed to \text{length} the program \text{length/3} of Figure 6.2 is derived.
Consider renaming the predicate \( \tau \) to \textit{path} and defining the abstract predicates as follows:

\[ \theta([], []). \]
\[ \alpha_1(-, -, Y, Y). \]
\[ \delta_1(-, -, Y, [\text{Start}, \text{End}[Y]). \]
\[ \alpha_2(\text{Start}, -, Y, [\text{Start}[Y]). \]
\[ \delta_2(-, -, Y, Y). \]

The resulting program is:

\[
\text{path(Start, End, Path) :-}\]
\[ \text{path(Start, End, [], Path).} \]
\[
\text{path(Start, End, X, [Start.End[X]]) :- edge(Start, End).} \]
\[
\text{path(Start, End, X, [Start[Y]]) :-}\]
\[ \text{edge(Start, Hop),} \]
\[ \text{path(Hop, End, X, Y).} \]

This program constructs the path between the two nodes. The list containing the nodes on the path using accumulator technique. A partial path up to the current node is carried along during traversal of the graph. The list is completed at the end node and returned as the recursion unfolds.

### 6.6 Prototype Support

A prototype editor was built using ProMiX, a Prolog partial evaluation system, see Chapter 5. Interpreters for the three techniques mentioned in Section 6.2 were written; their code is given in Appendix D. The unfolding strategy for partial evaluation was augmented to suspend the evaluation of abstract predicates.
For the purpose of manipulating by machine, the schema was represented using abstract data typing techniques [38]. A vector of abstract variables was simulated by a term of arity equal to the length of the vector, whose \( i^{th} \) argument corresponds to the \( i^{th} \) element of the vector. Abstract predicates were differentiated by providing additional meta-knowledge. For instance:

\[
\text{abstract\_predicate(init\_acc(_))}.
\]

declared \text{init\_acc/1} as an abstract predicate:

The programs in Figure 5.1 and 6.1 may be generated by partially evaluating the technique interpreter for propagating context up with respect to the programs of Figures 3.2 and 4.1, respectively. Similarly, programs such as those in Figure 6.2 that use accumulator technique may be generated by compiling their corresponding skeleton programs with the accumulator technique interpreter of Figure 6.3.

The other interesting results were the generation of programs using 'difference-lists'. A difference list was represented as two arguments instead of a \( \cdot/2 \) structure. This translation maps the head and tail of a difference list to input and output accumulators. Programs were changed to incorporate computation using difference list by specializing them with the accumulator technique interpreter of Figure 6.2.

### 6.7 Related Works

The earliest tools developed to assist in the construction of programs were text editors and compilers. The view of programming as an exploratory process has led to a generation of interactive programming tools. The support
provided by these tools for incremental program development is of interest to our work. There are a large number of interactive programming environments. It is practically prohibitive to compare our work with every individual system. We therefore make comparisons with some representatives of these environments. The choice of representative is simply guided by the work we are most familiar with.

A wide collection of papers on interactive environments have been compiled in [4]. This collection is relatively old now, however, it is still a good representative of research in this direction today. A survey of software development environments is available in [25].

6.7.1 Structure-oriented editors

Structure oriented editors are also called language sensitive editors or syntax-directed editors. These editors give the user interactive tools to enter programs in terms of language constructs rather than character strings. Interacting directly with program constructs and relieves the user from the tedium of remembering details of the syntax. The contemporary structure editors allow structure as well text operations.

From the perspective of this thesis the structural editing facilities are of interest. The provision for creating and modifying a program by such editors can be summarized by the following extract from [88]:

Programs are created top-down by inserting new statements and expression at a cursor position within the skeleton of previously entered templates . . . Structural changes to the program are accomplished by removal and insertion of whole templates and phrases.
This quote talks about the Cornell Program Synthesizer. It is true for other structure editors too. Modifying a program by removing and inserting whole templates amounts to rewriting the whole text again. Although it guarantees structural integrity of the program at every step it is not very helpful for applying patterns of changes discussed in Chapter 6. The existing support provided by these editors is not adequate for incremental program development.

It is easy to code information about the semantics of a language along with its syntax [68]. This makes it possible to expand the capability of these editors to provide static-semantic analysis, program execution, and debugging [69]. It should also be possible to extend these editors to support the structured editing operations we indentify. The following quotation from [57] supports this hypothesis:

... many language-dependent operations, which are hard or impossible to achieve with text editors, become possible. For example, text editors are good at performing operations of the form “replace all occurrences of a letter a by the letter b”. They can not carry out an operation such as “replace all occurrence of the variable a that are part of an expression by variable b”. Because a structural editor embodies knowledge about such notions as variable and expression, it can carry out such operations.

6.7.2 Knowledge based specification assistant

The Knowledge-Based Specification Assistant (KBSA) [43] is designed to assist in the task of writing specification. It provides high-level editing assistance for
this task. The support is based on an incremental method of writing specification given in [30] and also discussed in Chapter 2. This method of development does not identify enhancement as an operation of incremental development of programs. Thus the repertoire of high-level commands provided by KBSA does not provide automated support for incorporating techniques. We suggest inclusion of such a command in their repertoire.

On the other hand, as pointed out in Chapter 2, in a practical scenario a program would be developed using incremental method proposed by [30] and stepwise enhancement. For the workbench to be useful it would be worthwhile borrowing commands from KBSA and adapting them for logic programs and Prolog.

6.7.3 The Programmer's Apprentice

The Programmer's Apprentice (PA) [73] is the flagship of projects for building intelligent assistant. The following quotation summarizes its aim [96]:

The intention is for the Programmer's Apprentice to act as a junior partner and critic, keeping track of the details and assisting with the easy parts of the programming process while the programmer focuses on the hard parts of the process.

To demonstrate the utility of an assistant approach Rich and Waters developed a Knowledge-Based editor in Emacs (KBEmacs). The paper [96] gives a session of developing programs using KBEmacs. The programmer communicates its needs using a series of high-level commands comprising of programming clichés. These are familiar terms associated to standard methods for dealing with a task. The use clichés enables the programmer to communicate with the system
in a high level rather than starting from the first principles. The program
development proceeds by interaction between the user and the assistant.

A closer look at the session with KBEmacs reveals that the program de-
velopment with the assistant has the same flavor as stepwise refinement [100].
Some clichés specify high-level structure of a program. The expansion of clichés
to a program is similar to refining a task in stepwise refinement. Besides there
are clichés that modify a program. The effect of modification is analogous to
making a change in decisions made by some previous refinement.

This similarity in the process of programming using PA and stepwise refine-
ment may indicate that PA is not very conducive to the software development
method presented in this thesis. This is however only a hypothesis. The truth
or falsity of the hypothesis is not immediately obvious from the published
material [72, 73, 95] and [96].

6.7.4 Programming knowledge representation

The heart of any intelligent programming tool is knowledge about pro-
gramming. Most automatic programming environments have concentrated on iden-
tifying and representing programming knowledge available in computer sci-
ence text books. They have focussed on encoding algorithms, data structure
selection decisions, operations on data structures, and program transforma-
tion [38, 71]. In essence the primary focus has been a program and not the
process of programming.

Unlike algorithms, programming techniques play a supporting role in pro-
grams. They are operational conventions built on top of the skeleton pro-
vided by a program. These conventions cannot be abstracted using program
schemas [56] without using a specific instance of the underlying program. They cannot, therefore, be represented using program schemas without losing their generality.

Since techniques are best understood by their operational effect they can be best represented by a formalism that captures the operation of a program. We find Prolog meta-interpreters [84] most suitable for this job. The programming knowledge for a technique can be abstracted by enhancing a Prolog meta-interpreter. Interestingly in KBSA the initial version of high-level editing commands were implemented as meta-programs [43]. They used PADDLE the meta-programming language of POPART for this purpose.

*Analysis of Technique Representation Scheme*: Rich has compiled a summary and comparison of various methods used to represent programming knowledge or clichés [72]. According to him a suitable knowledge representation scheme should meet the following desiderata:

**Expressiveness**: The representation must be able to express as many different kinds of clichés as possible.

A general statement about the expressiveness of interpreter schema to represent techniques is hard to make; it is adequate to represent the techniques we identify.

**Convenient combination**: The methods of combining clichés must be easy to implement, and the properties of combination should be evident from the properties of its parts.

Two or more techniques may be combined by composing their interpreters [83]. This is in contrast to the inability of combining general program schemas.
Semantic soundness: The representation must be based on a mathematical foundation that allows correctness conditions to be stated.

The representation is semantically sound since it uses Prolog for representation. It should be possible to develop a proof schema for its instances. This can be done by extending the method used by Misra [58] to develop proof schemas for programming principles.

Machine manipulability: It must be possible to manipulate the representation effectively using computer tools.

A technique interpreter can be manipulated easily and efficiently by existing Prolog systems. This is evident from the success of partial evaluation technique for specializing interpreters systems, see Chapter 5.

Programming-language independence: The representation should not be tied to the syntax of any particular programming language.

The representation strongly depends on the meta-programming capabilities of Prolog and hence is not language independent.

6.7.5 Program schemas

A program schema is a program in which the constants and terms are replaced with uninterpreted functions and variables do not have any domain associated to them. Instantiating a schema implies associating an interpretation the uninterpreted functions and domain to the schema variables. A formal definition of logic program schemas is given in [102]. An excellent survey of various types of program schemas and their properties may be found in Manna [56].
The technique interpreter schema and the schema generated after specializing the technique interpreter differ from the definition of schema above. These schemas are partially instantiated in that some of its terms and constants are already interpreted. Such partially instantiated schemas are used by Misra to represent programming principles [58].

Program schemas have not been used extensively in the logic programming community. Brna et. al. [14] have suggested the use of program schemas to represent techniques. As observed in a previous subsection program schemas are not adequate to represent techniques. They can only be used to represent families of program with the same flow of control. O'Keefe gives a set of three program schemas to compute functions of lists along with examples of how these may be used to generate programs [62]. Two of these schemas are called: “the tower method” and “the linear method”. These schemas may be generate in our system by specializing the propagate-down context and accumulator technique interpreter schema, respectively, with a program that traverses lists.

6.7.6 Translation of DCG to Prolog

Pereira and Shieber [66] translate a DCG Program by partially evaluating a DCG interpreter. The result of their translation is a Prolog program similar to the DCG program, but with two extra arguments added to each procedure. One argument is used to carry the sequence of input tokens. A procedure parses some prefix of the input sequence and returns the remaining sequence in the second argument.

Results similar to that due to translating DCG programs to Prolog can be achieved by specializing the technique interpreter schema with respect to DCG
program. To get these results one has to modify the accumulator technique interpreter of Figure 6.2 to interpret the 'TokenList' and '{ '}' escape mechanism of DCG. In a way the method of incorporating accumulator technique generalizes the method and effect of translating DCG programs to Prolog.

6.7.7 Superimposition

The use of transformational methods in algorithm design is also used to help tackle the complexity of designing concurrent and distributed algorithms. The method of superimposition is of special interest to this thesis [12, 19]. Using this technique:

"... one imposes an additional control activity achieving one concern on top of some basic activity, achieving another concern" [12].

For example superimposition can be used to introduce the control activity necessary for detecting distributed termination into an set of concurrent processes developed independently. It is done by developing an algorithm that detects termination of an underlying program. This is achieved by providing some additional control communications and read-only access to the local states of the underlying program, not vice-versa. This algorithm when superimposed on any program imposes the control activity for detecting distributed termination. The superimposed control does not interfere with the existing control.

Introducing programming techniques is very similar to superimposition. A technique is developed as an independent algorithm. It introduces additional variables for propagating (communicating) information and has read-only access to the variables of the underlying program. The difference is that a
technique does not impose a control activity but introduces new computation around the control provided by the underlying program.

Interestingly superimposition has been included as a program composition primitive in UNITY, a programming language for designing concurrent programs [58]. Considering the correspondence between incorporating techniques and superimposition this is very heartening. It is our hope that someday the mechanism to incorporate techniques would become a widespread programming language construct.
Chapter 7

Program Composition

Sibling programs, as discussed in Chapter 1, are programs that are extensions of the same parent program. These programs have similar control structure. There is a correspondence between the syntactic elements of these programs inherited from their common parent.

This chapter presents an algorithm for composing pairs of sibling programs. The programs to be composed and the resulting program is specified by a join specification. The join specification along with the knowledge of correspondence between syntactic elements is used to compose the given programs.

The algorithm presented in this chapter uses the knowledge of similarity between structure of sibling programs to compose programs. In an environment supporting automatic enhancement, the enhancement relationships can be saved during program development. This knowledge will reduce the need for user interaction, leaving with the user with the task of identifying the programs to be composed. The algorithm is completely automatic for its domain of application.

7.1 Join Specifications

Composition is an operation over two programs resulting into a third program. The operation is specified using a join specification which is an expression of the following type:
\[ T \subseteq R \times B. \]

where \( T, R, \) and \( B \) are atomic formulae. The atoms \( R \) and \( B \) specify the procedures to be composed; they are called \textit{join operands}. These procedures should be siblings; their definition is assumed to exist in the program database. The atom \( T \) specifies the result of the composition. It is called the \textit{join target}.

Following is an example of a join specification:

\[ \text{simple.length}(S, E, L) \subseteq \text{simply.connected}(S, E) \times \text{length}(S, E, L). \]

The atom \( \text{simply.connected}(S, E) \) refers to the program in Figure 4.3 and \( \text{length}(S, E, L) \) to that in Figure 6.2. These programs are sibling procedures as they are extensions of the program \textit{connected} of Figure 4.2. The expression above specifies that the procedure \( \text{simply.connected}/2 \) and \( \text{length}/3 \) be composed to generate a procedure \( \text{simple.length}/3 \). The composition should be performed such that the first and second arguments of \( \text{simply.connected}/2 \) are the same as the corresponding arguments of \( \text{length}/3 \). These also form the arguments of the join target.

### 7.2 Composing Corresponding Clauses

The basic operation in our algorithm is composing two clauses. The operation only makes sense for corresponding clauses, that is clauses that are extensions of the same clause. The process of composing clauses may be summarized as:

- compose the heads of the clauses
- locate the corresponding pairs of inherited goals in the clause bodies
- compose these corresponding pairs of goals
The composed clause consists of the result of composing all the inherited goals and adding all the non-inherited goals of the two clauses. The order of the goals is defined by the order of goals in the input clauses and the operand order.

The composition algorithm is summed up by the program fragment below. In order to compose the bodies, it is necessary to locate inherited goals. The parent clause is carried along as an argument for this reason.

\% compose_clauses(+ParentClause, +Clause1, +Clause2, -Result)

compose_clauses((Head0 :- Body0), (Head1 :- Body1),

    (Head2 :- Body2), (JHead :- NBody)) :-

    compose_head(Head1, Head2, JHead),

    compose_body(Body0, Body1, Body2, JBody),

    body_to_normal_form(JBody, NBody).

The body_to_normal_form(JBody, NBody) flattens nested conjunctions and removes redundant true goals from JBody. Composing the head of the clause comes directly from the join specification.

compose_head(Head1, Head2, JHead) :- JHead \leftarrow Head1 \Join Head2.

Composition of clause bodies uses the body of the parent clause as a guiding skeleton to traverse the operand bodies. The program fragment in Figure 7.1 describes the algorithm. It operates on the premise that all goals appearing in the parent body appear in the operand body either in the same form or as an enhancement or modulant of the parent goal.

Composing inherited goals is easily described. If the inherited goals had been modified due to any enhancement, modulation or mutation they are
% compose_body(+ParentBody, +Body1, +Body2, -JoinedBody)
compose_body(true, Body1, Body2, (Body1, Body2)).

compose_body(Body0, Body1, Body2,
             (Prefix1, Prefix2, ComposedGoal, RestJoin)) :-
    /* get the first goal of parent as Goal0 */
    split(Body0, Goal0, Rest0),

    /* Align the two bodies with respect to the parent goal */
    align_inherited_goal(Goal0, Body1, Prefix1, Goal1, Rest1),
    align_inherited_goal(Goal0, Body2, Prefix2, Goal2, Rest2),

    /* Compose the inherited goals */
    compose_inherited_predicate(Goal0, Goal1, Goal2, ComposedGoal),

    /* Join the Rest of the bodies, recursively */
    compose_body(Rest0, Rest1, Rest2, RestJoin).

Figure 7.1: Compose bodies of corresponding clauses

composed with respect to a join specification existing in the system. Otherwise
they are simply unified.

% compose_inherited_predicates(+Parent, +Operand1, +Operand2, -Result)

compose_inherited_predicates(Parent, Goal1, Goal2, Composed) :-
    Parent ⊃ Goal1,
    Parent ⊃ Goal2,
    Composed ⊑ Goal1 ∼ Goal2, !.

    /* goal merge */

compose_inherited_predicates(Parent, Goal1, Goal2, Goal1) :-
    Goal1 = Goal2.

The procedure align_inherited_goal/6 searches the body of a clause for the
first goal that corresponds to a given parent goal. It uses the position of the
goal to split the body into three parts: Prefix - the sequence of goal preceding
the inherited goal, Goal - the inherited goal, and Rest - remaining goals after the inherited goals. An empty prefix or rest sequence is returned as true.

The composition algorithm above assumes conjunction as the only control construct in the program. It treats meta-predicates such as not, disjunction, and all solutions predicates as atomic goals. A meta-predicate appearing in the sibling programs may be inherited from their common parent. It is possible that its meta-argument be enhanced as a result of the extension. To generate correct result from composition the meta-arguments of two inherited meta-predicates should be composed too. On the contrary if a meta-predicate is not inherited but is added due to extension it can simply be treated as an atomic goal.

The correct composition algorithm may be generated very easily by step-wise enhancement itself. The control of this program is provided by a program that traverses a Prolog clause such that it visits every symbol that can potentially be invoked as a predicate. This requires decomposing the structure of a meta-logical predicate and recursively traversing its meta-arguments. This program may then be enhanced to simultaneously carry two child clauses corresponding to the operand clauses. The traversal of the operand clause would be guided by the traversal of the parent clause thus going from one inherited goal in an operand clause to the next. The traversal of second-order predicates such as setof/3 in the parent’s body may be matched with the corresponding second-order predicates in the child clauses. These goals may then be composed recursively.
7.3 Compose Sibling Programs

Given the algorithm for composing corresponding clauses, composing sibling programs is very simple. It requires composing all pairs of corresponding clauses.

The composition of two sibling procedures $R$ and $B$ with respect to a given join specification $T \Leftarrow R \bowtie B$ is the set of clauses obtained by the composition of all pairs of corresponding clauses of the two procedures with respect to the join specification. The difficulty lies in identifying sibling procedures and pairing the corresponding clauses. Given two arbitrary program it may generally not be possible to say if there exists a parent-child relationship between them. Therefore, it would not be possible to match the inherited components. It calls for some additional information. We provide this information by giving the extension relationships between a parent and a child program. That two procedures are siblings can be determined from the knowledge of their lineage. Any two procedures that have some common ancestor from which they can be enhanced are siblings.

In our system the enhancement relationships, or $\geq$, between procedures is specified by the user. These relationships could be automatically collected or inferred in a program development environment with a ‘structure editor’ that has enhancement as a primitive operation.

Considering procedures as a sequence of clauses, as in Prolog, pairing of corresponding clauses becomes trivial, having once identified sibling procedures and assuming enhancements do not disturb clause order. This implies that the $n^{th}$ clause of a child procedure is enhanced from the $n^{th}$ clause of the parent procedure. In the context of Prolog’s first-clause-first execution model
this assumption is not too restrictive. Thus clauses at the same position in the sequence of clauses of two sibling procedures are corresponding clauses.

