Autonomous Learning
by
Incremental Induction and Revision

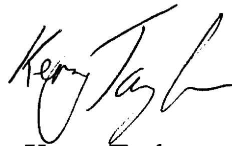
A thesis submitted for the degree of
Doctor of Philosophy

Kerry Lea Taylor

February 1996

THE AUSTRALIAN NATIONAL UNIVERSITY

This thesis is my own original work except where due acknowledgement is made in the text.



Kerry Taylor
4 February 1996

For my father, James Maxwell Taylor.

Acknowledgements

I first acknowledge the contribution to this thesis dissertation made by my triumvirate of supervisors. To Dr Claude Sammut of the University of New South Wales goes the credit for the initial inspiration for the work. He sparked my interest in machine learning as an area of study, and suggested the framework for incremental theory revision incorporating inverse resolution and declarative diagnosis. Claude directed the early stages of my research and focused my attention on experimental results in the final stages. In between, his deep knowledge of the area of work enabled him to suggest fruitful lines of enquiry for problems that arose.

I thank Professor Robin Stanton, Dean of the Faculty of Engineering and Information Technology at the Australian National University, for my education in the wider fields of artificial intelligence and computational logic. His broad and extremely perceptive view of my work and its context were challenging, thought-provoking and ultimately beneficial to its quality.

Dr Graham Williams, of the Division of Information Technology, CSIRO, was an invaluable source of practical assistance. Graham encouraged a regular dialogue of communication on a wide range of subjects: from the very particular to the very general. He also read and re-read interminable drafts, advised on detailed questions of style and expression and frequently drew my attention to helpful literature and research tools. I am especially grateful to Graham for making himself so readily available for assistance when I needed it.

I am also grateful to the people of the Department of Computer Science at the Australian National University for the provision of research facilities and a friendly working environment. Particular credit for this must go to Robin for setting the tone. But academic staff, research staff, support staff and post-graduate students too numerous to name individually have all made the place a happy and rewarding one in which to work.

The friendship of Ross Parker was especially valuable to me.

Most of all, I am grateful for the unfailing support and encouragement given freely by my husband, Trevor Vickers. Trevor was also a great help with matters of written expression, organisation, direction and equanimity.

I thank the youngest participant in the project: my son, Byron Vickers. Byron's remarkable capacity to learn was a source of wonder and inspiration. His demand for teaching was a source of frequent but welcome distraction.

The imminent birth of my second son, Darwin Vickers motivated the conclusion of the work and the commencement of a whole new teaching and learning project!

Abstract

An inductive learning capability is generally acknowledged to be crucial to artificial intelligence in machines. This work investigates learning by an autonomous agent that continuously interacts with the environment in which it is embedded. A novel domain-independent learning algorithm is proposed.

Learning is viewed as the incremental revision of a developing theory of multiple interdependent concept definitions. When an observation of the environment reveals an error in the present theory, the reason for the error is located by diagnosis and experimentation. To correct an error, alternative generalizing or specializing revisions are investigated. Revision hypotheses are evaluated by experimentation leading to further theory improvement by discovery.

The theory itself is the source of domain-dependent background knowledge directing hypothesis formulation. The environment is the source of observations of concepts and the testbed for hypothesis experimentation. The environment also generates unexpected interruptions imposing arbitrary time limits on learning.

The learning model is developed in the Inductive Logic Programming framework in which logic programs are used to represent theories. New theoretical results based on logic programming concepts underpin the key components of generalization by absorption and specialization by generalization of exceptions. The investigation of alternative revisions proceeds by a staged heuristic search through partially developed hypotheses, enabling graceful performance degradation when time is short.

The algorithm is implemented as a software agent called MINERVA. The environment is modelled by a parameterised random sampling tool called SAMPLER. Empirical results demonstrating the learning performance of MINERVA in several domains modelled by SAMPLER are reported.

Contents

Acknowledgements v

Abstract vii

Contents ix

Figures xvii

1 Introduction **1**

1.1 The Problem 1

1.2 Research Framework 2

1.3 Role of the Learner 2

1.3.1 The learner in the environment 4

1.4 Key Features 4

1.4.1 First order predicate calculus knowledge representation 5

1.4.2 Domain independence 6

1.4.3 Incremental revision in response to incremental input 6

1.4.4 Learning from background knowledge 7

1.4.5 Multiple interspersed concept learning 7

1.4.6 Simple input language 7

1.4.7 Active experimentation 8

1.4.8 Interruptibility for anytime learning 9

1.4.9 Theoretical foundation 9

1.5 Structure of the Thesis 10

2 Learning Illustrated **11**

2.1 Introduction 11

2.2 Discovering New Facts 13

2.3 Learning from Background Knowledge 14

2.4 Returning to an Earlier Concept 15

2.5 Interruption 15

2.6 Learning a Symmetric Relation 16

2.7 Learning Exceptions 17

2.8 Removing Redundant Clauses 18

2.9 Diagnosis 19

2.10 Generalizing an Exception 20

2.11 Missing Answer Diagnosis 21

2.12 Replacing a Redundant Rule 21

2.13 Simplifying Exceptions 24

2.14 Summary 26

3 Foundation Concepts **27**

3.1 Introduction 27

3.2 Inductive Concept Learning 27

3.2.1 Learning in the Inductive Logic Programming framework 28

3.3 Notation 29

3.4 Preliminary Concepts of Logic Programming 30

3.4.1 Terms 30

3.4.2 Clauses and programs 31

3.4.3 Substitutions 31

3.5 Preliminary Concepts of Inductive Logic Programming 32

3.5.1 Generality models 32

3.5.2 Generalization 35

3.5.3 Predicate invention 40

3.6 Interpreters 41

3.6.1 Negation as failure 42

3.6.2 Interpreter deficiencies 43

3.7 Declarative Diagnosis 52

3.7.1 Diagnostic tools 52

3.7.2 Contradiction backtracing 53

3.7.3 Missing answers 54

3.8 Summary 56

4 Incremental Learners and Theory Revision Systems **57**

4.1 Introduction 57

4.2 Incremental ILP Learners 57

4.2.1 Marvin 57

4.2.2 Mis 58

4.2.3 Cigol 59

4.2.4 Clint 59

4.3 Batch ILP Learners 60

4.4 Other Revising Learners 61

4.4.1 Knowledge base refinement 61

4.4.2 Operator planning 62

4.4.3 Science 63

4.5 Approaches to Revision Problems 64

4.5.1 Diagnosis 64

4.5.2 Generalization 64

4.5.3 Specialization 66

4.6 The AGM Logic of Belief Revision 67

5 The Learning Algorithm **69**

5.1 Introduction 69

5.2 Design Principles 69

5.2.1 Strategy outline 69

5.2.2 Redundancy 72

5.2.3 The top-level algorithm 72

5.3 Revision 74