**Examples of composition**

Consider the procedures `simply_connected` of Figure 4.3 and `length` of Figure 6.2. These two programs are clearly sibling procedures as they are extensions of the program `connected` of Figure 4.2. The composition of these two procedures with respect to the join specifications:

\[
\text{simple}_{\text{length}}(S, E, L) \leftarrow \text{simply}_{\text{connected}}(S, E) \bowtie \text{length}(S, E, L).
\]

\[
\text{simple}_{\text{length}}(S, E, I, O) \leftarrow \text{simply}_{\text{connected}}_{\text{x}}(S, E) \bowtie \text{length}(S, E, I, O).
\]

and the enhancement relationships:

\[
\text{connected}(S, E) \supseteq \text{simply}_{\text{connected}}(S, E).
\]

\[
\text{connected}_{\text{x}}(S, E) \supseteq \text{simply}_{\text{connected}}(S, E, V).
\]

\[
\text{connected}(S, E) \supseteq \text{length}(S, E, I_1).
\]

\[
\text{connected}_{\text{x}}(S, E) \supseteq \text{length}(S, E, \text{In}, \text{Out}).
\]

is the program:

\[
\text{simple}_{\text{length}}(\text{Start}, \text{End}, L) :-
\]

\[
\text{simple}_{\text{length}}(\text{Start}, \text{End}, \text{L}, \text{L}, \text{Start}).
\]

\[
\text{simple}_{\text{length}}(\text{Start}, \text{End}, \text{L}, \text{Visited}).
\]

\[
\text{simple}_{\text{length}}(\text{Start}, \text{End}, \text{Li}, \text{Lo}, \text{Visited}) :-
\]
Lm is Li+1,
edge(Start, Hop),
not member(Hop, Visited),
simple_length(Hop, End, Lm, Lo, [Hop|Visited]).

This program has the composite functionality of \textit{simply\_connected}/2, 3 and \textit{length}/2, 3. The former program checks for connectivity in a directed graph. It avoids cycles in the graph. The latter program computes the length of the path between two nodes in a graph. This program does not terminate if it enters a path containing cycle. The program resulting from their composition inherits behavior from both the programs. It computes the length of a path and also avoids cycles. It therefore returns the length of a simple path between any two nodes.

Example of composing meta-interpreters

The idea of enhancement and composition was triggered from the problem of composing Prolog meta-interpreters raised in [82]. It is therefore natural to demonstrate the use of our algorithm in composing Prolog meta-interpreters.

The following meta-interpreter is an extension of the vanilla meta-interpreter of Figure 3.2.

\begin{verbatim}
proof(true,fact).
proof((A,B),(PA,PB)) :-
    proof(A,PA),
    proof(B,PB).
proof(A,sys) :-
\end{verbatim}
sys(A), call(A).

proof(A,A-PB) :-
    unfold_clause(A,B).
    proof(B,PB).

It has been extended to construct the proof tree of the reductions made to prove a goal. We call this program \textit{proof}. The extension relationship between vanilla and proof is:

\[
\text{solve}(\text{Goal}) \supseteq \text{proof}(\text{Goal, Proof}).
\]

A Prolog meta-interpreter that executes a Prolog goal such that it performs only those computation whose search tree does not exceed a depth is called \textit{depth bounded} meta-interpreter or simply \textit{depth}. The bound on the depth of execution may be specified by the predicate \texttt{depth_bound/1}.

\[
\text{depth}(\text{Goal}) :- \text{depth_bound(D)}, \text{depth}(\text{Goal}, D).
\]

\[
\text{depth}(\text{true}, D).
\]

\[
\text{depth}((A,B), D) :-
    \text{depth}(A, D),
    \text{depth}(B, D).
\]

\[
\text{depth}(A, D) :-
    \text{sys}(A), \text{call}(A).
\]

\[
\text{depth}(A, D) :-
    D > 0,
    D1 \text{ is } D-1,
    \text{unfold_clause}(A,B),
    \text{depth}(B, D1).
\]
This program can be generated by a sequence of modulation and extension of the vanilla interpreter. This program may also be created by incorporating the technique propagate-context-down in the vanilla interpreter. The modulation and enhancement sequence from vanilla to depth goes through the following intermediate program.

```
solve(Goal) :-
solve_x(Goal).
solve_x(true).
solve_x([(A,B)]) :-
solve_x(A),
solve_x(B).
solve_x(A) :-
sys(A), call(A).
solve_x(A) :-
unfold_clause(A,B),
solve_x(B).
```

The modulation of the interpreter creates a clause that may used to initialize depth. This modulant is useful for other meta-interpreters that need some initializations before propagating some context down in the program. This meta-interpreter may be called an initializing modulant of vanilla. The extension relationship between the initializing modulant and the depth interpreter is as follows:

```
solve(Goal) ⪰ depth(Goal).
solve_x(Goal) ⪰ depth(Goal, Depth).
```
The problem raised in [82] was the need for combining meta-interpreters so as to merge their functionality. In this case we would like to compose the proof and depth interpreters to create a meta-interpreter that performs depth bounded execution of a Prolog query and constructs the proof tree for a successful computation. In order to be composable the two interpreters should be siblings, that is they should be extensions of the same interpreter. The proof and depth interpreters are extensions of different interpreters. This violates the requirement for the composition algorithm.

One may observe that the two parent meta-interpreters are actually modulations of each other. This knowledge can be used to overcome the problem. The proof meta-interpreter may be brought to have the same structure as the depth interpreter by modulating it. The modulations performed on vanilla to derive the initializing modulant may be replayed on proof resulting in the program:

\[
\text{proof}(\text{Goal, Proof}) \leftarrow \text{proof}_x(\text{Goal, Proof}).
\]

\[
\text{proof}_x(\text{true, fact}).
\]

\[
\text{proof}_x((A,B), (PA, PB)) \leftarrow
\phantom{\text{proof}_x(A, PA),}
\phantom{\text{proof}_x(B, PB).}
\]

\[
\text{proof}_x(A, sys) \leftarrow
\phantom{\text{sys}(A), call(A).}
\phantom{\text{proof}_x(A, A-PB) : -}
\phantom{\text{unfold_clause}(A, B),}
\phantom{\text{proof}_x(B, PB).}
\]

This program may be called the modulated proof meta-interpreter. It has
the following extension relationship with the initializing modulant of vanilla:

\[ \text{solve}(\text{Goal}) \supseteq \text{proof}(\text{Goal}, \text{Proof}). \]
\[ \text{solve}_x(\text{Goal}) \supseteq \text{proof}_x(\text{Goal}, \text{Proof}). \]

The modulated proof meta-interpreter and the depth meta-interpreter are extensions of the same program, namely the initializing modulant. These programs are therefore siblings. These programs when composed using the following join specifications:

\[ \text{pd}(\text{Goal}, \text{Proof}) \Leftarrow \text{proof}(\text{Goal}, \text{Proof}) \bowtie \text{depth}(\text{Goal}). \]
\[ \text{proof}_\text{depth}(\text{Goal}, \text{Proof}, \text{Depth}) \Leftarrow \text{proof}_x(\text{Goal}, \text{Proof}) \bowtie \text{depth}(\text{Goal}, \text{Depth}). \]

result into the program.

\[ \text{pd}(\text{Goal}, \text{Proof}) :- \]
\[ \text{depth}_\text{bound}(\text{D}), \text{proof}_\text{depth}(\text{Goal}, \text{Proof}, \text{D}). \]
\[ \text{proof}_\text{depth}(\text{true}, \text{fact}, \text{D}). \]
\[ \text{proof}_\text{depth}((\text{A, B}), (\text{PA, PB}), \text{D}) :- \]
\[ \text{proof}_\text{depth}(\text{A, PA, D}), \text{proof}_\text{depth}(\text{B, PB, D}). \]
\[ \text{proof}_\text{depth}(\text{A, sys, D}) :: \text{sys}(\text{A}), \text{call}(\text{A}). \]
\[ \text{proof}_\text{depth}(\text{A, A-PB, D}) :: \]
\[ \text{D} > 0, \text{D1 is D-1}, \]
\[ \text{unfold}_\text{clause}(\text{A, B}), \]
\[ \text{proof}_\text{depth}(\text{B, PB, D1}). \]

This program has the composite behavior of proof and depth meta-interpreter, as desired by [82]
7.4 Engineering Issues

A practical environment for software composition needs several issues to be resolved. The problem of finding if two given programs are related or identifying corresponding clauses is not straightforward. We address these and related problems in this section. The solutions that we suggest require a combination of providing meta-knowledge about the program, programming convention, and additional support by the underlying Prolog system.

- **Automatic generation of join specifications**: Programs being composed may be made up of several procedures. In our system we provide $S$ the set of all predicates of the parent program that are modified to derive the new programs. The enhancement and modulation relationship is provided as meta-knowledge. Join specifications are generated for all enhancements and modulations of all predicates that belong to the set $S$.

- **Automatic generation of new procedure names**: Generating new procedure names using `gensym` is usually not a good idea. We suggest following some naming convention such that new names may be derived from the names of the predicates being composed.

- **Finding corresponding clauses**: Our present system uses clause order. We manually modulate programs to bring them to the same modular form. We believe this could be done by the machine.

- **Pairing Inherited Goals and Arguments**: Our present system depends on goal order for matching goals and relies on meta-knowledge for the arguments.

- **Composing Mutants**: We annotate mutant clauses. While composing we query the user about what needs to be done at the equivalent stage for the mutant clauses. We recommend having some default rules to reduce this in-
teraction.

- *Handling Large Systems*: Real-life programs have a large amount of baggage besides the control structure. When composing a large program the whole program may not be relevant for composition. This situation arises when composing various programs to generate a Prolog Tracer, see Chapter 8. We suggest classifying the system in composable and non-composable parts. The composition algorithm may concentrate only on the parts to be composed.

- *Retaining Symbolic Names for the Variables*: To retain actual symbolic names used of the source program in the composed program requires support from the underlying Prolog system.

### 7.5 Related Works

The paper [50] presents a family of methods for composing programs. The methods are based on *clausal-join*, a specialized sequence of unfold and fold transformations [87] to merge two clauses. Clausal-join, in conjunction with varying strategies for selecting clauses from two programs resulted in a family of methods for generating new programs from existing ones.

The algorithm for composing corresponding clauses described in this chapter and also in [83] is a modified form of clausal-join. The former is a ternary operation defined only on two clauses having a strict extension relationships with a third clause. On the other hand clausal-join is an operation defined on two arbitrary clauses with no restriction on their lineage.

In [50] we defined a composition algorithm called *1-1 join*. This algorithm composed programs by performing clausal-join of corresponding clauses. The algorithm was too weak for the purpose of composing Prolog meta-interpreters.
The weakness of 1-1 join was tracked to a limitation of clausal-join. Clausal-join could not compose inherited goals that had local variables in their arguments. Local variables appearing in the clauses from different procedures, though inherited from the same parent clause, have different identity. They remain unaffected by the unifications of variables in the head of a clause. This led to the creation of a special algorithm meta-join for composing meta-interpreters.

The algorithm for composing corresponding clauses does not have this limitation of clausal-join. The problem is by-passed by using the extension relationships to identify and unify the corresponding arguments of corresponding inherited goals. This is implicitly done by the procedure `compose_inherited_predicates/4` using the conjunctive goals `(Parent ⊨ Goal1, Parent ⊨ Goal2)`.

Interest in merging two or more programs has been prevalent in computer science for a long time. It has been studied by researchers in program transformation paradigm to improve efficiency of programs. It has also been studied by researchers in software engineering to merge different versions of the same program. Our interest in composition aligns with work in the latter category while we use techniques studied and developed for program transformation.

### 7.5.1 Merging multiple traversal of data

If two programs that traverse the same data structure are composed the result is a program that performs the computations of both the programs in a single traversal. Methods that produce analogous results have been studied for the sake of creating efficient programs from inefficient ones.

Loop jamming, an optimization technique used in compiler construction to merge adjacent loops in a procedural language program [1]. It merges only
those loops that have exactly the same iteration conditions. Loops having
the same iteration condition may be considered as extensions of the same
empty loop. Thus the restrictions imposed by loop jamming are similar to
that imposed by our composition algorithm.

The transformation systems of Burstall and Darlington [15] and their fol-
lowers are geared towards producing efficient programs starting from an initial
program. They start with a specification that is executable in the same domain
as the transformation result. The system transforms this specification into an
efficient program by applying a sequence of simple transformation rules se-
lected by the user. The transformation system may also be used to merge
multiple traversal of data into one.

Bird and Pettorossi in their respective papers [7] and [67] demonstrate pro-
gram transformation techniques and strategies to “eliminate multiple travers-
sals” in the functional programming domain. Their aim, like ours, is to merge
the functions that independently traverse the same data structures, into a
single function that traverses the data structure only once but performs the
computation for both functions. Bird uses lazy evaluation techniques for the
specific purpose of merging multiple traversals for functional programs. Pe-
ttorossi’s “tupling strategy” is designed specially for this purpose. Turchin’s
supercompiler also “merges multiple passes” [90]. In the logic programming
paradigm Furukawa and Ueda perform “process fusion” of Flat Concurrent
Prolog programs for similar purposes [34].

The works cited above have a much broader scope than ours. The difference
in scope is largely due to differences in their intended use. Our composition
algorithm is primarily aimed at filling the needs of stepwise enhancement. For
this specialized domain of application it is better suited than the generalized transformations. It requires very little human interaction since the necessary transformation sequences are pre-selected.

There is also a difference between our approach to composition with that taken by the transformational systems. These systems do not utilize knowledge of any syntactic relationships between programs and rely totally on the semantic relationship. We split the task of program composition into the problem of associating syntactic relationship and that of performing the composition. We consider this splitting a significant contribution of our work. Crystallizing the questions related to syntactic relationship provides a clearer target to aim at for fully automated program composition. We are not aware of any other work that takes our approach in any programming paradigm.

7.5.2 Merging different versions of a program

In a multi-programmer software project a system may be modified concurrently by more than one person. This leads to the creation of multiple versions of the same program. These versions have to be merged at a later stage. The merging of concurrent update is an important issue in programming environments. The Source Code Control System (sccs) [74] and Revision Control Systems (rcs) [89] are two popular version control programs available on UNIX. These systems support features for merging independent updates to a program. They treat a program as a sequence of text strings and perform the merge without any regard to the syntax or semantics of the programming language.

In contrast experiments with merging programs using syntactic or semantic knowledge have also been reported. Berzins gives a partial semantic rule
to "merge extensions" for applicative programs [6], Horwitz et. al. give an algorithm for "integrating non-interfering versions" of simple procedural programs [39]. The Knowledge Based Specification Assistant provides support for "combining parallel elaborations" of specifications written in the language Gist [43].

In the following discussion I use the word enhancement to mean extension, versions, or elaborations in the context of the work [6], [39], and [43] respectively. These works and our composition algorithm are similar in that they all merge different enhancements performed to the same program and that they all use the syntax and semantics of a language for the purpose. Besides differing on the language each supports, these works mutually differ on what they mean by an enhancement, the method they employ for performing the merge, and the result expected after the merge.

Comparison with Berzins work [6]: In his view enhancement is a change that "extends the domain of partial function without altering any of the initially defined values". He uses a combination of rules that merge the syntactic domain and the semantic domain of programs to merge enhancements in two programs. It explicitly disallows merging of changes that alter the number of arguments of a function and hence is not suitable for our view of enhancement.

Comparison with Horwitz et. al's work [39]: They treat any modification to a program as an enhancement. They use program dependence graphs to represent a program [40]. The modifications performed in a child program with respect to its parent are identified by comparing suitable slices of the programs. Two programs enhanced from a common program are merged by merging their program dependence graphs. The merge is guided by the knowledge of program
modifications identified by program slicing.

This work is very flexible and promising. It would be interesting to explore it can be extended to logic programming and Prolog.

*Comparison with support in KBSA [43]:* Enhancements in the framework of KBSA are changes performed using the high-level editing commands it provides. Parallel changes performed on the same specifications are performed by replaying the evolutionary transformations used for one specification on the other [31]. This system can not perform compose programs with the kind of enhancements discussed in this thesis. The reason being that such enhancements are not supported by their repertoire. However this system can potentially be expanded to suit to our needs. This may be done by expanding the set of evolutionary commands as per discussion in Chapter 6.
Chapter 8

Developing a Prolog Tracer by Stepwise Enhancement

A Prolog tracer helps in tracing a Prolog program. It is, in essence, an interpreter of Prolog with provision to inspect or modify the program's state. Writing a Prolog tracer is not a trivial programming exercise. The complexity arises due to two reasons. The first reason is implicit in the very nature of the programming task. It is an exercise in meta-programming which usually defies intuition. The second reason is an offshoot of the first. A Prolog tracer is expected to have a long repertoire of features. Developing a tracer equipped with all the features at the same time is unimaginable. Each command has its own special needs and implementing them simultaneously is almost impossible.

Conventionally a tracer is built by first implementing an interpreter that provides one tracer feature and then iteratively modifying it to accommodate others. This process gets increasingly cumbersome as new features are added. Each feature may call for some program restructuring and require some special state information to be carried as an argument of the tracer. If one is not cautious the continuous tampering with the code very soon deteriorates it structure making the program incoherent. Caution is also required to maintain the status quo of the already implemented features. This need for extra care, compounded with the extensive editing required to add arguments, makes every iteration for adding a feature increasingly laborious.
Figure 8.1: Byrd box model

In this chapter we describe our experience of implementing a Prolog tracer by stepwise enhancement. The problem of building a tracer is decomposed into that of building several independent partial tracers, each providing a set of related features. Since these features are related to controlling the execution of a Prolog program, the partial tracers are by enhancing a Prolog meta-interpreter. The independent partial tracers are composed into a single program. This program is the required tracer as it has the composite functionality of all the partial tracers.

In the following section a quick overview of the working of a Prolog tracer is given. The discussion is not intended as a tutorial on Prolog tracers. For details one is referred to [13] or manual of any Prolog implementation.

8.1 Prolog Tracers

A Prolog tracer, commonly referred to as a debugger, is a debugging tool available with most popular Prolog implementations. These tracers closely follow the Byrd box model [16] of execution for Prolog. In this model there are four "ports" that correspond to the execution of a goal. They are call, exit, redo, and fail. Figure 8.1 gives a schematic representation of a goal invocation. The
call port corresponds to the first time a goal is called. If the goal succeeds then the exit port is visited, and subsequent goals executed. When those goals fail the redo port of the goal is visited to find more solutions. If an alternative solution is found the exit port is visited again. When there are no more solutions for a goal the fail port is visited.

A Prolog tracer facilitates a user to interactively step through the execution of a Prolog program from port to port. The information displayed typically consists of the name of the port, the instantiation state of the goal at the port, the depth of the goal in the proof tree.

The following example gives the information displayed by a typical tracer. Consider the program segment given below. It defines a relation \textit{parent}(X, Y), read as X is a \textit{parent} of Y.

\begin{verbatim}
parent(X, Y) :- mother(X, Y).
parent(X, Y) :- father(X, Y).
\end{verbatim}

The following is a set of facts giving the relation \textit{mother}(X, Y), read as X is a mother of Y,

\begin{verbatim}
mother(sue, john).
mother(mary, rob).
\end{verbatim}

and \textit{father}(X, Y), read as X is a father of Y.

\begin{verbatim}
father(john, rob).
father(peter, john).
\end{verbatim}

On stepping through the complete execution of the query

\begin{verbatim}
1 ?- parent(X, rob).
\end{verbatim}
the DEC-10 Prolog [13] tracrer gives

(1) 0 Call: parent(_407, rob) ?
(2) 1 Call: mother(_407, rob) ?
(2) 1 Exit: mother(mary, rob) ?
(1) 0 Exit: parent(mary, rob) ?

X = mary

The last line gives a value of X for which the query parent(X, rob) is true. The other lines show the goal and the port being entered. The first of the two numbers in each line is a unique number associated to an invocation of a goal at the call port. This number is assigned in the chronological order in which the goals are invoked. It is used as a reference for the goal by some tracer commands. The second number is the depth of the current goal in the proof tree. The '?’ at the end of the line is a prompt from the tracer indicating it is waiting for a command. In the trace above the tracer was asked to step to the next port at each query.

After receiving one answer if the tracer is asked to find more solutions it attempts to retry the query and shows the trace of the computation during backtracking. For instance:

(1) 0 Redo: parent(mary, rob) ?
(2) 1 Redo: mother(mary, rob) ?
(2) 1 Fail: mother(_407, rob) ?
(3) 1 Call: father(_407, rob) ?
(3) 1 Exit: father(john, rob) ?
(1) 0 Exit: parent(john, rob) ?

\[ X = \text{john} \]

\[ X = \text{john} \] is yet another answer for the above query. An attempt to retry the query, shown as \textit{Redo}, resulted into retrying the goal '(2)' or the \textit{mother/2} goal. This goal failed and would not be entered again during this execution of the program. Now the second clause of \textit{parent/2} is selected and a \textit{father/3} goal invoked. The next goal is numbered 3 even though the execution of goal number 2 is completely over.

A request to find more solution again results in the trace

(1) 0 \textit{Redo}: parent(john, rob) ?
(3) 1 \textit{Redo}: father(john, rob) ?
(3) 1 \textit{Fail}: father(_407, rob) ?
(1) 0 \textit{Fail}: parent(_407, rob) ?

no

A \textit{no} indicates no more solutions to the query exist. This may also be seen from the fact that the top level goal has entered the \textit{Fail} port, so there is no other goal to be invoked.

Just stepping through the code from port to port as done above is hardly good for debugging a program. One may wish to execute some goals without displaying the trace, or to quietly execute until reaching a spy point, or even revert execution to some previous state in the trace. Most tracers are equipped
with a repertoire of commands that provide such support. Here is subset of such commands:

\[
\begin{align*}
\text{abort} & \quad \text{abort execution and return to top level} \\
\text{creep} & \quad \text{take a single step and interact again} \\
\text{retry } < \text{NodeNum} > & \quad \text{return to the call port of the goal with id } < \text{NodeNum} >. \\
\text{skip} & \quad \text{prove the current goal ignoring spypoints} \\
\text{true} & \quad \text{forcibly succeed the current goal} \\
\text{fail } < \text{NodeNum} > & \quad \text{forcibly fail the goal with id } < \text{NodeNum} > \\
\text{leap} & \quad \text{interact again only when a spy point is found} \\
\text{quasi-skip} & \quad \text{interact again either when a spy point is found} \quad \text{or when you return to this goal again} \\
\text{show ancestors N} & \quad \text{show previous N ancestors}
\end{align*}
\]

These commands may be given when the tracer gives a '?' prompt at the end of the status line. Most tracers associate a single character to each command (usually the first character of the command name). For ease of usage, besides 'c', the 'space' and 'carriage return' characters are also mapped to 'creep' - the command to go to the next port.

At a time only one command may be given. The tracer performs the action corresponding to the command and displays the status line, and the queries for the next command. The action may cause the execution to move to a different port. It may simply display some information (e.g. show ancestors) and not cause a movement from the port.
8.2 Decomposing the Problem

A Prolog tracer provides several features. Each of these features constitute a logically independent subproblem. The logic for retrying a goal is independent from that of skipping a goal or performing a leap up to a spy point. Thus, an obvious decomposition of its programming task is one performed around these commands. Besides forming a separate logical entity these commands are also complex enough to deserve independent attention.

We divide the tracer into the following logical subproblems:

- port-to-port tracer
- leap and quasi-skip tracer
- retry and fail tracer
- skip and true tracer
- show ancestor tracer

This decomposition is performed by grouping together commands that have similar processing needs. The similarity between the commands in a group would become clear from the discussion later.

Each logical subproblem above represents a partial tracer. A tracer because it allows tracing the execution of a Prolog program, and partial because it does not provide all the support provided by a complete tracer. The port-to-port tracer allows stepping through every port of the program exhaustively. The leap and quasi-skip tracer displays the status at a port and allows leap and quasi-skip commands. Similarly the other partial tracers also provide some features of the final tracer.

What is interesting further is that though the collection of these partial tracers provide all the tracer commands, the final tracer cannot be realized by
simply putting the collection together and adding some top level procedure that passes control between them. This is because each of the partial tracers trace the object program separately. To be able to pass control between these tracers their interpreters have to synchronized. This cannot be done.

To provide the correct behavior the Prolog tracer consists of one program that traces the execution of an object program and simultaneously provides the features of all the partial tracers. In other words the logic for various partial tracers is interwoven in the complete tracer.

The above discussion implies that though the partial tracers constitute logical subproblems of a tracer, they can not be developed separately and put together using conventional programming approaches to form a tracer. When logical subproblems cannot be mapped onto physical subcomponents, as is the case with developing a Prolog tracer, the method of stepwise enhancement suggests developing programs that exhibit partial behavior of the problem by enhancing a common subprogram called a skeleton program. These partial programs are then composed to generate a final program.

The partial tracers stated above can be enhanced from a partial tracer that has provision to interactively step through every port in the execution of the program. Such a program may be called a stepper and is a good candidate for a skeleton program. The steps in developing a tracer by stepwise enhancement may then be summarized as:

- Develop the stepper.
- Enhance the stepper into various partial tracers.
- Compose these partial tracers to form the tracer.

A stepper identified above is a partial tracer that enables one to step
through the execution of a Prolog program stepping from one port to the
next port interactively. Before it enters a port it stops and displays the status
of the goal at that port. On receiving a creep command it proceeds to the next
port. On entering a port it displays the same status as that displayed by the
tracer.

To develop a stepper we once again look at its various subproblems. In order
to step through the execution of a Prolog from port to port it is essential that
the stepper be able to execute a Prolog program and also identify its ports.
Further the depth and the unique node number must be displayed. The three
tasks, that of

- identifying ports,
- computing depth, and
- associating a unique number to every goal

are independent of each other. It would therefore be desirable that they be de-
veloped as separate subprograms and be made procedures of the final stepper.

Once again we have subproblems which are logically independent but cannot
be mapped into separate physical subcomponents in the final program. The
stepper should display the port, depth, and node number during the same ex-
ecution of the program. Having three different subprograms performing their
functions on different computations would not serve the purpose.

The above discussion suggests that the stepper itself may be developed by
stepwise enhancement. To do so we first identify a program that may form
the skeleton of the stepper. This is not hard. One thing common to the three
tasks mentioned above is that they all execute a Prolog program. A Prolog
interpreter is therefore an obvious choice as a skeleton for developing a stepper
by stepwise enhancement.

The steps to develop a stepper by stepwise enhancement may be summarized as:

- Develop a Prolog interpreter, call it `solve`.
- Enhance `solve` to identify the entry into the ports of a goal, call it `port`.
- Enhance `solve` to compute depth, call it `depth`.
- Enhance `solve` to give a unique number to each goal, call it `nodenum`.
- Compose the above interpreters into one interpreter, say `pre-stepper`.

The program `pre-stepper` generated after composing at this step is not the required stepper. This program just 'identifies' the ports and computes the required values. It neither display the status, nor does it interact with the user. Thus as a last step:

- Enhance `pre-stepper` to display the status information and provide user interface, call it `stepper`.

### 8.3 Developing a Prolog Interpreter

The overall plan of enhancement and composition steps for developing the tracer are outlined in Figure 8.2. The first important milestone in this plan is to develop a Prolog interpreter. Since we intend to develop the tracer in Prolog this interpreter has to be written in Prolog. For the purpose of writing node number, port, and depth the abstraction level of the vanilla meta-interpreter of Figure 3.2 is sufficient. It may therefore be used as the starting skeleton program.

In our actual experiment we have constructed a tracer that implements cut and allows tracing of extra-logical predicates. These are however not necessary
Figure 8.2: Schematic for developing a tracer
to describe the development method and hence omitted.

8.4 Developing Pre-stepper from Interpreter

Having developed the interpreter the next milestone in developing a tracer is building a pre-stepper. This program forms the basis for building the stepper. The following subsections discuss its development as per the steps outlined in Figure 8.2.

8.4.1 Enhancement to identify ports

The story told by the tracer is based on the Byrd box model of Prolog [16]. Following this model of execution, a goal may be inspected at its four ports: call, exit, redo, and fail. Hence, it is important to identify these ports corresponding to each goal that is traced.

It may be noted that only atomic goals have ports and that all system goals and user defined goals are atomic. In the solve interpreter of Figure 3.2, the last two clauses process such goals. The clause

\[
solve(Goal) :- sys(Goal), \text{call}(Goal).
\]

evaluates system defined predicate. The predicate call/1 passes the goal to the underlying Prolog inference engine for evaluation. The invocation of this solve clause by the engine leads to entering the ‘call’ port of the system goal. Similarly, a retry on the clause leads to a retry on the system goal. Conversely, if the goal succeeds the execution of ‘is solve clause succeeds and may be retried later, if the goal fails the clause fails and may not be retried again.

The same may be said for the fourth clause of Figure 3.2:
solve(Head) :- unfold_clause(Head, Body), solve(Body).

This clause is responsible for processing user-defined goals. The invocation, success, retry, and failure of the object goal can be mapped to equivalent states of this solve clause.

The vanilla interpreter therefore uses Prolog's backtracking mechanism implicitly to provide backtracking for the object program. The entry and exit for ports of object program goals can be identified with respect to the entry and exit of these two clauses. The program in Figure 8.3 is enhanced from Figure 3.2 using this knowledge. The procedures call_port/1, fail_port/1, exit_port/1, and retry_port/1 are invoked just before entering the call, fail, exit, and retry port of an object program goal. These procedures (and their later extensions) are (expected to be) deterministic. The procedures call_and_fail_port/1 and exit_and_retry_port/1 link to the port identifying clauses. The order of clauses in these link procedures is significant as they assume that the first clause is selected on call and the second one on retry.

The clauses corresponding to the last two clauses of vanilla have been folded into a new procedure port_exec/1 to provide a single point for operating on atomic goals - both user and system defined goals.

8.4.2 Enhancement to compute depth

The interpretation of a Prolog program is a recursive operation. The execution of a goal triggers the execution of goals in the body of its clause. The goals in the body of a clause are therefore one level deeper in the execution tree than the procedures whose body it is in.

Program depth of Figure 8.4 is a Prolog interpreter that keeps track of the
port(true).
port((GoalA, GoalB)) :-
    port(GoalA), port(GoalB).
port(Pred) :-
    [call call_and_fail_port(Pred), fail]
    [call port_exec(Pred), fail]
    [call exit_and_retry_port(Pred), fail]
    [exit exit_and_retry_port(Pred), redo]

port_exec(Goal) :-
    sys(Goal).
    call(Goal).
port_exec(Head) :-
    unfold_clause(Head, Body),
    port(Body).

call_and_fail_port(Goal) :- call_port(Goal).
call_and_fail_port(Goal) :- fail_port(Goal), fail.

exit_and_retry_port(Goal) :- exit_port(Goal).
exit_and_retry_port(Goal) :- retry_port(Goal), fail.

% procedures called before entering the corresponding
% ports of the object goal
call_port(Goal). fail_port(Goal).
exit_port(Goal). retry_port(Goal).

Figure 8.3: Identify 'ports' of object program
depth(Goal) :- depth(Goal, 0).

depth(true, Depth).
depth(((GoalA,GoalB), Depth) :-
    depth(GoalA, Depth),
    depth(GoalB, Depth).
depth(Pred, Depth) :-
    Depth1 is Depth+1,
    depth_exec(Pred, Depth1).

depth_exec(Goal, Depth) :-
    sys(Goal),
    call(Goal).
depth_exec(Head, Depth) :-
    unfold_clause(Head, Body),
    depth(Body, Depth).

Figure 8.4: Compute depth

depth of the object goals. It is enhanced from vanilla.

8.4.3 Enhancement for numbering node

The retry and fail commands of the tracer refer to goals that have already
been invoked. To help the user in referencing they associate a unique number
with each goal, called the node number. A node number is associated to every
goal that is partially or fully evaluated whether with success or failure, but
not to goals that are pending evaluation. The number is unique in that no two
goals whatever their state of computation may have the same identifier.

This is achieved by assigning numbers to the goals in the chronological
order of their invocation. A new number is generated by incrementing the last
number assigned. The program nodenum in Figure 8.5 uses this logic to assign
node numbers to goals.
The requirement that the number returned by \texttt{next_node_number/1} be unique and not be repeated even after a computation fails calls for some destructive computation. The following program segment performs this work.

\begin{verbatim}
initialize_nodenum(Number) :-
    retractall(\$nodenum'(X)),
    asserta(\$nodenum'(Number)).
\end{verbatim}

\begin{verbatim}
next_node_number(NewNumber) :-
    retract(\$nodenum'(LastNumber)).
    NewNumber is LastNumber + 1,
    asserta(\$nodenum'(NewNumber)).
\end{verbatim}

The procedures \texttt{retractall/1}, \texttt{retract/1}, \texttt{is/2}, and \texttt{asserta/1} are standard system defined primitives, and have the usual meaning.

### 8.4.4 Composing enhanced interpreters

The interpreters \texttt{nodenum}, \texttt{depth}, and \texttt{port} provide the three subtasks identified for a stepper. The subtasks are performed as different interpreters. For a stepper these tasks should be performed simultaneously. The enhanced interpreters cannot be used directly to compute both the values while entering a port of a goal.

The same interpreter should compute depth, number nodes, and identify ports of a goal. We generate this interpreter, \texttt{pre-stepper} of Figure 8.6, by composing the interpreters \texttt{nodenum}, \texttt{depth}, and \texttt{port} of Figure 8.3, 8.4 and 8.5 respectively. As these interpreters are all enhanced from vanilla interpreter of Figure 3.2 they interpret the Prolog program in the same way. The extra
nodenum(Goal) :-
    initialize_nodenum(0),
    nodenum(Goal).

nodenum(true).

nodenum((GoalA, GoalB)) :-
    nodenum(GoalA),
    nodenum(GoalB).

nodenum(Pred) :-
    next_node_number(NewNodenum),
    nodenum_exec(Pred).

nodenum_exec(Goal) :-
    sys(Goal),
    call(Goal).

nodenum_exec(Head) :-
    unfold_clause(Head, Body),
    nodenum(Body).

Figure 8.5: Associate unique node number
pre_stepper(Goal) :-
    initialize_node_num_pre_stepper(0),
    pre_stepper(Goal,0).

pre_stepper(true, Depth).
pre_stepper((GoalA,GoalB), Depth) :-
    pre_stepper(GoalA, Depth),
    pre_stepper(GoalB, Depth).
pre_stepper(Pred, Depth) :-
    next_node_number_pre_stepper(NewNodeNumber),
    Depth1 is Depth+1,
    call_and_fail_port_pre_stepper(Pred),
    pre_stepper_exec(Pred, Depth1),
    exit_and_retry_port_pre_stepper(Pred).

pre_stepper_exec(Goal, Depth) :-
    sys(Goal),
    call(Goal).
pre_stepper_exec(Head, Depth) :-
    unfold_clause(Head, Body),
    pre_stepper(Body, Depth).

Figure 8.6: Pre-stepper

computation required for their individual function is added on the skeleton provided by vanilla. The composition process merges the extra computation performed by these interpreters around this skeleton.

8.5 Predicate Naming Conventions

In describing the programs so far we have presented the complete text resulting after enhancement. All affected predicates were renamed and those whose functionality did not change retained the same name.

In the later discussion we follow a convention for naming the predicates of
a program such that the new program (child) may be described by stating the
initial program (parent) and presenting only the the clauses that are altered.
The other clauses of the child can be derived by renaming the predicates of
the parent program.

As a convention every program has a name associated to it. All user de-
finite predicates of this program has this name as a part of their predicate
name. The predicates of program pre-stepper have pre-stepper in their name,
for example pre-stepper_exec, exi!_and_retry_port.pre-stepper. The clauses of
the child program that are inherited unchanged from the parent program may
be derived by replacing the string “parent” by “child” in the predicate name
of these unchanged clauses.

At this point we assume that all procedures of pre-stepper that are borrowed
as is from the operands of composition are suffixed with the string pre-stepper.
The predicate call_port of program port thus becomes call_port.pre-stepper, and
initialize.nodenum becomes initialize.nodenum.pre-stepper. This is reflected in
the segment of pre-stepper of Figure 8.6.

8.6 Developing Stepper from Pre-Stepper

The program pre-stepper performs the subtasks required for the stepper. It
next stage in the development enhancing it to display the port status before
entering every port and accepting user commands.

The enhancement for stepper requires trapping the execution just before
entering a port of the object program and performing the necessary interaction.
The procedure call_port.pre-stepper/1 is called before entering the call port of
the object goal. This procedure is thus the obvious choice to put the trap
before entering the call port. Similarly, the procedures \texttt{fail\_port\_pre\_stepper/1}, \texttt{exit\_port\_pre\_stepper/1}, and \texttt{retry\_port\_pre\_stepper/1} are the right choice for setting traps at the entry into the fail, exit, and retry ports respectively.

The status displayed at each port includes the node number, depth, port, and the goal. In the procedures selected above only the port and goal are available. The node number and depth information should therefore be propagated to these procedures from the third clause of \texttt{pre\_stepper/2}. The enhancement requires changing the third clause of \texttt{pre\_stepper/2} to:

\begin{verbatim}
stepper(Pred, Depth) :-
    Depth1 is Depth+1,
    next_node_number(NewNodeNum),
    \% propagate Depth and Node number to the port handlers
    call_and_fail_port_stepper(Pred, Depth1, NewNodeNum),
    stepper_exec(Pred, Depth1),
    \% propagate Depth and Node number to the port handlers
    exit_and_redo_port_stepper(Pred, Depth1, NewNodeNum).
\end{verbatim}

and changing the port handling procedures to that given in Figure 8.7. The modifications includes propagating the depth and node number information, and addition of predicate \texttt{wait\_at\_port\_stepper/4} that calls \texttt{process\_command/4} to display the the status line and process commands. The predicate \texttt{wait\_at\_port\_stepper/4} succeeds exactly once and thus preserves the interpretation behavior of the Prolog object program.

The command interface provided by \texttt{process\_command/4} is written with future enhancements in mind. It is therefore termed a \textit{generic command interface} and is described in the following subsection.
call_and_fail_port_stepper(Goal, Depth, NodeNum) :-
    call_port_stepper(Goal, Depth, NodeNum).

call_and_fail_port_stepper(Goal, Depth, NodeNum) :-
    fail_port_stepper(Goal, Depth, NodeNum).
    fail.

exit_and_redo_port_stepper(Goal, Depth, NodeNum) :-
    exit_port_stepper(Goal, Depth, NodeNum).

exit_and_redo_port_stepper(Goal, Depth, NodeNum) :-
    redo_port_stepper(Goal, Depth, NodeNum),
    fail.

% Extended procedures for processing at the goal's ports

call_port_stepper(Goal, Depth, NodeNum) :-
    wait_at_port_stepper(call, Goal, Depth, NodeNum).

fail_port_stepper(Goal, Depth, NodeNum) :-
    wait_at_port_stepper(fail, Goal, Depth, NodeNum).

exit_port_stepper(Goal, Depth, NodeNum) :-
    wait_at_port_stepper(exit, Goal, Depth, NodeNum).

redo_port_stepper(Goal, Depth, NodeNum) :-
    wait_at_port_stepper(redo, Goal, Depth, NodeNum).

wait_at_port_stepper(Port, Goal, Depth, NodeNum) :-
    process_command(Port, Goal, Depth, NodeNum),!.

% To prevent this procedure from failing or backtracking

wait_at_port_stepper(_Port, _Goal, _Depth, _NodeNum).

.  Figure 8.7: Port handling
8.6.1 Generic user interface

The user interface is activated by calling `process_command/4` and is designed with an eye towards its eventual usage. Several actions, for instance parsing a command or displaying the status of a goal, for different tracer commands are similar. There are other actions, such as marking a '***' in front of a status line of a goal defined as a spy point, that are different.

This observation led us to design a generic user interface for the stepper. It is generic because it performs work common across various trace features and provides a platform to add the specific requirements for individual features. The latter is provided by means of software hooks at specific points. Hooks are calls to as-yet-undefined procedures at specific points in the program. These procedures have some expected behavior and would be defined by the extension for individual features.

Hooks in the user interface code enable incremental addition of features as and when they are developed or composed.