5.3.1 Specialization by exception 74

5.3.2 Generalization 76

5.4 Interpreting a Program 77

5.4.1 The language of programs 78

5.4.2 Interpreters 79

5.5 Diagnosis 80

5.5.1 Questions 84

5.5.2 Search order 84

5.5.3 Invented predicates 85

5.5.4 Non-terminating diagnosis 85

5.5.5 Interrupted diagnosis 86

5.6 Experiments 87

5.6.1 Experiments about self-recursive predicates 89

5.6.2 Experiments about invented predicates 90

5.7 Redundancy 91

5.7.1 Sleep 92

5.8 Summary 93

6 Heuristic Search for a Revision **95**

6.1 Introduction 95

6.2 Heuristic Evaluation of a Revision 96

6.3 Value Guided Simplification 97

6.4 Value Guided Specialization 97
6.4.1 Choosing a specializing revision 98
6.4.2 Assimilating a specializing revision 100
6.5 The Search for a Generalization 100
6.5.1 Partial hypotheses 101
6.5.2 Development of a partial hypothesis 102
6.5.3 Hypothesis syntax checking 106
6.5.4 Hypothesis testing 107
6.6 Evaluating a Partial Hypothesis 109
6.6.1 Value of a partial hypothesis 109
6.6.2 Best background clause 112
6.7 Hypothesis Assimilation 112
6.7.1 Adequacy 113
6.7.2 Assimilating a partial hypothesis 113
6.8 Minimum Description Length 113
6.9 Summary 114

7 Modelling the Environment **115**
7.1 Introduction 115
7.2 The Target 115
7.3 Sample Strategies 116
7.3.1 Probability distributions 116
7.3.2 Deterministic strategies 117
7.3.3 Complexity-probability strategies 118
7.3.4 Arity-probability strategies 119
7.3.5 Positive strategies 120
7.3.6 Alternating strategies 120
7.3.7 User-control of example selection 121
7.4 Target Program Interpreters 121

7.4.1 Testing ground atoms 121
7.4.2 Generating ground atoms 122
7.5 Time Limits 122
7.6 Other SAMPLER features 123
7.6.1 Sleep control 123
7.6.2 Statistics 123
7.7 Summary 123

8 Performance evaluation 125

8.1 Introduction 125
8.1.1 Experimental design 125
8.2 Learning by Absorption 126
8.2.1 Results 127
8.3 Incremental Learning 128
8.3.1 Results 129
8.4 Theory Refinement 129
8.4.1 Experimental method 131
8.4.2 Induction experiment 131
8.4.3 Refinement experiment 132
8.5 Learning Multiple Predicate Theories 134
8.5.1 Results 135
8.6 Learning with MINERVA 135
8.6.1 The graph domain 135
8.6.2 Learning interleaved interdependent predicates 138
8.6.3 Varying the learning time 140
8.6.4 Varying the sleep pattern 141
8.6.5 Tolerating noise 142
8.6.6 Learning from positive examples 143
8.6.7 Confusing the background knowledge 144