8.6.2 Generic command interface

The program in Figure 8.8 provides a generalized platform for developing an interface for the various commands. It provides the skeleton to display status, query commands, and process commands. During each of these operations it provides software hooks to perform actions specific to a particular tracer command.

8.6.3 Hooks

The generic user interface has the following hooks.
process_command(Port, Goal, Level, NodeNum) :-
    print_node_state(Port, Goal, Level, NodeNum),
    query_command(Cmd, Argument, Port),
    process_option(Cmd, Argument, Port, Goal, Level, NodeNum).

print_node_state(Port, Goal, Level, NodeNum) :-
    print_node_state_hook(Port, Goal, Level, NodeNum),
    fail.

print_node_state(Port, Goal, Level, NodeNum).

query_command(next, [], Port) :-
    unleashed_port(Port), !.

query_command(Cmd, Argument, Port) :-
    query_option(Char, Argument),
    map_char_to_command(Char, Cmd).

map_char_to_command(Char, Cmd) :-
    map_char_to_command_hook(Char, Cmd), !.

map_char_to_command(Char, Char).

process_option(Cmd, Argument, Port, Goal, Level, NodeNum) :-
    process_option_hook(Cmd, Argument, Port, Goal, Level, NodeNum).

process_option(Cmd, Argument, Port, Goal, Level, NodeNum) :-
    % Command not processed at any hook
    % Give an error message and get another command
    invalid_command_message(Cmd),
    process_command(Port, Goal, Level, NodeNum).

Figure 8.8: Generic command interface
- \texttt{print\_node\_state\_hook(+Port, +Goal, +Level, +NodeNum)}

  This is called before printing the status of the goal. Printing of `*` for spy point, or `>` for skip may be done at this goal. These actions have side-effects. If multiple actions are specified, they should be ordered manually for correct behavior of the composed result.

- \texttt{map\_char\_to\_command\_hook(+Char, -Cmd)}

  This maps a command character to some internal name.

  \begin{verbatim}
  map\_char\_to\_command\_hook(0\texttt{c}, creep).
  map\_char\_to\_command\_hook(0\texttt{a}, abort).
  map\_char\_to\_command\_hook(0\texttt{r}, retry).
  \end{verbatim}

  Note \texttt{0\texttt{X}} is the numeric code for character \texttt{X}.

- \texttt{option\_not\_valid\_at\_port\_hook(+Port, +Cmd)}

  Some actions may not be valid at all ports, for instance skip does not make sense at the exit or fail port. This hook is used to flag such actions.

- \texttt{process\_option\_hook(+Cmd, +Arg, +Port, +Goal, +Level, +NodeNum)}

  This hook processes a tracer option. For the bare stepper the following clauses do the required processing.

  \begin{verbatim}
  process\_option\_hook(next, _Argument, _Port, _Goal, _Depth, _NodeNum).
  process\_option\_hook(abort, _Argument, _Port, _Goal, _Depth, _NodeNum)
     :- abort.
  \end{verbatim}

- \texttt{help\_message\_hook(+Char, -Message)}

  This hook gets the help document for a command.
help_message_hook(abort, 'ABORT execution').
help_message_hook(next, 'Go to NEXT step').

Some options may require additional information. To show ancestors of a goal, one has to carry the list of its ancestors, or to show the clause selected for a goal the relevant clause should be available. To add these features the generic command interface would have to be enhanced to propagate the requisite information from the interface to the hooks. This may be done by propagating context via the port handler and the command interface.

8.7 Developing Tracer from Stepper

The final milestone outlined in Figure 8.2 is building the tracer. It is built by enhancing the stepper separately for each set of related features. The resulting partial-tracers are then composed to get the complete tracer.

In the discussion that follows we describe only the pieces of code affected while modifying the interpreter of Figures 8.6 and 8.3. We use italics to show newly added goals or clauses. Description of the behavior of support procedures used to cause a change is given in pseudo-code. The actions at the hooks are described in text. The tracer resulting from composing these pieces is given in Appendix F.

8.7.1 Enhancements for leap and quasi-skip

When debugging large programs tracing through its complete execution may become quite tedious. One may instead wish to inspect its flow only on entry into the ports of some specific goals, called spy points. The leap and quasi-skip
commands provide this support. They execute the object program without interacting with the user or displaying the status until a spy point is reached. A goal is designated as a spy point by a separate command outside the tracer.

Quasi-skip differs from leap in that it puts the interaction off only for the execution of the current goal, and can be requested only at the call or redo ports of a goal. For a quasi-skip interaction is restarted either when a spy point or either of exit, redo, or fails port of the same goal is reached.

The stepper can be enhanced to a partial tracer leap that provides these two commands. Processing these commands requires remembering the fact that leap or quasi-skip is requested, and by-passing the command processing till the interaction enabling condition is satisfied.

That 'leap' or 'quasi-skip' is in progress is remembered by asserting

control_flag(leap, NodeNum).

or

control_flag(quasi_skip, NodeNum).

to the Prolog database in the process_option_hook, where NodeNum is the node number of the port where that command was invoked. This flag is used by the partial tracer to globally announce the selection of these commands and the arguments required for their processing to other parts of the stepper. The predicate leap_or_quasi_skip_in_progress used later succeeds if one of the above control_flag/2 exists in the database.

To inhibit the interaction the procedure wait_at_port_stepper/4 is modified just before control is passed to the generic interface.

wait_at_port_leap(Port, Goal, Depth, NodeNum) :-
resume_trace_at_quasi_skip_node(NodeNum),
resume_trace_at_spy_point(Goal),
process_command(Port, Goal, Depth, NodeNum),
!
wait_at_port_leap(Port, Goal, Depth, NodeNum).

The predicate `resume_trace_at_spy_point/1` guards the command interaction part. When either leap or quasi-skip request is being processed, it checks if the goal whose port is being entered is a spy point. If so, it removes the control flag and lets the normal interaction to happen, otherwise it simply fails.

```prolog
resume_trace_at_spy_point(Pred) :-
    leap_or_quasi_skip_in_progress, !,
    spy_point(Pred),
    remove_control_flag,
    resume_trace_at_spy_point(Pred).
```

The predicate `resume_trace_at_quasi_skip_node/1` provides the difference in behavior between leap and quasi-skip.

```prolog
resume_trace_at_quasi_skip_node(NodeNum) :-
    control_flag(quasi_skip, StartNum), !.
    StartNum = NodeNum,
    remove_control_flag,
    resume_trace_at_quasi_skip_node(NodeNum).
```
Note the use of '!" in the two procedures. It makes the procedures fail if a
leap or quasi-skip is in progress but the condition to reinitiate interaction is
not met. If neither leap nor quasi-skip is in progress the above two predicates
simply succeed, letting the tracing continue as usual.

8.7.2 Enhancements for skip, unconditional true

Like quasi-skip, skip causes tracing to be stopped until the current goal is
completely executed. It differs from quasi-skip in that it is defined only at the
call port, and it completely ignores the spy-points. Interaction is resumed only
after the skipped goal succeeds or fails. On the contrary a true forces a goal
to succeed unconditionally.

When neither a goal nor its subgoals are traced it is a waste to have the
tracer interpret it. This goal may rather be executed directly by the Prolog
gine, avoiding the runtime and memory overhead due to interpretation. The
enhancements to build a partial tracer skip from stepper reflect this decision.
The basic interpretation method of goals is modified. The command un condi-
tional true is grouped with skip for the similarity of the changes required.

A change in the method of interpreting some goals in skip as compared to
stepper/3 is done by first by creating a new clause, say

\[
\text{stepper\_exec\_extend(Pred, Depth) :-}
\text{stepper\_exec(Pred, Depth).}
\]

in stepper/3, and replacing the call to stepper\_exec/3 in its last clause by a
call to stepper\_exec\_extend/3.

Enhancement to 'skip' is achieved by modifying the stepper\_exec\_extend/2
procedure as follows.
skip_exec_extend(Pred, Depth) :-
    test_and_reset_unconditional_true, !.

skip_exec_extend(Pred, Depth) :-
    test_and_reset_skip, !,
    call(Pred).

skip_exec_extend(Pred, Depth) :-
    skip_exec(Pred, Depth).

Like the previous commands, that a skip or unconditional true command is in progress is `assert’ed` by the user interface as control flags: respectively, 

control_flag(skip, _).

or

class_control_flag(unconditional_true, _).

The function of the predicates introduced in `skip_exec_extend/2` may be fairly obvious from the context of this discussion and the names of the new predicates. They are defined as follows:

test_and_reset_unconditional_true :-
    control_flag(unconditional_true, _).
    reset_control_flag.

test_and_reset_skip :-
    control_flag(skip, _),
    reset_control_flag.

The predicate `reset_control_flag` removes the definition of `control_flag/2` predicate from the database.
8.7.3 Enhancements for retry and fail

The retry and fail commands take an argument \( N \) which is the node number of a goal that has already been invoked. Retry causes the tracer to go back to the call port of the goal corresponding to this node. Fail forces this goal to fail. The \( retry \) partial tracer is also developed by enhancing \( stepper \).

The operand of the two commands is a goal that has appeared in the computation thus far. To perform the required action the tracer should fall back to its state that processed this goal. Due to the nature of these commands the enhancements in this case are very non-trivial.

During command processing a control flag similar to the previous commands is asserted. The control flag gives which of the commands is invoked and the target node number.

The backtracking of the tracer is performed by forcing a failure at all the choice-points between the node where the command is given and the node that is to be failed or retried. The following code is modified to ripple failures upwards from ports of those goals that are not fully evaluated.

call_and_fail_port_retry(Goal, Depth, NodeNum) :-
    call_port_retry(Goal, Depth, NodeNum).

call_and_fail_port_retry(Goal, Depth, NodeNum) :-
    fail_this_node(NodeNum),
    fail_if_going_back_for_retry_for_fail.
    fail_port_retry(Goal, Depth, NodeNum),
    fail.

exit_and_redo_port_retry(Goal, Depth, NodeNum) :-
    exit_at_port_retry(Goal, Depth, NodeNum).


exit_and_redo_port_retry(Goal, Depth, NodeNum) :-
  fail_if_going_back_for_retry_for_fail,
  redo_at_port_retry(Goal, Depth, NodeNum),
  fail.

: wait_at_port_retry(Port, Goal, Depth, NodeNum) :-
  process_command(Port, Goal, Depth, NodeNum), !,
  \% trigger failure if a retry or fail
  fail_if_going_back_for_retry_for_fail.

wait_at_port_retry(_, _, _, _).

The following procedures are used to force a failure due to retry or fail.

fail_if_going_back_for_retry_or_fail :-
  not retry_or_fail.

This predicate fails if the tracer is going back to retry or fail a goal, it succeeds otherwise. It causes the fail port of all goals on the way to the selected goal from being bypassed. Thus the tracer does not query for user commands at these fail ports. The following predicate checks if the current node is selected for being failed. If so it removes the control_flag/1 from the database. This inhibits the bypassing of fail port of the selected goal thereby leading to its failure, as required by the fail command.

fail_this_node(NodeNum) :-
control_flag(fail, Nodeum)
   -> reset_control_flag
   ; true.

Besides bypassing entry into the ports of active goals the choice points corresponding to the object program goals have to be failed too. This would inhibit selection of unselected clauses. A similar block is required for the system goals being retried. Such choice-points are created in the procedure stepper Exec/2. Modifying it as follows 'cuts' the choice-points when failures are triggered due to a retry or fail command.

retry_exec_extend(Goal, Depth) :-
   retry_exec(Goal, Depth),
   (true:retry_or_fail,!;fail).
retry_exec(Goal, Depth) :-
   sys(Goal), !, call(Goal).
retry_exec(Head, Depth) :-
   unfold_clause(Head, Body), retry(Body, Depth).

The predicate retry_or_fail/0 succeeds if control_flag(retry...) or control_flag(fail...) is true.
The above changes take care of backtracking to the state where the target goal was first invoked. If a ‘fail’ is requested the actions required for failing a goal is also taken care of. What remains is revisiting the call port of the target goal if a ‘retry’ is in effect. This is performed by adding the goal check_to_stop_retrying/1 to the last clause of the procedure stepper/2 to give:

\[
\text{retry}(\text{Pred}, \text{Depth}) :- \\
\text{Depth1 is Depth+1}, \\
\text{next_node_number}(\text{NewNodeNum}), \\
\text{check_to_stop_retrying}(\text{NewNodeNum}), \\
\text{call_and_fail_port_retry}(\text{Pred}. \text{Depth1}. \text{NewNodeNum}), \\
\text{retry_exec}(\text{Pred}. \text{Depth1}), \\
\text{exit_and_redo_port_retry}(\text{Pred}. \text{Depth1}. \text{NewNodeNum}).
\]

The newly added procedure is defined as:

\[
\text{check_to_stop_retrying}(\_\text{Current\_Node}). \\
\text{check_to_stop_retrying}(\text{Current\_Node}) :- \\
\text{retry_this_node}(\text{Current\_Node}), \\
\text{reset_retry}, \\
\text{reassert_old_node_number}(\text{Current\_Node}). \\
\text{check_to_stop_retrying}(\text{Current\_Node}).
\]

The first clause is invoked before entering the call port of an object goal and simply succeeds. The second clause is selected after this goal failed. The predicate \text{retry\_this\_node/1} checks if this node is requested to be retried. If it is, the control flag is removed, the unique count reset to this node number, and
the call port reentered. The reentry into the call port is performed by succeeding the \texttt{check_to_stop_retrying/1} procedure. The recursive call is performed to enable a subsequent retry on the same goal.
Chapter 9

Closing Remarks

In the landmark paper on structured programming [27] written in 1975 Dijkstra notes:

"We may expect that computers will become more directly accessible for the individual user and we may expect that the latter should like to use its capabilities for the text manipulations involved in program composition. At present, I am rather unsure about the true nature of the text manipulations the user would then like to perform – It is certainly more structured than deletion and insertion of characters or lines!"

A lot has happened in the fourteen years since these statements were made. Current computer systems certainly provide better environments for program construction than those of the seventies. It is however surprising that the capabilities they provide for manipulating text for the purpose of composing programs are still far from being termed as structured. Changes that are conceived on programs are still performed on text using enriched forms of insertion and deletion commands.

The lack of structured text manipulation support, in my opinion, is because the methods of programming prevalent today do not encourage manipulation of existing programs. In that respect, stepwise enhancement conceptualizes the manipulations commonly performed on programs. The enhancement and
composition tools provide a mechanism to manipulate a program at the same level of abstraction at which the changes are conceived. These ideas are a step in the direction of identifying methods and tools that enable structured manipulation of text.

9.1 Choosing a Programming Method

A question that I have asked myself very often and others have asked me too is:

When should one use stepwise enhancement?

This question intrigued me till I realized that programming is not an exercise in picking up one of the various programming methods: stepwise refinement [27, 100], module decomposition and information hiding [65], object-oriented programming [24], meta-programming [48], Jackson System of Programming [17], abstract data typing, or what have you. It is an exercise in problem-solving. Like any other problem-solving exercises the steps in program construction involve decomposing the problem into subproblems, solving the subproblems, and putting their solutions together to get the solution to the final problem.

The first step in constructing a program for a problem is not what method to use but what is the most natural decomposition of the problem. The intent is to break the problem into intellectually manageable pieces. What is intellectually manageable or what is a natural decomposition would differ from problem to problem. It may also differ from one person to another, or for that matter between different stages in the life of the same person.

The choice of a programming method comes next. The various program-
ming methods differ on how they decompose a problem and then how they combine the solutions to the subproblems into the solution for the complete problem. Once you have pieces whose complexity you can manage, you choose the method that best captures the decomposition. An important concern when making this choice is that the method should provide a simple way to combine the partial solutions into a complete solution.

In practice however the knowledge of various programming methods plays a significant role in decomposing a problem. If you are not familiar with meta-programming there is a little chance that you would decompose a problem into logic and control [48]. The lack of this knowledge would not prohibit you from writing programs to solve a problem made for meta-programming. On the contrary the knowledge of meta-programming would enable you to arrive at the most natural solution for this problem and possibly in less time.

Stepwise enhancement adds to the arsenal of programming methods. It fills in a void left out by other methods. The method is specially useful for programs like the tracer that have several independent features built on the same flow of program control. However, given a problem, deciding whether its program has such characteristics and if so what its flow of control is, is beyond the scope of the method. Such decisions are best left to the programmer. This is analogous to leaving the decision of identifying the modules of a program to the programmer [65]. The correct decomposition of the problem at times evolves over time. The decomposition of a compiler into lexical analyzer, parser, code optimizer and generator evolved from years of research.
9.2 Future Works

The thesis identifies some relationships between program structures and the process of constructing logic programs. These relationships, in my opinion, are not restricted to logic programs. The greater abstraction of logic languages enabled in making these relationships explicit: they implicitly exist in other languages.

It would be worth attempting to identify traits similar to enhancements in other programming languages. To do so we need to have a better understanding of the characteristics of what a skeleton program is independent of the language. Our initial experiments with enhancement and composition tools demonstrate the potential of their use in program development. We believe that these tools should be integrated with other interactive programming environments. For instance, a repertoire of primitives to enhance and compose programs may be added to a language sensitive editor generated by the synthesizer generator [68], or in the high-level editing commands of the knowledge based specification assistant [43], or as language specific extension of Emacs [79].

A system as that outlined above may be integrated with other software engineering environments to record the history of development of a program in terms of enhancements and composition. This can be used to replay the program construction to aid in understanding the program for program maintenance and modification. This would make explicit the various components that went into making the final program. It also gives a maintainer the ease of understanding independent issues in isolation of the others and to explicitly focus on the interdependence during composition. Modifications to the sys-
tem can be performed by modifying the effected component and replaying the construction. Such a system has been envisioned by Scherlis and Scott [76].
Appendix A

Overview of Appendices

The following appendices contain listings of all the systems described in the thesis:

**Interpreter for “full” Prolog**: The partial evaluator, the Prolog tracer, and the technique interpreters interpret Prolog programs. In the thesis these systems have been presented as enhancements of the vanilla interpreter of Figure 3.2.

This section presents an interpreter for “full” Prolog. The enhancements performed on the vanilla interpreter to obtain a “pure” Prolog tracer or other “pure” Prolog meta-interpreters may be applied to this interpreter to create a corresponding system for “full” Prolog.

**ProMiX Partial Evaluation System**: The code for all the modules of ProMiX, the partial evaluation system described in 5 is presented. In contrast to the discussion in the thesis, the code presented partially evaluates “full” Prolog.

**Technique Interpreters**: contains technique interpreters for the three programming techniques identified in Chapter 6.

**Program Composition**: contains code for the composition algorithm discussed in Chapter 7.
Tracer : contains the result of composing the partial tracers developed in
Chapter 8. Only the core of the tracer is presented here. Most of other
pieces of code have been discussed in the chapter.

Support Procedures : This section compiles all the general purpose proce-
dures used in the Prolog interpreter, partial evaluator, tracer, and the
program composition system. In most cases only the expected behavior
of the predicate is given. The code for these predicates is reasonably
straight forward and should not be difficult to write.

The code presented in thesis has been tested on SICStus Prolog [18] Version
0.6 #13.

Conventions of header comments

At the beginning of some predicates you may notice a block of comments such
as:

\% mix(+GOALS, -RESIDUES)
\% partially evaluates GOALS and returns the suspended computation
\% as RESIDUES
\% Used for partially evaluating queries (like when testing or
\% demonstrating ProMiX)

The block documents the procedure following it. The first line declares the
predicate being commented. It gives the name, number of arguments, and
mode declaration of the predicate. Besides it also associates a name to each
argument. In the later lines of the block these names are used to refer to the
respective arguments. As a convention only upper case characters and '-' are
used to form an argument name. This makes it convenient to distinguish the use of a word as a reference to an argument from its use as an English word just for description. In several cases, such as the use of the words GOALS and RESIDUES above, the argument name serves both purposes: it is a symbolic reference to an argument and it is meaningful in the context used.

The above block for instance is a comment about the predicate with functor name mix and arity 2. The +GOALS in the first argument says that the mode of that argument is + and its name GOALS. Similarly the mode of the second argument is – and its name RESIDUES. The mode + means an argument used for input and – an argument used for output. When an argument can be used both for input and output it is denoted by ?. The comment in the block above therefore says that mix/2 partially evaluates the structure bound to its first argument and returns the suspended computation by binding it to the variable given as the second argument.

A comment about ‘index of predicates’

The index of predicates listed at the end of the thesis does not list all the predicates. In particular it only lists those predicates in the appendix described with a comment block and predicates described in the thesis but referred to in the appendix.

When a predicate is not given in the list of predicates its definition should be within a page or two of its usage. This is because all predicates that used outside their local scope (file) are commented.

\footnote{This argument naming convention is borrowed from GNU-Emacs.}
Appendix B

Interpreter for "full" Prolog

/*
 * A PROLOG META-INTERPRETER FOR "FULL" PROLOG
 *
 * The interpreter implements cut, meta-logical predicates (such as
 * not/1, call/1, ;/2) and the set predicates such as (setof/3,
 * bagof/3, findall/3)
 *
 * Though it uses the equivalent predicates of the native interpreter
 * to implement these features, it may be noticed that all
 * user-defined predicates are interpreted by this interpreter. The
 * interpreter is therefore of interest
 *
 * 1) for using as a skeleton for developing meta-programs such as
 *    a tracer and partial evaluator.
 * 2) when the "data" Prolog program being interpreted is not
 *    stored in the Prolog database but instead is saved as say
 *    $\$program(X,Y)" facts.
 *    Such a situation arises when an implementation inhibits or
 *    restricts the use of clause/2 predicate
 *
 * To aid in the second point the interpreter uses unfold_clause/2
 * (instead of clause/2) to get the definition of a clause from the
 * database. Appropriate definition of this clause should be provided
 * by the user.
 *
 * Predicates used by this interpreter and whose definitions may
 * depend on the underlying Prolog implementations are:
 *
 * is_interpreted/1, is_compiled../1, is_built_in/1, evaluate/1
 */
The behavior of these predicates may be obvious from their names and their usage. They may be interfaced with appropriate system predicates.

Other predicates used but not defined here are:

- `is_set_predicate/1` and `is_meta_predicate/1`

This predicates test if their arguments belong to the specific class of predicates. These predicates are easy to define; and are also defined elsewhere in the thesis.

```prolog
% prolog_goal(+GOALS).
% evaluate Prolog GOALS

prolog_goal(Goals) :-
solve_goal(Goals, Cut, AfterCut),
(Cut == !, !, solve_goal(AfterCut) : true).
```

% solve_goal(+GOALS, -CUT, -AFTERCUT)
% evaluates a sequence of GOALS till it either finds a CUT or the sequence is exhausted.
% In the former case it binds CUT to '!' otherwise to 'true'
% AFTERCUT are the conjunctive goals that follow the cut
% If '!' is the last atom in GOALS then AFTERCUT is bound to 'true'

```prolog
solve_goal(!, !, true) :-!.
solve_goal(!, !, AfterCut) :-!.
solve_goal((GoalA, GoalB), Cut, AfterCut) :-!
    solve_goal(GoalA, CutA, AfterCutA),
solve_goal(Goal, Cut, AfterCut) :-
    is_meta_predicate(Goal), !,
    solve_meta_goal(Goal, Cut, AfterCut).
solve_goal(Goal, true, true) :-
    is_set_predicate(Goal), !,
    solve_set_goal(Goal).
```
solve_goal(Goal, no, true) :-
    solve_atomic_goal(Goal).

solve_conjunct(+PREV_CUT, +PREV_AFTERCUT, +GOALS, -CUT, -AFTERCUT)
% processess '!' in a conjunction
% PREV_CUT is '!' if the conjunctive goals before GOALS has cut
% PREV_AFTERCUT is the sequence of goals after this cut but
% before GOALS
% If the previous goals had a cut then GOALS is not evaluated
% and is returned as AFTERCUT by conjuncting with PREV_AFTERCUT
% else it is evaluated

solve_conjunct(!, AfterCut, Goals, !, (AfterCut,Goals)) :-!.
solve_conjunct(_, _, Goals, Cut, AfterCut) :-
    solve_goal(Goals, Cut, AfterCut).

solve_atomic_goal(+GOAL)
% executes an atomic GOAL
% Only goals that can be interpreted are explicitly interpreted
% Built-in primitives or compiled goals cannot be interpreted
% and are hence evaluated directly by the system
% The execution is transparent to any cut encountered in the evaluation
% of GOAL

solve_atomic_goal(Head) :-
    is_interpreted(Head), !,
    unfold_clause(Head, Body),
    solve_goal(Body, Cut, AfterCut),
    (Cut = !,
     !,
     prolog_goal(AfterCut)
     ; true).
solve_atomic_goal(Goal) :-
    ( is_built_in(Goal)
     ; is_compiled(Goal)),
    evaluate(Goal).

solve_meta_goal(+META_PREDICATE, -CUT, +AFTERCUT)
% Interprets a META_PREDICATE
solve_meta_goal(call(X), Cut, AfterCut) :- solve_goal(X, Cut, AfterCut).
solve_meta_goal([A;B], Cut, AfterCut) :-
    solve_goal(A, Cut, AfterCut)
; solve_goal(B, Cut, AfterCut).
note { solve_meta_goal([A—B;C], Cut, AfterCut) :-
    solve_goal(A, CutA, AfterCutA)
    → (CutA = !,
        !,
        Cut = !,
        AfterCut = (AfterCutA —B;C)
        ; solve_goal(B, Cut, AfterCut))
    ; solve_goal(C, Cut, AfterCut).
}

solve_meta_goal(+X, no, true) :- \+ prolog_goal(X).

% solve_set_goal(+SET_PREDICATE)
% Interprets a SET_PREDICATE

solve_set_goal(findall(X,Y,Z)) :- findall(X, prolog_goal(Y), Z).
solve_set_goal(bago(X,Y,Z)) :- bagof(X, prolog_goal(Y), Z).
solve_set_goal(setof(X,Y,Z)) :- setof(X, prolog_goal(Y), Z).
Appendix C

ProMiX Partial Evaluation System

C.1 Kernel

C.1.1 Mix

/*
 * The core of the partial evaluator houses here.
 * Created afresh after phasing out old code that served almost 2
 * years.
 * It takes care of ALL control structures and predicates of
 * Prolog. Code for meta predicates and set predicates is in the
 * files set.pro and meta.pro
 */

:- current_op(X, Y, =), op(X, Y, /=).

/*
 * ENTRY POINTS
 * - drive_mix(+GOAL, CLAUSES)
 * - mix(+GOAL, -RESIDUE)
 *
 * Other entry points for experimenting
 *
 * The following returns one RESIDUE at a time. They differ on how
 * they handle cut and are needed in different contexts
 *
 * - mix_opaque(+GOAL, -RESIDUE)
 * - mix_opaque(+GOAL, -RESIDUE, +ANCESTORS)
 * - mix_transparent(+GOAL, -RESIDUE)
 * - mix_transparent(+GOAL, -RESIDUE, +ANCESTORS)
 */
The following returns a "bag" of all the RESIDUES. The parameter
HOWTOCUT tells how the cut is to be handled. It may be "opaque"
or "transparent"

- mix_bag(+HOWTOCUT, +GOAL, -RESIDUES)
- mix_bag(+HOWTOCUT, +GOAL, -RESIDUES, +ANCESTORS)

Entry point USED BY THE DRIVER FOR SPECIALIZING INTERPRETERS.
The driver performs one level of unfolding and passes the unfolded
CLAUSES.
- mix_clauses(HEAD, CLAUSES, RESIDUE, +ANCESTORS)

In all the above ANCESTORS is a list of atoms that are copies of
goals in the ancestral chain upto the given goal.
This list is generate internally. The form without ANCESTOR is
maintained to give an easy interface to when experimenting.

\[
\text{drive_mix}(+GOAL, -\text{MIXEDCLAUSE})
\]
A driver for mix for the specific purpose of specializing
interpreters. It performs one level of unfolding of the
interpreter GOAL. The other levels are suppressed by the
unfold rules
MIXEDCLAUSE is a clause from mixing GOAL. Only one clause is
returned at a time. Clauses due to hidden cut are returned too.

\[
drive\_mix(Goal, MixedClause) :-
\]
  is_interpreted(Goal),
  all_clauses(Goal, Clauses),
  copy_hook(Goal, CopyGoal),
  mix_clauses(Goal, Clauses, MixedBody, [CopyGoal]),
  filter_residue(MixedBody, ResBody, CutClauses-[]),
  member(MixedClause, [(Goal:-ResBody)|CutClauses]).

\[
\text{mix}(+GOALS, -\text{RESIDUE})
\]
partially evaluates GOALS and returns the suspended computation
as RESIDUE
Used for partially evaluating queries (like when testing or
demonstrating ProMiX)
mix(Goals, Residue) :-
    mix_opaque(Goals, Residue, []).

% mix_opaque(+GOALS, -RESIDUE, +ANCESTORS)
% partially evaluates GOALS and returns the suspended
% computation as RESIDUE.
% If GOALS contain 'cut' it is assumed to have local effect.
% i.e. with in the scope of GOALS only.
% If the goals before cut are evaluable it evaluates the cut ...
% .... find the res by playing with it. The RESIDUE does not
% contain any indication about the encounter with cut
% Is useful for handling the second cut in a body and cuts in
% meta predicates where its effect is transparent
% May be used handling top-level queries (mix/2)

mix_opaque(Goal, Residue) :-
    mix_opaque(Goal, Residue, []).

mix_opaque(Goal, Residue, Ancestors) :-
    mix_transparent(Goal, MixGoal, Ancestors),
    !,
    mix_rest(MixGoal, Residue, Ancestors)
    ; Residue = MixGoal
).

mix_rest(MixGoal, Residue, Ancestors) :-
    evaluable_cut(MixGoal, BeforeCut, AfterCut),!
    mix_opaque(AfterCut, ResAfterCut, Ancestors),
    merge_ands(BeforeCut, ResAfterCut, Residue).

mix_rest(MixGoal, Residue, Ancestors) :-
    extract_residue(MixGoal, Residue).

% mix_transparent(+GOALS, -RESIDUE, +ANCESTORS)
% partially evaluates GOALS and returns the suspended
% computation as RESIDUE.
% If a cut is encountered in GOAL it is evaluated to cut
% the choice points for conjunctions and an
% indication to this effect is also returned in the RESIDUE.

mix_transparent(Goals, Residue) :-
mix_transparent(Goals, Residue, []).

mix_transparent(Var, Residue, _Ancestors) :-
    var(Var), !,
    make_atomic_residue(call(Var), Residue).
mix_transparent(true, Residue, _Ancestors) :-
    !, make_atomic_residue(true, Residue).
mix_transparent((X, Y), MixXY, Ancestors) :-!
    mix_and(X, Y, MixXY, Ancestors).
mix_transparent(X, Residue, Ancestors) :-
    is_control_predicate(X), !,
    mix_meta(X, Residue, Ancestors).
mix_transparent(X, Residue, Ancestors) :-
    is_set_predicate(X), !,
    mix_set(X, Residue, Ancestors).
mix_transparent(Goal, Residue, Ancestors) :-
    mix_atomic(Goal, Residue, Ancestors),
    ( has_cut(Residue),
      !
    ; true
    ).

/* CONJUNCTIVE GOALS */

mix_and(X, Y, MixXY, Ancestors) :-
    ( var(X)
    ; is_side_effect(X)
    ; is_extra_logical(X)
    ),!
    mix_bag(transparent, X, MixX, Ancestors).
mix_bag(transparent, Y, MixY, Ancestors).
merge_ands(MixX, MixY, MixXY).

% no need to cut choice points since X cannot have a !
mix_and(GoalA, GoalB, MixAB, Ancestors) :-
mix_transparent(GoalA, MixA. Ancestors).
mix_conj(MixA, GoalB, MixAB, CutAnd, Ancestors),
  \% If one of the conjuncts has a cut, cut conjuncts
  ( CutAnd = cut_and, !
    ; true )
).

mix_conj(MixX, Y, MixXY, cut_and, _Ancestors) :-
  has_cut(MixX), !,
  merge_ands(MixX, Y, MixXY).
  \% The following is a hack to inhibit backtracking over residue
  \% containing hidden cut. Backtracking generates multiple
  \% identical "extension" clauses.

mix_conj(MixX, Y, MixXY, cut_and, Ancestors) :-
  has_hidden_cut(MixX), !,
  mix_bag(transparent, Y, MixY, Ancestors),
  merge_ands(MixX, MixY, MixXY),
  ( has_cut(MixXY)
    \rightarrow CutAnd = cut_and
    : CutAnd = no )
).

mix_conj(MixX, Y, MixXY, CutAnd, Ancestors) :-
  mix_transparent(Y, MixY, Ancestors),
  merge_ands(MixX, MixY, MixXY),
  ( has_cut(MixXY)
    \rightarrow CutAnd = cut_and
    ; CutAnd = no )
).

/\* ATOMIC GOALS  */

mix_atomic(!, Residue, _Ancestors) :-!,
  make_cut_residue(!, Residue).
mix_atomic(X, Residue, _Ancestors) :-
  immediately_evaluable(X), !,
  evaluate(X),
  make_atomic_residue(true, Residue).
mix_atomic(Read, MixB, Ancestors) :-
computation_is_looping(Head, Ancestors), !,  
MixB = Head.
mix_atomic(Head, MixB, Ancestors) :-  
is_interpreted(Head),  
should_unfold(Head), !,  
copy_hook(Head, Parent),  
all_clauses(Head, Clauses),  
mix_clauses(Head, Clauses, MixB, [Parent | Ancestors]).
mix_atomic(Goal, Residue, _Ancestors) :-  
make_atomic_residue(Goal, Residue).

哗 * CLAUSES *

mix_clauses(Goal, [(Head:-Body)|RestCl], Residue, Ancestors) :-  
copy_hook(Goal, Head),  
mix_transparent(Body, MixBody, Ancestors),  
(has_cut(MixBody),  
!, /* stop clause selection */  
commit_clause(Goal, Head, MixBody, RestCl, Residue, Ancestors)  
; Goal = Head,  
Residue = MixBody  
).
mix_clauses(Head, [X|RestCl], MixBody, Ancestors) :-  
mix_clauses(Head, RestCl, MixBody, Ancestors).

% % Processing of clause after encountering a cut done here  
commit_clause(Goal, Goal, MixBody, _RestCl, Residue, Ancestors) :-  
%
% if residue before cut is evaluable, commit this clause  
evaluable_cut(MixBody, BeforeCut, AfterCut), !.  
mix_opaque(AfterCut, ResAfterCut, Ancestors),  
merge_conjunctions(BeforeCut, ResAfterCut, Residue).

commit_clause(Goal, Head, WMixBody, RestCl, Residue, _Ancestors) :-  
%
% else stop partial evaluation and return  
% the partially evaluated clause and the set of clauses after it  
% *** This part would be improved in later versions ***  
extract_residue(WMixBody, Body),  
make_clauses_with_cut(Goal, [(Head:-Body)|RestCl], Residue).

% % BAG OF RESIDUE
Support for conditions when all the residues for a goal are required at the same time

Example: side-effect predicates, after cut, meta-predicates and set predicates

Three such predicates are defined. They differ on whether they perform any postprocessing on the RESIDUES

mix_bag(+HOWTOCUT, +GOALS, -RESIDUES, +ANCESTORS)
mix_bag_eval(+HOWTOCUT, +GOALS, -RESIDUES, +ANCESTORS)
mix_bag_evalX(+HOWTOCUT, +GOALS, -RESIDUES, +ANCESTORS)

HOWTOCUT can be "opaque" or "transparent". See description above about the two

mix_bag/4 returns RESIDUES without any postprocessing

mix_bag_eval/4 checks if the bag of residues is determinate (that is the bag have one element). If so it evaluates the RESIDUES if it is evaluable. The result of evaluation "true" or "fail" is returned as the RESIDUES. Otherwise the bag is returned

mix_bag_evalX/4 is the same as mix_bag_eval/4, except if the evaluation of the residue results in a "fail" it fails.

mix_bag(HowToCut, Goals, Residue) :-
mix_bag(HowToCut, Goals, Residue. []).

mix_bag(HowToCut, Goals, Residue, Ancestors) :-
variables_in(Goals, Vars),
mix_bag_x(HowToCut, Goals, Vars, AllR, Ancestors),
generate_unifications_for_bag(AllR, Vars, Residue).

mix_bag_x(opaque, Goals, Vars, AllR, Ancestors) :-
findall(sol(R,Vars),
    (nonvar(Vars).mix_opaque(Goals - R,Ancestors)),
   AllR).

mix_bag_x(transparent, Goals, Vars, AllR, Ancestors) :-
C.1.2 Partially evaluate meta-predicates

/* PARTIALLY EVALUATE META-PREDICATES call/1, :/2, ~/3, not/1 */
/* No amount of documentation would aid in understanding this part. */
/* I therefore leave you in silence. Comments would only clutter */
/* the text and your thought. */
/* */
/* But before I shut up: a tip for PORTING. */
/* NOT (like God) has various names. Some systems refer to it by */
/* not/1 and others by \\+. To make porting easy I have abstracted */
/* the structure from this code by using make_not/2. */
/* (see cross reference to locate where it is). I may probably move it */
/* around while restructuring the code */
/* */
% mix_meta(+GOAL, -RESIDUE, +ANCESTORS)
mix_meta(call(X), Residue, Ancestors) :-
  mix_transparent(X, Residue, Ancestors).
mix_meta(NotX, Residue, Ancestors) :-
  make_not(_X, NotX),
  mix_meta_not(NotX, Residue, Ancestors).
mix_meta((X—Y:Z), Residue, Ancestors) :-.
  mix_meta_if((X—Y:Z), Residue, Ancestors).
mix_meta((X:Z), Residue, Ancestors) :-
mix_meta_or((X,Z), Residue, Ancestors).

/* if you are curious. This code can be unfolded and optimized. I */
/* like it like this because it */
/* 1. allows one to cluster related code */
/* 2. gives direct access to predicates when spy-pointing, */
/* 3. should be possible to do the unfolding by self partial evaluation */
not /* mix_meta_not(+GOAL, -RESIDUE, +ANCESTORS) */
mix_meta_not(NotX, Residue, Ancestors) :-
  make_not(X, NotX),
  mix_bag(opaque, X, MixX, Ancestors),!
  eval_first_residue(MixX, EvalMixX),
  mix_not(EvalMixX, NotMixX),
  make_meta_residue(NotMixX, Residue).

mix_not(true, _):- !, fail.
mix_not(fail, Residue):- !, make_atomic_residue(true, Residue).

/* IF-THEN-ELSE (—) */

/* mix_meta_if(+GOAL, -RESIDUE, +ANCESTORS) */
mix_meta_if((X—Y;Z), Residue, Ancestors) :-
  mix_bag.eval(opaque, X, MixX, Ancestors),!
  /* local cut after X */
  mix_if(MixX, Y, Z, Residue, Ancestors).

mix_if(true, Then, _Else, Residue, Ancestors) :-
  !,
  mix_commit(Then, Residue, Ancestors).
mix_if(fail, _Then, Else, Residue, Ancestors) :-
  !,
  mix_commit(Else, Residue, Ancestors).
mix_if(MixX, Then, Else, MixXYZ, Ancestors) :-
  mix_bag(transparent, Then, MixY, Ancestors),
mix_bag(transparent, Else, MixZ, Ancestors),
merge_ifs(MixX, MixY, MixZ, MixXYZ).
merge_ifs(MixX, MixY, MixZ, Residue) :-
    ( has_cut(MixX)
    ; has_cut(MixY)
    ; has_cut(MixZ)
    ).!
make_cut_residue((MixX—MixY;MixZ), Residue).
merge_ifs(MixX, MixY, MixZ, Residue) :-
    make_meta_residue((MixX—MixY;MixZ), Residue).

% choose one of the two goals in mix_commit depending on how
% you like the residue for if when the condition is evaluable I
% like it split in many clauses. Note a preceding side-effect
% or .. predicate would override this anyway.
mix_commit(X, MixX, Ancestors) :-
    mix_transparent(X, MixX, Ancestors).
    % mix_bag_eval_r(transparent, X, MixX, Ancestors).

/*@ DISJUNCTION :/2 */

% mix_meta_or(+GOAL, -RESIDUE, +ANCESTORS)
mix_meta_or((X,Y), Residue, Ancestors) :-
    mix_bag_eval(transparent, X, MixX, Ancestors),
    mix_or(MixX, Y, Residue, CutOr, Ancestors).
    ( CutOr = cut_or,
    !
    : true)
    ; mix_bag_eval_r(transparent, Y, Residue, Ancestors).

mix_or(fail, _Y, _MixY, true, _Ancestors) :- !, fail.
mix_or(MixX, _Y, Residue, true, _Ancestors) :-
    evaluable_residue(MixX), /* but not fail */
    evaluate(MixX),
    !, make_atomic_residue(true, Residue).
mix_or(MixX, _Y, MixX, cut_or, _Ancestors) :-
    evaluable_cut(MixX, _Before, _After).
    mix_or(MixX, Y, MixXorY, cut_or, Ancestors) :-
    mix_bag(transparent, Y, MixY, Ancestors),
merge_ors(MixX, MixY, MixXorY).

merge_ors(MixXC, MixYC, Residue) :-
    ( has_cut(MixXC)
    ; has_cut(MixYC)
    ), !,
    make_cut_residue((MixXC;MixYC), Residue).
merge_ors(MixX, MixY, Residue) :-
    make_meta_residue((MixX;MixY), Residue).

C.1.3 Partially evaluate set-predicates

/* PARTIALLY EVALUATE SET PREDICATES: setof, bagof, findall
 * No comments, this code may be best understood by concentration.
 */

% mix_set(+SETPRED, -RESIDUE, +ANCESTOR)
% evaluates SETPRED if fully evaluable then computes the set
% operation at compile time. Otherwise the RESIDUE consists of a
% similar set operation on a partially evaluated goal.

mix_set(findall(X,Y,Z), Residue, Ancestor) :-
    mix_bag(opaque, Y, MixY, Ancestor),
    evaluate_set(MixY, findall(X,MixY,Z), Residue).

mix_set(bagof(X,V|Y,Z), Residue, Ancestor) :- !,
    mix_set_x(Y, MixY, NewVars, Ancestor),
    evaluate_set(MixY, bagof(X,(V,NewVars)|MixY,Z), Residue).

mix_set(bagof(X,Y,Z), Residue, Ancestor) :-
    mix_set_x(Y, MixY, NewVars, Ancestor),
    evaluate_set(MixY, bagof(X,NewVars|MixY,Z), Residue).

mix_set(setof(X,V|Y,Z), Residue, Ancestor) :-
    mix_set_x(Y, MixY, NewVars, Ancestor),
    evaluate_set(MixY, setof(X,(V,NewVars)|MixY,Z), Residue).
mix_set(setof(X,Y,Z), Residue, Ancestor) :-
    mix_set_x(Y, MixY, NewVars, Ancestor),
    evaluate_set(MixY, setof(X, NewVars|MixY.Z), Residue).

mix_set_x(Y, MixY, NewVars, Ancestor) :-
    mix_bag(opaque, Y, MixY, Ancestor),
    extra_variables(MixY, Y, NewVars).

% evaluate_set(+MIX, +SETPRED, -RESIDUE)

evaluate_set(MixY, SetPred, Residue) :-
    (evaluable_bag(MixY) ->
        SetPred,
        make_atomic_residue(true, Residue)
    ; make_meta_residue(SetPred, Residue)).
C.1.4 Procedures for manipulating residue

/ *
* The mix code is transparent to the structure of RESIDUE. This is
* achieved by abstracting the structure, and performing all the
* tests, creation, and manipulation of residue through routines
* provided here.
* *
* The benefits of this approach are obvious and need not be
* elaborated upon.
* *
** DATA ABSTRACTION FOR RESIDUE AND NOTES ON HANDLING CUT
* *
* A RESIDUE has one of the following structures
*
* `$$with_cut'(BEFORECUT, GOALS, AFTERCUT)
* `$$with_cut'(GOALS)
* `$$hidden_cut'(GOAL, CLAUSES)
* GOALS : Prolog Goals
* The `$$with_cut'/1,3 forms are used for residue of goals in the
* same clause body. They encapsulate GOALS that contain a `!' and
* have not been evaluable.
* *
* `$$with_cut'(BEFORECUT, GOALS, AFTERCUT) form is used for tracking
* cuts in a conjunctive sequence. BEFORECUT is the "residue" for goals
* before the GOALS containing cut. AFTERCUT are the goals AFTER the
* GOALS containing `!'.
* *
* The `$$with_cut'/1,3 structures are used to pass the information
* that a cut was found to the SCOPE of the `!' 
* *
* CUTTING GOALS
* *
* When the residue of a SUBGOAL `has_cut'/1 a `!' is executed to
* inhibit backtracking over previous goal and a residue `$$with_cut
* is returned to the outer levels. (See mix_and/2, mix_meta_if/2 etc)
* *
* CUTTING CLAUSES
* If the RESIDUE of a clause body has cut/1
  * if it is of the form "$$with_cut"/3
    * and BEFORE CUT is an evaluable_residue/1 during partial evaluation
      * and GOALS = "!", the clause selection is terminated
        * the presence of cut is forgotten
  * else
    * "we have in hand GOALS that has cut/1 that cannot be evaluated",
      * the "remains" of this clause and REST_CLAUSES are packed in a
      * "$hidden_cut"/2 structure and returned as RESIDUE
  *
  * RESERVED STRUCTURES
  *
  * The programs being partially evaluated should not contain the following
    * structures.
      * "$hidden_cut/2
      * "$with_cut/1,3
      * "$unifs/1 - used to remember unifications generated internally
  */

% make_atomic_residue(+GOAL, -RESIDUE)
% create RESIDUE for atomic GOAL

make_atomic_residue(Goal, Goal).

% make_meta_residue(+GOAL, -RESIDUE)
% create RESIDUE for a meta GOAL

make_meta_residue(MetaRes, MetaRes).

% make_clauses_with_cut(+GOAL, +CLAUSES, -HIDDEN_CUT)
% creates residue with HIDDEN_CUT when unfolding of GOAL unfolds
% a cut.

make_clauses_with_cut(Goal, Clauses, "$hidden_cut'(NewGoal,NewClauses)) :-
gensym(NewNum),
Goal =.. [Functor|Args],
append_atoms(Functor, NewNum, NewFunctor).
NewGoal =.. [NewFunctor|Args],

rename_clauses_heads(\text{Clauses, NewFunctor, NewClauses}).

rename_clauses_heads([], NewFunctor, []).
rename_clauses_heads([(H::B)|\text{Cls}], NewFunctor, [(\text{NewH}::B)|\text{NewCls}]) :-
  H =.. []|\text{Args},
  NewH =.. [NewFunctor|\text{Args}],
  rename_clauses_heads(\text{Cls}, NewFunctor, NewClauses).

:- dynamic \$\$gensym'/1.

gensym(Sym) :-
  (retract($$gensym'(X)); X = 0).!,
  Num is X+1,
  asserta($$gensym'(Num)),
  append_atoms('.extension..', Num, Sym).

% make_cut_residue(+GOALS, -WITH.CUT)
% creates a WITH.CUT structure with GOALS as residue

make_cut_residue(!, $$\text{with_cut'(true,!,true')}).
make_cut_residue((RX::RY;RZ), $$\text{with_cut'(\text{R}sX::\text{R}sY;\text{R}sZ)}) :-!,
  extract_residue(RX, RsX),
  extract_residue(RY, RsY),
  extract_residue(RZ, RsZ),
make_cut_residue((RX;RY), $$\text{with_cut'(\text{R}sX;\text{R}sY)}) :-!,
  extract_residue(RX, RsX),
  extract_residue(RY, RsY).

% merge_and(+RESIDUE1, +RESIDUE2, -RESIDUE)
% creates RESIDUE representing a conjunction of RESIDUE1 and RESIDUE2

merge_and(fail, _X, fail) :-!.
merge_and(MiRX, MiRY, _MiXY) :-
  has_cut(MiRX),
  has_cut(MiRY),
  fatal_error('merge_and/3 received both residues with cut').
merge_and($$\text{with_cut'(Before,Cut,After)}, MiRY,
$$\text{with_cut'(Before,Cut,AfterCut))}:-
!.
extract_residue(MixY, ResY),
merge_conjunctions(After, ResY, AfterCut).
merge_ands(MixX, "$with_cut'(Before,Cut,After),
    "$with_cut'(BeforeCut,Cut,After)):-

!,
extract_residue(MixX, ResX),
merge_conjunctions(ResX, Before, BeforeCut).
merge_ands(MixX, MixY, "$with_cut'(MixXandY)) :-
    ( has_cut(MixX)
    ; has_cut(MixY)
    ),!
extract_residue(MixX, ResX),
extract_residue(MixY, ResY),
flatten_conjunction((ResX,ResY), MixXandY).
merge_ands(MixX, MixY, MixXandY) :-
    flatten_conjunction((MixX,MixY), MixXandY).

/* SUPPORT PROCEDURES FOR HANDLING RESIDUE */

% has_hidden_cut(+RESIDUE)
% succeeds if RESIDUE had a hidden cut

has_hidden_cut("$hidden_cut'(X...Xs)).

% has_cut(+RESIDUE)
% succeeds if RESIDUE has cut

has_cut("$with_cut'(X)).
has_cut("$with_cut'(X...X)).

% extract_residue(+RES_STRUCTURE, -GOALS)
% extracts the GOALS in the RES_STRUCTURE.

extract_residue("$with_cut'(Goals), Goals) :-!
extract_residue("$with_cut'(Before,Cut,After), Goals) :-!
    flatten_conjunction((Before,Cut,After), Goals).
extract_residue(Goals, Goals).

% evaluable_cut(+RESIDUE, -BEFORECUT, -AFTERCUT)
% succeeds if conditions for evaluating a cut in the residue are
% satisfied. Returns the residue BEFORECUT and the goals AFTERCUT.

evaluable_cut(!, true, true).
evaluable_cut($with_cut'(BeforeCut, !, AfterCut), BeforeCut, AfterCut) :-
evaluable_residue(BeforeCut).

/\ SUPPORT TO PROCESS BAG OF RESIDUES /\

% generate_unifications_for_bag(+SOLUTIONS, +VARS, -BAGS)

generate_unifications_for_bag([], _Vars, fail) :- !.
generate_unifications_for_bag([Sol], Vars, Res) :- !.
\gx(Sol, Vars, Res).
generate_unifications_for_bag([Sol|Us], Vars, Bag) :-
\gx(Sol, Vars, Res).
generate_unifications_for_bag(Us, Vars, Ts),
merge_ors(Res, Ts, Bag).

% \gx(+SOLUTION, +VARS, -RESIDUE)
\gx(sol(Res, U), Vars, Residue) :- !.
generate_unifications(Vars, U, Unifications),
translate_to_unifs(Unifications, Unifs),
merge_ands(Unifs, Res, Residue).
\gx(Sol,...) :-
% A trap for errors during development */
\writeI(["Error: Incorrect structure '"Sol." to filter_residue_bag "]).

% generate_unifications(+VARS, +VALUES, -UNIFICATIONS)
% if the VALUE corresponding to a VAR is bound it generates a
% unification for it. If a variable the two variables are
% unified.

generate_unifications([], [], true).
generate_unifications([Var|XVars], [CopyVar|CVars], TUnifs) :-
  var(CopyVar), !,
  Var = CopyVar.
generate_unifications(XVars, CVars, TUnifs).
generate_unifications([Var|XVars], [CopyVar|CVars], Unifs) :-
generate_unifications(XVars, CVars, TUunifs),
merge_conjunctions([Var = CopyVar], TUunifs, Unifs).

% translate_to_unifs(+UNIFS, -UNIF_STRUCTURE)

translate_to_unifs(true, true):-.!
translate_to_unifs(Unifications, "$\$unifs$(Unifications)$).

"$\$unifs$(X) :- X. /* Just in case you wish to execute $\$unif/1 */

% evaluable_residue(+BAG)
% Tests when a BAG has residue that is evaluable. The conditions
% at present may be too strong. They may be relaxed after
% gaining proper experience.

evaluable_residue("$\$unifs$(\_X)$).
evaluable_residue(true).
evaluable_residue(fail).

% eval_unifs_if_determinate(+BAG, -RESIDUE)
% evaluates a BAG if determinate. The BAG itself is RESIDUE
% otherwise.

eval_unifs_if_determinate("$\$unifs$(Unifs)$, Residue) :-!.
call_hook(Unifs),
make_atomic_residue(true, Residue).
eval_unifs_if_determinate(Bag, Bag).

% evaluable_bag(+BAG)
% succeeds if all the residues in BAG are evaluable

evaluable_bag([R|Rs]) :- !.
evaluable_residue(R),
evaluable_bag(Rs).
evaluable_bag(R) :- evaluable_residue(R).

% eval_first_residue(+BAG, -RESIDUE)
% Hard to comment.
This is used in handling not/1.
May also be used for handling condition part of
if-then-else. (It isn't at present)

\emph{NOTE:} The following codes "knows" the structure of residue. If this
structure is changed, the code should be changed accordingly.

\begin{verbatim}
  eval_first_residue(tail, fail):-!.
  eval_first_residue(true, true):-!.
  eval_first_residue(true, Y, Residue) :-!,
      make_atomic_residue(true, Residue).
  eval_first_residue(Bag, Bag).
\end{verbatim}

\textit{/* SUPPORT TO EXTRACT HIDDEN CLAUSES CONTAINING CUT */}

\begin{verbatim}
  filter_residue(+RESIDUE, -FILTERED_RESIDUE, -CLAUSES)
  extract a difference list of CLAUSES left hidden in RESIDUE and
  return FILTERED_RESIDUE

  filter_residue(Residue, FResidue, Clauses-Tail) :-
      filter_residue(Residue, FResidue, Clauses. Tail).

  filter_residue(X, X, Tail, Tail) :- var(X),!.
  filter_residue("$\text{hidden\_cut}'(Goal, Clauses), Goal, Head, Tail) :-!,
      filter_hidden_residues(Clauses, Head, Tail).
  filter_residue("$\text{unify)'(X, Tail, Tail) :-!
      filter_residue((GoalB,GoalC), FGoalBC, Head, Tail) :-!,
      filter_residue(GoalB, FGoalB, Head, TailX),
      filter_residue(GoalC, FGoalC, TailX, Tail),
      merge_conjunctions(FGoalB, FGoalC, FGoalBC).
  filter_residue(MetaPred, FMetaPred, Head, Tail) :-
      meta_property(\textquoteleft MetaPred.
      MetaProperty\textquoteright ,!.
      functor(MetaPred, Functor, Arity),
      functor(FMetaPred, Functor, Arity),
      filter.m(MetaPred, MetaProperty, FMetaPred, 1, Arity, Head, Tail).
  filter_residue(Atom, Atom, Tail, Tail).
\end{verbatim}

\textit{/* filter_residue of the meta-arguments of meta-predicates */}
C.1.5 Rewrite

/*
 * Supports rewriting of interpreter goals in meta-predicates
 * as well
 * The actual rewrite rule is separated from the text so that
 * and taken as a parameter
 */

% rewrite(+PREDICATE, -RE_PREDICATE)
% translates the PREDICATE into RE_PREDICATE. All 'predicates' that
% arguments to the PREDICATE structure are translated recursively.

rewrite(Var, RVar) :-
  var(Var),!,
  rewrite_atom(Var, RVar).
rewrite(MetaPred, RMetaPred) :-
  meta_property(_, MetaPred, Property),!,
  functor(MetaPred, Functor, Arity),
functor(RMetaPred, Functor, Arity).
rewrite_m(Arity, MetaPred, Property, RMetaPred).
rewrite(Atom, RAtom) :-
  rewrite_atom(Atom, RAtom).

rewrite_atom(Atom, RAtom) :-
  expand_predicate(Atom, RAtom), !.
  rewrite_atom(Atom, Atom).

rewrite_m(0, _Pred, _Property, _RPred) :-!
rewrite_m(NthArg, Pred, Property, RPred) :-
  arg(NthArg, Pred, A),
  arg(NthArg, Property, P),
  arg(NthArg, RPred, F),
  N1 is NthArg - 1,
  (is_meta_argument(P) \n   -> rewrite(A, F)
   ; F = A),
  rewrite_m(N1, Pred, Property, RPred).

%! clean_rewrite_relation —
%! removes the rewrite relations memo-ed in the database.
C.2 Driver

/* ENTRY POINTS:
   mix(+FLAVOR, +FILE(S))
   mix(+FLAVOR, +FILE(S))
   Mixes FLAVOR to variable meta argument. Useful when the
   interpreters input is in two parts - one a Prolog
   program and the other some argument. (See palindrome and
dog for example).
*/

/***** MIXING TO A FLAVOR WITH VARIABLE ARGUMENTS ****/

mix(+FLAVOR, +FILE(S))

mixv(Flavor, DataFiles) :-
  consult_data_files(DataFiles),
  mix_to_flavor(Flavor, [Var]).

/***** INTERFACE FOR MIXING TO A LIST OF FILES ****/

mix(+FLAVOR, +FILES)

mix(_,Flavor, Files) :-
  \+ ground(Files),
  writeln(['ERROR: Variable given for a file or files']),
  !.fail.
mix(Flavor, Files) :-
  consult_data_files(Files),
  mix_x(Flavor, Files).
mix(_,Flavor, [ ]) :-!
mix(Flavor, [File|Files]) :- !,
  mix(Flavor, File),
  mix(Flavor, Files).
mix(Flavor, File) :-
preds_in_file(File, PredList-[ ]), !,
mix_to_flavor(Flavor, PredList).

mix_to_flavor(Flavor, PredList) :-
meta_goal(Flavor, IntGoal, MetaArg),
mix_and_output(IntGoal, MetaArg, PredList),
fail.
mix_to_flavor(_, Flavor, _PredList).

/*
* ENTRY POINTS
*
* TO DRIVE TESTS
* mixd(+GOAL) - display one residue
* mixb(+GOAL) - perform mix_bag and display residue
* TO DRIVE GENERAAL PROGRAMS
* peval(+GOAL, -RESIDUE) - Returns one Residue
* peval.procedure(+GOAL, -RESIDUES) - returns ALL RESIDUES
* TO DRIVE SPECIALIZATION OF INTERPRETERS
* pevalInt(+INT.GOAL. +META_ARG, +LANG_ATOMS, -RESIDUES)
* mix_and_output(+INT.GOAL, +INT_ARG, +LANG_ATOMS)
* */

/**** DRIVER TO RUN TESTS ****/

% mixd(+GOAL)
% partially evaluates GOAL and displays ALL Residual clauses

mixd(Goal) :-
peval(Goal, Clause),
display_clauses([Clause]),
fail.
mixd(_Goal).
/* Test what "BAG" means */
mixb(Goal) :-
mix_bag(transparent, Goal, Residue),
flatten_conjunction([Goal:-Residue], Cl),
display_clauses([Cl]).

/**** DRIVER FOR GENERIC GOALS ****/

% peval(+GOAL, -RESIDUE)
% partially evaluate GOAL and return one RESIDUE
% Others are returned on backtracking

peval(PevalGoal, ResidueClause) :-
drive_mix(PevalGoal, MixedClause).
FlattenedClause = MixedClause,
rewrite(FlattenedClause, ResidueClause).

% peval.procedure(+PREDICATE, -RESIDUES)
% generate all RESIDUES after partially evaluating PREDICATE

peval.procedure(PevalGoal, Head, Tail) :-
findall(NewClause, peval(PevalGoal, NewClause), NewProc),
make_diff_list(NewProc, Head-Tail).

/**** DRIVER FOR SPECIALIZING INTERPRETERS ****/

% peval.int(+INT.GOAL, +INT.ARG, +LANG.ATOMS, -RESIDUES)
% Partially evaluates INT.GOAL wrt all atoms in LANG.ATOMS
% and return result in a list of RESIDUES. One entry INT.GOAL has
% the variable INT.ARG at its meta-argument position.

peval.int(IntGoal, IntArg, LangAtoms, Residues) :-
peval.int(IntGoal, IntArg, LangAtoms, Residues, []).

peval.int(_, _, [], Head, Ihead).
peval.int(IntGoal, IntArg, [LAtom|LangAtoms], Ihead, Tail) :-
create_goal_to_peval(IntGoal, IntArg, LAtom, PevalGoal),
peval.procedure(PevalGoal, Ihead, Tail),
peval.int(IntGoal, IntArg, LangAtoms, Tail1, Tail).
% create_goal_to_peval(+INT.GOAL, +INT.ARG. +LAtom, -PEVALGOAL)

create_goal_to_peval(IntGoal, IntArg, LAtom, CopyIntGoal) :-
    copy_hook((IntArg,IntGoal), (LAtom,CopyIntGoal)).

% mix_and_output(+INT.GOAL, +INT.ARG, +LANG_ATOMS)
% same as peval_int/4 except it outputs the RESIDUES to the
% standard output

mix_and_output(IntGoal, IntArg, LangAtoms) :-
    clean_rewrite_relation,
    peval_int(IntGoal, IntArg, LangAtoms, NewProg),
    display_clauses(NewProg).
C.3 Knowledge for controlling unfolding

/\* META-KNOWLEDGE FOR COUNT META-INTERPRETER  */

/\* ABOUT THE INTERPRETER GOALS */

\% meta_goal(GOAL, METAARG)
\% GOAL is an interpreter goal, and METAARG is argument
\% that carries the object program structure.

meta_goal(count(Goal, Count), Goal).

/\* ABOUT THE SUPPORT PREDICATES USED IN THE INTERPRETER */

\% imperative(+PREDICATE)
\% PREDICATE used by the interpreter. These predicates are always
\% unfolded.
\% There is no imperative predicate in the count interpreter

\% export(+PREDICATE)
\% PREDICATE used by the interpreter. These predicates are never
\% unfolded. They are simply exported to the residue.
\% There is no export predicate in the count interpreter

\% immediate(+PREDICATE)
\% PREDICATE that can be completely evaluated immediately by the
\% Prolog engine (instead of unfolding).

/\* ABOUT THE LANGUAGE PROCESSED */

\% structure_decomposed(+TERM)
\% TERM is a structure part of the language accepted by the
\% interpreter.

structure_decomposed((..)).
structure_decomposed(true).

\% terminalAtom(+TERM)
% TERM is a structure that is a terminal atom of the language
% accepted by the interpreter.

terminal_atom(X) :- sys(X).

/\ FOR DRIVER TO SPECIALIZE INTERPRETER  */

% flavor(+FLAVOR, +IGOAL)
% FLAVOR is a name associate to an interpreter. IGOAL is a
% meta-goal of this interpreter.

flavor(count, count(_Goal_Count)).

% should_unfold(+GOAL)
% succeeds if a GOAL is instantiated enough to be unfolded

should_unfold(Goal) :-
    meta_goal(Goal, MetaArg),!
    ( structure_decomposed(MetaArg)
    ; terminal_atom(MetaArg)).

should_unfold(Goal) :- imperative(Goal),!.
should_unfold(Goal) :-
    \+ export(Goal),
    \+ immediate(Goal).

% computation_is_looping(+GOAL, +ANCESTORS)
% succeeds if GOAL "belongs" to the list of ANCESTORS
%
% The "belongs to" checks may be one of four tests given by
% Frank van Harmelen (frank@aipna.ed.ac.uk) in his thesis
% These are user selectable

computation_is_looping(_Goal, _Ancestor) :-
    get_value(loop_check_type, off),!,
fail.
computation.is_looping(Goal, Ancestor) :-
    computation.is_looping_x(Goal, Ancestor).

computation.is_looping_x(Goal, [Ans|Ancestors]) :-
    check_loop(Goal, Ans), !.
computation.is_looping_x(Goal, [Ans|Ancestors]) :-
    computation.is_looping_x(Goal, Ancestors).

% check_loop(+GOAL, +ANS)
% Loop determination checks for ProMIX

check_loop(Goal, Ans) :-
    get_value(loop_check_type, Type),
    check_loop(Type, Goal, Ans).

check_loop(unify, Goal, Ans) :-
    % Goal unifies with an Ancestor
    unifiable(Goal, Ans).

check_loop(instance, Goal, Ans) :-
    % Goal is more general than an Ancestor
    more_general(Goal, Ans).

check_loop(strict_instance, Goal, Ans) :-
    % Goal is a strict instance of an Ancestor
    more_general(Goal, Ans),
    \+ more_general(Ans, Goal).

check_loop(variant, Goal, Ans) :-
    % Goal is an alphabetic variant of an Ancestor
    instance_off(Goal, Ans),
    instance_off(Ans, Goal).

check_loop(less_general, Goal, Ans) :-
    % Goal is less general than Ancestor
    less_general(Goal, Ans).

% help_param(?VARIABLE, -HELPMESSAGE, -TESTFORVALUE, -DEFAULT)
help_param(loop_check_type).
["The variable "loop_check_type" controls the type of test"],
["performed to check if a GOAL *belongs* to the list of ANCESTORS"],
[""],
["The values it can be set to are"],
[""],
["* unify - GOAL unifies with ANS"],
["* instance - GOAL is more general than ANS"],
["* strict_instance - GOAL is strictly more general than ANS"],
["* variant - GOAL is a symbolic variant of ANS"],
["* off - loop checking is turned off"],
["* less_general - GOAL is less general than ANS (test purpose)"],
["* where ANS is some goal in the ANCESTORS list"]
],
X | (nonvar(X),
    member(X,[off,unify,instance,strict_instance,variant,less_general]))
),
instance % DEFAULT TEST
).

/* SUPPORT FOR TESTING, CREATING and MANIPULATING METAGOALS */

% is_meta_goal(?GOAL.)
% tests if GOAL is a metagol

is_meta_goal(MetaGoal) :-
    meta_goal(-Flavor, MetaGoal, _MetaArg).

% meta_goal(+FLAVOR, ?METAGOAL, ?METAARG)
% test or get the METAGOAL and its METAARGument corresponding to a
% flavor

meta_goal(Flavor, MetaGoal, MetaArg) :-
    flavor(Flavor, MetaGoal),
    meta_goal(MetaGoal, MetaArg).
plug_in_object_goal(+METAGOAL, +GOAL)  
Put GOAL as a meta-argument of METAGOAL

plug_in_object_goal(ProcGoal, PevalGoal) :-
is_meta_arg(ProcGoal, PevalGoal).

get_meta_argument(+METAGOAL, -GOAL)  
get the GOAL at the meta-argument position of METAGOAL

get_meta_argument(MetaGoal, ObjGoal) :-
is_meta_arg(MetaGoal, ObjGoal).

get_rest_arguments(+METAGOAL, -RESTARGS)  
Get the list of RESTARGS in METAGOAL except the meta argument.  
NOTE: Due to the use of delete, s the following may break if there  
is another argument structurally equivalent to the meta-argument.

get_rest_arguments(MetaGoal, RestArgs) :-
is_meta_arg(MetaGoal, MetaArg),
MetaGoal =.. [MetaArgs1],
delete_s(MetaArgs1, MetaArg, RestArgs).

get_goal_arguments(+GOAL, -ARGUMENTS)  
get ARGUMENTS of the object GOAL

get_goal_arguments(Goal, Args) :-
Goal =.. [Functor[Args].

is_meta_arg(+METAGOAL, -METAARG)  
test or get the METAARGument of a METAGOAL

is_meta_arg(MetaGoal, MetaArg) :-
meta_goal(MetaGoal, MetaArg).

create_goal_template(+METAGOAL, -GMETAGOAL)  
Given a METAGOAL it creates a template GMETAGOAL that has the  
same function and arity as METAGOAL. At its meta-argument  
positions it has a term with the same functor and arity as the
% meta-argument of METAGOAL. All other arguments of the meta-goal
% and its meta-argument are different and new variables

create_goal_template(MetaGoal, GMetaGoal):-
    get_meta_argument(MetaGoal, ObjGoal),
    generalize_term(MetaGoal, GMetaGoal),
    generalize_term(ObjGoal, GObjGoal),
    plug_in_object_goal(GMetaGoal, GObjGoal).
Appendix D

Technique Interpreters

D.1 Accumulator Technique

/∗ [INTERPRETER ENCODING ACCUMULATOR TECHNIQUE ∗/

acc(Goal, Result) :-
    init_acc(Init),
    acc(Goal, Init, Result).

acc((GoalA,GoalB), In, Out) :- !,
    acc(GoalA, In, Mid1),
    acc(GoalB, Mid1, Mid2),
    comp_after_conjunct((GoalA,GoalB), Mid2, Out).

acc(Goal, In, Out) :-
    evaluate_direct(Goal), !,
    call(Goal),
    comp_after_direct(Goal, In, Out).

acc(Head, In, Out) :-
    unfold_clause(Head, Body),
    comp_at_head(Head, In, Mid1),
    acc(Body, Mid1, Mid2),
    comp_after_atomic(Head, Mid2, Out).

/∗ SUPPORT PREDICATES ∗/

% evaluate_direct(+GOAL) —
% succeeds if the GOAL should be directly evaluated instead of by
% the technique interpreter. This provides a mechanism for
% excluding user-defined goals and system primitives from
% participating in the computation performed by a technique.
D.2 Propagate Context Up

/* INTERPRETER ENCODING CONTEXT-UP TECHNIQUE */

% context_up(+GOAL, -CONTEXT)
% Returns CONTEXT from executing GOAL.

context_up((GoalA,GoalB), Context) :- !,
    context_up(GoalA, FromA),
    context_up(GoalB, FromB),
merge_context_at_conjunct([GoalA, GoalB], FromA, FromB, Context).
context_up(Goal, Context) :-
evaluate_direct(Goal), !,
call(Goal),
context_after_direct_goal(Goal, Context).
context_up(Head, Context) :-
unfold_clause(Head, Body),
context_up(Body, FromBody),
context_after_atomic_goal(Head, FromBody, Context).

/* ABSTRACT PREDICATES USED BY THE TECHNIQUE */

% merge_context_at_conjunct(+GOALSAB, +FROMA, +FROMB, -CONTEXT) —
% merges the contexts FROMA and FROMB obtained from evaluating
% conjunctive GOALSAB and returns as CONTEXT.

% context_after_direct_goal(+GOAL, -CONTEXT) —
% computes the CONTEXT to be returned after evaluating a
% GOAL directly by the Prolog engine.

% context_after_atomic_goal(+GOAL, +FROMBODY, -CONTEXT) —
% computes the CONTEXT after evaluating the GOAL for a
% user-defined predicate. FROMBODY is the context obtained
% from evaluating the body of a clause of GOAL.

D.3 Propagate Context Down

/* INTERPRETER ENCODING CONTEXT-DOWN TECHNIQUE */

% context_down(+GOAL, +CONTEXT)
% carries the CONTEXT down while executing GOAL

context_down([GoalA, GoalB], Context) :- !,
    context_down(GoalA, Context),
    context_down(GoalB, Context).
context_down(Goal, _Context) :-
evaluate_direct(Goal), !,
call(Goal).
context_down(Head, Context) :-
    unfold_clause(Head, Body),
    push_context_down(Head, Context, NewContext),
    context_down(Body, NewContext).

/* ABSTRACT PREDICATES USED BY THE TECHNIQUE */

% push_context_down(+GOAL, +CONTEXT, -NEWCONTEXT) —
% computes a NEWCONTEXT from the CONTEXT received on unfolding a
% GOAL. The NEWCONTEXT is used for the evaluation of the body of
% the clause.
Appendix E

Program Composition

E.1 Compose Corresponding Clause

/* COMPOSE CLAUSES WHICH ARE ENHANCED FROM A GIVEN CLAUSE */

:- op(900,xfy,⇒).
:- op(900,xfy,⇐).
:- op(900, xfy, '¬').
:- op(900, yfx, '⊃').
:- op(900, yfx, '∀').

% compose_clauses(+PARENTCLAUSE, +JOINARG1, +JOINARG2, -RESULT)
% Joins the clauses JOINARG1 and JOINARG2 giving RESULT assuming
% they are enhancements of the clause PARENTCLAUSE

compose_clauses((_,Head0 :- Body0), (Head1 :- Body1), (Head2 :- Body2), Js) :-
    /* join the heads and bodies of the two clauses */
    join_head(Head1, Head2, JHead),
    join_body(Body0, Body1, Body2, JBody),
    /* convert the result to normal form */
    flatten_conjunction((JHead :- JBody), Jf),
    strip_true(Jf, Js).

% join_head(+HEAD1, +HEAD2, -JHEAD)
% joins the heads HEAD1 and HEAD2 of two clauses giving JHEAD
% joining the head means simply to fold the two given goals
% with respect to the join specification

join_head(Head1, Head2, JHead) :-
    JHead ← Head1 '∧' Head2.
% join_body(+PARENTBODY. +BODY1. +BODY2. -JOINEDBODY)
% Returns JOINBODY the result of joining BODY1 and BODY2, body of the
% clauses being joined.
% PARENTBODY is the body of the clause from which the operand clauses
% are enhanced

join_body(Var, Var, Var, Var) :-
    var(Var), !.
join_body(true, Body1, Body2, (Body1, Body2)) :- !.
join_body(Body0, Body1, Body2, (Prefix1, Prefix2, Composed, RestJoin)) :-

    split(Body0, B0, Rest0),
    align_inherited_goal(B0, Body1, Prefix1, Remains1),
    align_inherited_goal(B0, Body2, Prefix2, Remains2),
    split(Remains1, Goal1, Rest1),
    split(Remains2, Goal2, Rest2),
    compose_predicates(B0, Goal1, Goal2, Composed),
    join_body(Rest0, Rest1, Rest2, RestJoin).

% join_body_x(+PARENTBODY. +BODY1. +BODY2. -JOINEDBODY)
% same as join_body/6 but returns the BODY in normal form.
% is used internally for joining arguments of meta-predicates

join_body_x(Body0, Body1, Body2, Js) :-
    join_body(Body0, Body1, Body2, JBody),
    flatten_conjunction(JBody, J),
    strip_true(Jf, Js).

% align_inherited_goal(+PARENT. +OPERANDBODY. -PREFIX. -REMAINING)
% traverses down the OPERANDBODY in search of a goal
% which is either an enhancement, extension or is the same as the
% PARENT goal.
% Goals preceding the matching goal (as per above criterion) are
% returned as PREFIX goals. The trailing goals are returned as
% REMAINING goals
% If the OPERANDBODY is empty (ie just true), move_up_to fail.

align_inherited_goal(_, Pred, true, _, _) :- !, fail.
align_inherited_goal(Pred, (B, Body), true, (B, Body)) :-
inherited(Pred, B), !.
align_inherited_goal(Pred, (B, Body), (B, Prefix), Aligned) :-!
align_inherited_goal(Pred, Body, Prefix, Aligned).
align_inherited_goal(Pred, Goal, true, Goal) :-
inherited(Pred, Goal).

% inherited(+PRED, -GOAL)
% is a support utility for align_inherited_goal
% PRED is the parent goal being used for movement, and GOAL is a
% goal in body of operand clause.
% GOAL is inherited if it unifies with PRED, or if it is an
% enhancement or extension of PRED.

inherited(Pred, Goal) :-
  unifiable(Pred, Goal), !.
inherited(Pred, Goal) :-
  enhanced_or_extended(Pred, Goal), !.
inherited(Pred, Goal) :-
  is_meta_predicate(Pred),
  is_meta_predicate(Goal),
  functor(Pred, F, N),
  functor(Goal, F, N).

% compose_predicates(+PARENT, +OPREAND1, +OPREAND2, -RESULT)

compose_predicates(Goal0, Goal1, Goal2, Composed) :-
  is_meta_predicate(Goal0),
  compose_meta_predicate(Goal0, Goal1, Goal2, Composed), !.
compose_predicates(B0, Goal1, Goal2, Composed) :-
  enhanced_or_extended(B0, Goal1),
  Composed ← Goal1 ⊂ Goal2, !.
compose_predicates(B0, Goal1, Goal2, Composed) :-
  enhanced_or_extended(B0, Goal2),
  Composed ← Goal1 ⊂ Goal2, !.
compose_predicates(Goal, Goal, Goal, Goal) :-!.
compose_predicates(B0, Goal1, Goal2, (Goal1, Goal2)).

% enhanced_or_extended(?PARENT, ?PREDICATE)
% PREDICATE is either enhanced or extended from PARENT
enhanced_or_extended(Predicate₀, Predicate) :-
    Parent '➢' Predicate₀.
enhanced_or_extended(_, Predicate₀) :-
    '➢' '—' Predicate₀.

E.2 Compose Procedures

% compose_procedures(+PREDICATE₁,+PREDICATE₂)
% composes the procedures referred by PREDICATE₁ and PREDICATE₂
% wrt the join specification and enhancement knowledge asserted
% in the database.
% displays the result on the standard output device

compose_procedures(Predicate₀, Predicate₂) :-
    Parent '➢' Predicate₀,
    Parent '➢' Predicate₂,
    all_clauses(Predicate₀, Clauses₀),
    all_clauses(Predicate₁, Clauses₁),
    all_clauses(Predicate₂, Clauses₂),
    compose_corr_clauses(Clauses₀, Clauses₁, Clauses₂).

compose_corr_clauses([], [], []).
compose_corr_clauses([Clause₀|Clauses₀],
    [Clause₁|Clauses₁],
    [Clause₂|Clauses₂]) :-
    compose_clauses(Clause₀, Clause₁, Clause₂, JoinedClause),
    display_clauses(JoinedClause),
    compose_corr_clauses(Clauses₀, Clauses₁, Clauses₂).

% compose_mutants(+PREDICATE₁, +PREDICATE₂) —
% compose predicates PREDICATE₁ and PREDICATE₂, one or both of
% which may be mutations of a common predicate.
% The composition is performed with user interaction.
% The result is displayed on the standard output device
% compose_meta_predicates(+PARENT, +PREDA, +PREDB, -RESULT)

compose_meta_predicates(Parent, PredA, PredB, Result) :-
  var(Parent).
compose_meta_predicates(Parent, PredA, PredB, Result) :-
  functor(Parent, Functor, Arity),
  functor(PredA, Functor, Arity),
  functor(PredB, Functor, Arity),
  functor(Result, Functor, Arity),
  meta_property(_, Parent, Property),
  compose_meta_predicates_x(Arity, Parent, PredA, PredB, Result, Property).

compose_meta_predicates_x(0, _Parent, _PredA, _PredB, _Result, _Property).
compose_meta_predicates_x(NthArg, Parent, PredA, PredB, Result, Property) :-
  NthArg > 0, 
  arg(NthArg, Parent, P),
  arg(NthArg, PredA, A),
  arg(NthArg, PredB, B),
  arg(NthArg, Result, R),
  arg(NthArg, Property, Prop),
  (is_meta_argument(Prop)
   — join_body_x(P, A, B, R)
   ; compose_arguments(NthArg, Parent, PredA, PredB, R)),
  N1 is NthArg - 1,
  compose_meta_predicates_x(N1, Parent, PredA, PredB, Result, Property).

% compose_arguments(+ARGNO, +PARENT, +PREDA, +PREDB, -ARG) —
%   composes non-meta-arguments of meta-predicates. For instance,
%   the first and third arguments of testof/3
%   ARGNO is the argument number
%   PARENT, PREDA, PREDB are the respective predicates and
%   ARG the result of composing the ARGNOth argument of PREDA and
%   PREDB
Appendix F

Tracer: Result of composition

%% trace( +GOAL ).
%% trace the execution of GOAL.

trace(Goal) :-
    reset_dynamic_database,
    tracer(Goal, 0).

%% tracer( +GOAL, +DEPTH )
%% Actual predicate to trace the execution of GOAL
%% GOAL : The goal to be traced
%% DEPTH : is the depth at which the Goal is in the search tree

tracer(true, _Depth).
tracer([GoalA, GoalB], Depth) :-
    tracer(GoalA, Depth),
    tracer(GoalB, Depth).
tracer(Pred, Depth) :-
    Depth1 is Depth + 1,
    next_node_number(NewNodeNum),
    check_to_stop_retrying(NewNodeNum),
    callLand_fail_port_tracer(Pred, Depth1, NewNodeNum),
    tracer_exec_extend(Pred, Depth1),
    exit_and_redo_port_tracer(Pred, Depth1, NewNodeNum).

%% tracer_exec_extend( +GOAL, +DEPTH )
%% An extension for tracer/2
%% It implements trace commands 'true' and 'skip'
%% If there is no such user directive it executes the GOAL.

tracer_exec_extend(_, Pred, _Depth) :-
test_and_reset_unconditional_true.
!
tracer_exec_extend(Pred, Depth) :-
  test_and_reset_skip,
  !,
call(Pred).
tracer_exec_extend(Pred, Depth) :-
  tracer_exec(Pred, Depth),
  (true:retry_or_fail!, fail).

%% tracer_exec(+GOAL, +DEPTH)
%% executes an atomic GOAL

tracer_exec(Goal, Depth) :-
  sys(Goal),
  !,
call(Goal).
tracer_exec(Head, Depth) :-
  unfold_clause(Head, Body),
  tracer(Body, Depth).

%% Procedures to do the processing of work to be done at various ports. They also track what port a goal is at
%%
%% The clauses for call_land.fail.port.tracer and
%% exit_land.redo.port.tracer are order sensitive. They determine what
%% port ie. call/fail, exit/redo the goal is at.

%% callLand.fail.port.tracer(+GOAL, +DEPTH, +NODENUM)
%% does processing for call/fail
%% NODENUM is a unique node number associated to the goal

call_land.fail.port.tracer(Goal, Depth, NodeNum) :-
call_port_tracer(Goal, Depth, NodeNum).
call_land.fail.port.tracer(Goal, Depth, NodeNum) :-
  fail_this_node(NodeNum).
  fail_if_going_back_for_retry_or_fail,
  fail_port_tracer(Goal, Depth, NodeNum),
  fail.
exit_and_redo_port_tracer(+GOAL, +DEPTH, +NODENUM) :-
   exit_port_tracer(Goal, Depth, NodeNum).

exit_and_redo_port_tracer(Goal, Depth, NodeNum) :-
   fail_if_giving_back_for_retry_or_fail,
   redo_port_tracer(Goal, Depth, NodeNum),
   fail.

extensions for handling specific ports

call_port_tracer(Goal, Depth, NodeNum) :-
   wait_at_port_tracer(call, Goal, Depth, NodeNum).
fail_port_tracer(Goal, Depth, NodeNum) :-
   wait_at_port_tracer(fail, Goal, Depth, NodeNum).
exit_port_tracer(Goal, Depth, NodeNum) :-
   wait_at_port_tracer(exit, Goal, Depth, NodeNum).
redo_port_tracer(Goal, Depth, NodeNum) :-
   wait_at_port_tracer(redo, Goal, Depth, NodeNum).

wait_at_port_tracer(+PORT, +DEPTH, +NODENUM) :-
   Common processing for all the ports
   it checks if the port is reached while in the middle of
   a quasi_skip, or skip, or retry operation
   if so it does not wait at the port

wait_at_port_tracer(Port, Goal, Depth, NodeNum) :-
   resume_trace_at_spy_point(Goal),
   resume_trace_at_quasi_skip_node(NodeNum),
   process_command(Port, Goal, Depth, NodeNum),
   !,
   fail_if_giving_back_for_retry_or_fail.
wait_at_port_tracer(_, Goal, _Depth, _NodeNum).

reset_dynamic_database
   initializes the dynamic database at the before starting
%%% the trace

reset_dynamic_database :-
    initialize_nodenum(0),
    reset_control_flag.
Appendix G

Support Procedures

% copy(+TERM, -COPY)
%   creates a term that is a COPY of TERM.
%   COPY is identical to TERM except that it uses a different
%   set of variables. A variable in TERM is substituted by
%   a new variable in COPY.
%   It is an enhancement of the term/1 skeleton. It uses
%   accumulator technique to carry the dictionary of old variables
%   to new variables mapping.

copy(Term1, Term2) :-
    copy(Term1, Term2, [], _Dict).

copy(Var1, Var2, InDict, OutDict) :-
    var(Var1),
    put_in_dict(InDict, OutDict, Var1, Var2).

copy(Term1, Term2, InDict, OutDict) :-
    functor(Term1, Functor, Arity),
    functor(Term2, Functor, Arity),
    copy_x(Arity, Term1, Term2, InDict, OutDict).

copy_x(0, _Term1, _Term2, InDict, InDict).

copy_x(NthArg, Term1, Term2, InDict, OutDict) :-
    NthArg > 0,
    arg(NthArg, Term1, A1),
    arg(NthArg, Term2, A2),
    copy(A1, A2, InDict, MidDict),
    N1 is NthArg -1,
    copy_x(N1, Term1, Term2, MidDict, OutDict).

% put_in_dict(INDICT, OUTDIC, VAR1, VAR2)
% does the necessary dictionary manipulation for copy/2

put_in_dct(InDict, InDict, Var1, Var) :-
    put_in_dct_x(InDict, Var1, Var2), !.
put_in_dct(InDict, [Var1-Var2][InDict], Var1, Var).

put_in_dct_x([OldVar-MatchVar][Dict], Var1, MatchVar) :-
    OldVar == Var1, !.
put_in_dct_x([X][Dict], Var1, Var2) :-
    put_in_dct_x(Dict, Var1, Var2).

% copy_hook(+TERM, -COPY)
% Just an interface to copy/2. It gives the flexibility
% replacing one definition for another without changing the code
% and deleting other definitions.

copy_hook(Term, CopyTerm) :- copy(Term, CopyTerm).

% variables_in(+TERM, -SET)
% SET of variables in TERM. (The set is represented as a list)
% It is an enhancement of the term/2 skeleton.
% It uses accumulator technique to build the set of variables.

variables_in(Term, Set) :-
    variables_in(Term, [], _Set).

variables_in(Var, InSet, OutSet) :-
    var(Var),
    put_in_set(InSet, OutSet, Var).
variables_in(Term, InSet, OutSet) :-
    functor(Term, Functor, Arity),
    variables_in_x(Arity, Term, InSet, OutSet).

variables_in_x(0, _Term, InSet, InSet).
variables_in_x(NthArg, Term, InSet, OutSet) :-
    NthArg > 0,
    arg(NthArg, Term, _),
    variables_in(A, InSet, MidSet),
    N is NthArg - 1,
\text{variables}\_\text{in}\_\text{x}(N, \text{Term, MidSet, OutSet}).

\%( \text{put}_\text{in}\_\text{-set}(+\text{INSET, -OUTSET, +VAR}) \% \\
\text{does the necessary set manipulation for variables}\_\text{in/3} \%

\text{put}\_\text{in}\_\text{-set}(\text{InSet, InSet, Var}) :- \\
\text{put}\_\text{in}\_\text{-set}\_\text{x}(\text{InSet, Var}).!.
\text{put}\_\text{in}\_\text{-set}(\text{InSet, [Var|InSet], Var}).
\%
\text{put}\_\text{in}\_\text{-set}\_\text{x}(\text{OldVar|InSet}, \text{Var}) :- \\
\text{OldVar == Var}.!.
\text{put}\_\text{in}\_\text{-set}\_\text{x}([\text{X}|\text{Set}], \text{Var}) :- \\
\text{put}\_\text{in}\_\text{-set}\_\text{x}(\text{Set, Var}).

\%( \text{extra}\_\text{variables}(+\text{TERM1, +TERM2, -EXTRA\_VARIABLES}) \%
\text{EXTRA\_VARIABLES used in TERM1 but not in TERM2} \%

\text{extra}\_\text{variables}(\text{Term1, Term2, Difference}) :- \\
\text{variables}\_\text{in}(\text{Term1, Vars1}), \\
\text{variables}\_\text{in}(\text{Term2, Vars2}), \\
\text{difference}\_s(\text{Vars1, Vars2}, \text{Difference}).

\%( \text{ground}(+\text{TERM}) \%
\text{is TERM ground} \%
\text{The term/1 skeleton is a mutant of this program.} \%

\text{ground}(\text{Term}) :- \\
\text{functor}(\text{Term, _Functor, Arity}), \\
\text{ground}\_\text{x}(\text{Arity, Term}).
\text{ground}\_\text{x}(0, _\text{Term}).
\text{ground}\_\text{x}(\text{NthArg, Term}) :- \\
\text{NthArg} > 0, \\
\text{arg}(\text{NthArg, Term, A}), \\
\text{ground}(\text{A}), \\
\text{N1 is NthArg -1}, \\
\text{ground}\_\text{x}(\text{N1, Term}).
% is builtin(+PREDICATE) —
%  true if PREDICATE is a primitive of the Prolog implementation.

% is compiled(+PREDICATE) —
%  true if PREDICATE is compiled into native code. The definition
%  of such predicates is usually not accessible, even though
%  they are user-defined and not system primitives.

% is interpreted(+PREDICATE) —
%  true if PREDICATE is interpreted (not compiled). The definition
%  of such predicates is accessible using clause/2 primitive
%  on most implementations.

% sys(+PREDICATE) —
%  a PREDICATE is sys if its definition is not accessible. These are
%  system defined primitives and compiled primitives. On some
%  implementations the definition of an interpreted primitive may
%  also not accessible.
%  ‘true’ and conjunction are not considered sys/i predicates.

% meta_property(+TYPE, +PREDICATE, +PROPERTY)
%  if PREDICATE belongs to TYPE then PROPERTY describes the
%  property of its meta_arguments
%
% PROPERTY = predicate(P1, P2, ... Pn) where Pi is property of
% argument i. Pi may be ‘.’ for a non-meta argument ‘+’ for a
% meta-argument
% If an argument of a predicate/term can eventually be invoked as
% a Prolog predicate it is called a meta-argument

meta_property(meta, (+-)., (+;+)).
meta_property(meta, -(+-)., --(+,+)).
meta_property(meta, if(-+)., if(+,-)).
meta_property(meta, if(--)., if(+,-)).
meta_property(set, findall(-,-), findall(-,+,-)).
meta_property(set, bagof(-,-), bagof(-,+,-)).
meta_property(set, setof(-,-), setof(-,+,-)).
meta_property(mata, \(+.\), \+(+)).
meta_property(meta, not(_), not(+)).
meta_property(meta, call(_), call(+)).
meta_property(none, [(_,_)], [(_+, +)]. /* I used here setof/2 and bagof/2 */

% is_meta_argument(+PROPERTY)
% tests if the property of an argument of a meta-predicate is
% meta (as per the meta-property/3 table defined above).

is_meta_argument(+).

% is_meta_predicate(+PREDICATE)
% tests if PREDICATE is a 'meta' predicate (according to the
% meta-property/3 table defined above)

is_meta_predicate(Predicate) :-
    meta_property(meta, Predicate, _Property).

% is_set_predicate(+PREDICATE)
% tests if PREDICATE is a 'set' predicate (according to the
% meta-property/3 table defined above)

is_set_predicate(Predicate) :-
    meta_property(set, Predicate, _Property).

% make_not(+PREDICATE, -NOT.PREDICATE) —
% Given a PREDICATE it creates a structure NOT.PREDICATE that
% corresponds the negation of PREDICATE.
% Since different implementation uses different structure such as
% not or \+ for this purpose, it has been abstracted out from
% the implementation.
The mechanism to access the definition of a clause in various Prolog implementations is not standard. Some provide complete access to the definition of any user-defined clause, others provide restricted access, and still others do allow any access to a predicate's definition. Besides, functionally compatible implementations often use different predicates for the same task.

The code in this thesis revolves significantly around the meta-programming capabilities of Prolog. To make the code portable it has been made transparent of the meta-programming support of any specific implementation. This is achieved by introducing the notion of meta-program database. This is the database in which the programs to be manipulated (by a meta-program) is stored.

When a system provides primitives to access the definition of predicates in the Prolog database, the meta-program database may be the same as the Prolog database. Otherwise a virtual program database as defined in Chapter 5 may be used to store and access information from this database.

The following primitives provide interface to meta-program database. These may be defined depending on the implementation.

reconsult_data_files(+FILES) —

Given a list of FILES it loads its contents in the meta-program database such that it is accessible using the unfold_clause/3 predicate.

Previous definitions, if any, of the procedures just loaded are removed.

unfold_clause(+HEAD, -BODY) —

Matches HEAD with the head of a clause in the meta-program database.
Returns suitably instantiated BODY of the first clause matched.
BODY is bound to ‘true’ if the clause is a fact.
Finds the next match on backtracking.
% coll_hook(+GOAL) —
% Executes the GOAL as a Prolog goal with respect to the program in
% the meta-program database.

% all_clauses(+HEAD, -CLAUSES) —
% Returns a list of all CLAUSES in the meta-program database whose
% heads unify with HEAD.
% CLAUSES is a list of (H:-B) structures, where H is the head of
% the clause after instantiating with HEAD and B is its body.
% B is bound to 'true' if the clause is a fact.

/* PREDICATES TO MANIPULATE A CLAUSE */

% flatten_conjunction(+CONJUNCTS, -FLATTENED_CONJUNCTS) —
% CONJUNCTS may be a clause or the body of a clause.
% FLATTENED_CONJUNCTS is semantically equivalent to CONJUNCTS
% except that nested conjunctions, if any, are flattened.

% strip_true(+CONJUNCTS, -STRIPPED_CONJUNCTS) —
% CONJUNCTS may be a clause or the body of a clause.
% FLATTENED_CONJUNCTS is semantically equivalent to CONJUNCTS
% except that all 'true' appearing in conjunction with other
% goals are removed.
% A conjunction of only 'true's is replaced with only one true.

% split(+BODY, -GOAL, -REST) —
% splits BODY such that it is a conjunction of GOAL and REST.
% If BODY is an atomic goal then GOAL is unified with BODY and
% REST with true.

% merge_conjunctions(+X, +Y, -XANDY) —
% If X and Y are goals XANDY is their conjunction. XANDY is in
% canonical form such that it does not have redundant 'true'

/* SOME I/O PREDICATES */

% fatal_error(+MESSAGE) —
% displays the MESSAGE on the standard output device and aborts
% the execution.
% writeln(+LIST) —
% writes the elements of the LIST (in order) on the standard
% output. Terminates the writing by a newline.
%
% display_clauses(+CLAUSES) —
% displays a list of CLAUSES on the standard output. The output
% is pretty-printed. It is output in a form which is re-readable
% by Prolog.
%
% preds_in_file(+FILE, -DIFF_PREDs_IN_PROGRAM) —
% gets a difference list DIFF_PREDs_IN_PROGRAM of predicates
% defined in a FILE.

/* PREDICATES TO INSTANCES */
%
% instance_of(+TERM1, +TERM2) —
% TERM1 is an instance of TERM2
%
% more_general(+TERM1, +TERM2) —
% TERM1 is more general than TERM2
%
% less_general(+TERM1, +TERM2) —
% TERM1 is less general than TERM2
%
% unifiable(+TERM1, +TERM2) —
% TERM1 is unifiable with TERM2. The terms are actually not
% unified.
%
% /=(+TERM1, +TERM2) —
% TERM1 does not unify with TERM2

/* MANAGING GLOBAL VARIABLES USING SIDE-EFFECTS */
%
% get_value(+GLOBAL_VARIABLE, -VALUE) —
% the VALUE for variable GLOBAL_VARIABLE is accessed from the
% database and returned
% fails if there is no value defined for this variable.
% set_value(+GLOBAL_VARIABLE, +VALUE) —
% the value of GLOBAL_VARIABLE is set to VALUE. Any subsequent
% get_value/2 (till next set_value/2) for this variable will
% return VALUE
% Previous value, if any, for this variable is removed
% Values of other variables are not altered
% Variables are maintained using side-effecting predicates

/* SOME GENERAL PURPOSE SUPPORT PREDICATES */

% append_atoms(+ATOM1, +ATOM2, -ATOM12) —
% create ATOM12 by concatenating the atoms ATOM1 and ATOM2.
% Thus if ATOM1 is 'abc' and ATOM2 is 'xyz' then ATOM12 is
% 'abcxyz'.

% make_diff_list(+LIST, -DIFF_LIST) —
% DIFF_LIST is a structure HEAD-TAIL such that HEAD is an
% incomplete list consisting of the elements of LIST terminated
% by the variable TAIL.

% member(?ELEMENT, +LIST) —
% ELEMENT unifies with a member of the LIST.

% member_s(+ELEMENT, +LIST) —
% ELEMENT is 'structurally equivalent' to an element of the list.

% delete_s(+LIST, +ELEMENT, -NEWLIST) —
% deletes the first element of LIST that is structurally
% equivalent to ELEMENT. NEWLIST is the remaining list.
% Succeeds even if there is no match. Returns unaltered LIST as
% NEWLIST.

% difference_s(+LIST1, +LIST2, -DIFFERENCE) —
% Given LIST1 and LIST2 returns DIFFERENCE a list of structures
% that appear either in LIST1 or LIST2 (not both).
% Only elements of LIST1 that are structurally equivalent to some
% element of LIST2 (and vice-versa) are not included in
% the list DIFFERENCE.
This page was deliberately left blank.
Bibliography


273


[37] A. Goldberg and D. Robson. *SMALLTALK-80 The Language and its Implementation*. Addison-Wesley, 1983.


Index of Predicates

\( =/2, 270 \)
align.inherited.goal/4, 255
all.clauses/2, 269
append.atoms/3, 271

call_and.fail.port_tracer/3, 260
call_hook/1, 269
check_loop/2, 246
check_to_stop_retrying/1, 203
clean_rewrite_relation/0, 230
comp_after_atomic/3, 251
comp_after_conjunct/3, 251
comp_after_direct/3, 251
comp_at_head/3, 251
compose_arguments/5, 258
compose_clauses/4, 254
compose_meta_predicates/4, 258
compose_mutants/2, 257
compose_predicates/4, 256
compose_procedures/1, 257
computation_is_looping/2, 215
context_after_atomic.goal/3, 252
context_after_direct.goal/2, 252
context_down/2, 252
context_up/2, 251
copy/2, 263
copy_hook/2, 264
create_goal_template/2, 248
create_goal_to_peval/4, 213

delete_s/3, 271
difference_s/3, 271
display_clauses/1, 270
drive_mix/2, 220
enhanced_or_extended/2, 258
eval_first_residue/2, 236
eval_unifs_if_determinate/2, 236
executable_bag/1, 236
executable_cut/3, 235
executable_residue/1, 236
evaluate_direct/1, 250
executable/1, 126
exit_and_redo_port_tracer/3, 261
expand_predicate/2, 103
export/1, 214
extra_logical/1, 88
extra_variables/3, 265
extract_residue/2, 234
fail_if_going_back_for_retry_or_fail/0, 203
fail_this_node/1, 203
fatal_error/1, 269
filter_residue/3, 237
flatten_conjunction/2, 269
flavor/2, 245
generalize_term/2, 106
generate_unifications/3, 235
generate_unifications_for_bag/3, 235
get_goal_arguments/2, 248
get_meta_argument/2, 248
get_rest_arguments/2, 248
get_value/2, 270
ground/1, 263
has_cut/1, 234
has_hidden_cut/1, 234
help_param/4, 217
immediate/1, 241
sys/1. 266

terminal_atom/1. 245
test_and.reset_skip/0. 201
test_and.unconditional_true/0. 201
trace/1. 259
tracer/2. 259
tracer.exec/2. 260
tracer.exec.extend/2. 259

unfold_clause/2. 268
unifiable/2. 270

variables.in/2. 264

wait.at.port_tracer/3. 261
writeln/1. 270