8.6.8 Extending the problem 145

8.7 Summary 147

9 Generalization by Absorption **149**

9.1 Introduction 149

9.2 Absorption 150

9.2.1 Features of absorption 152

9.3 Completeness of Absorption 154

9.3.1 Preliminary definitions 154

9.3.2 Inversion of SLD-resolution 155

9.3.3 Completeness of absorption for definite clauses 160

9.4 Eliminating Redundancy 162

9.4.1 Connex clauses 162

9.4.2 Duplicate clauses 164

9.4.3 Least general absorption 164

9.5 Reducing the Search Space Further 165

9.5.1 Free variables in a background clause 165

9.5.2 Choosing the root 165

9.6 Changing the Representation 166

9.6.1 Flattening for absorption 167

9.6.2 Limiting unit absorption 167

9.7 A Strategy for Generalization by k-unit Absorption 170

9.7.1 Free variables in a background clause 170

9.7.2 Choosing the root 171

9.7.3 Avoiding inverse substitution 171

9.8 Generalizing Normal Clauses 172

9.8.1 Normal subsumption 172

9.8.2 Normal absorption 174

9.8.3 Difficulties with normal absorption 176

9.9 The Strategy for Generalization in MINERVA 177

9.9.1 Implementing normal absorption 177

9.9.2 Implementing k-unit absorption 178

9.10 Summary 179

10 Conclusions **181**

10.1 Introduction 181

10.2 Summary of the Thesis 181

10.3 Improvements to MINERVA 182

10.4 Expanding the Environment 185

10.5 Concluding Remarks 187

Bibliography **189**

Figures

1.1 Architecture of an intelligent agent 3

1.2 Modelling the learning environment 5

2.1 Example families 12

5.1 Top level algorithm 73

5.2 Procedure best-revision 75

5.3 Missing answer diagnosis 81

5.4 Missing answer diagnosis (continued) 82

5.5 Contradiction backtracing diagnosis 83

6.1 Choosing a specializing revision 99

6.2 Processing partial hypotheses 102

6.3 Stages of development of a partial hypothesis 103

6.4 Developing a partial hypothesis 104

6.5 Applying absorption 105

6.6 Hypothesis testing 108

8.1 Experimental results for induction 132

8.2 Experimental results for refinement 133

8.3 Graph relations domain 136
8.4 Graph relations program 137
8.5 Learning three graph relations 139
8.6 Varying the learning time 140
8.7 Varying the sleep pattern 141
8.8 Tolerating noise 143
8.9 Learning from positive examples 143
8.10 Confusing the background knowledge 144
8.11 Learning five graph relations 145
8.12 Learning a single extra graph relation 146

9.1 Absorption as inversion of an SLD-derivation 156

1

Introduction

1.1 The Problem

How do intelligent beings learn? How is it possible for an agent to use observations of an environment to reason about events which are not observed; to predict the consequences of actions; to form an internal model of its world?

These questions have been asked many times in many fields of human endeavour. Behavioral psychologists like Piaget have studied childhood learning [Flavell 1987, Ginsburg and Opper 1979]. Philosophers including Hume, Popper and Peirce have attempted to explain the concepts they have called *induction* and later *abduction* as well as scientific theory formation [Gregory 1987]. Computer scientists, recognizing the importance of learning to intelligent behaviour in man-made systems, have studied *machine learning*.

The study of machine learning is usually focused on one or more of three major goals. The cognitive science approach aims to discover and reconstruct models of human behaviour as a way of gaining greater insight into the workings of the human brain [Van-Lehn 1990]. The second goal, encompassing most machine learning research, deals with "knowledge discovery": the formation of predictive rules from a large but finite number of empirical observations. The third major goal is to equip robots with the ability to learn from their environment as they work in it. This would enable them to function effectively and adaptively in environments which are not highly restricted and well understood in advance by the human designers of the robots. Thus an off-the-shelf robot could adapt to different working environments or a special-purpose robot could work in dangerous or remote locations about which little is known.

The primary aim of this thesis is to contribute to this latter goal, although it is motivated by a fascination with the remarkable learning ability of humans. Popper's

account of induction, which provides a clue to the human ability to learn from a very small number of observations is particularly influential:

> Without waiting, passively, for repetitions to impress or impose regularities upon us, we actively try to impose regularities upon the world. We try to discover similarities in it, and to interpret it in terms of laws invented by us. Without waiting for premises we jump to conclusions. These may be discarded later should observation show that they are wrong. [Popper 1969b]

The thesis describes an algorithm which might underpin the learning component of an "intelligent" active learning agent. When executed, the algorithm interacts in a restricted but well-defined manner with a simulated *environment*. It demonstrates the ability to acquire a predictive *theory* of the environment. In the course of developing the theory, the algorithm performs *experiments*, thus actively seeking learning experiences. It is *interruptible* — always aiming to learn as much as possible as soon as possible, and assimilating the best of what has been learnt when interrupted. When observations indicate errors in the developing theory, the algorithm *revises* the theory to correct them. It can develop useful, if sometimes incorrect, inductive hypotheses from very few observations, enabling it to continue to work in pursuit of other goals while waiting for new experiences from which to learn. This algorithm is called MINERVA. An extended example of its behaviour is given in chapter 2. Comprehensive experimental results are reported in chapter 8.

1.2 Research Framework

The thesis is developed within the framework of *Inductive Logic Programming* (ILP). In this framework knowledge acquired by machine learning is represented as a *logic program*. The name was coined by Muggleton [1991] to refer to the study of the automatic construction of logic programs from examples of the behaviour of the (unknown) programs. He describes the framework to be at the "intersection" of the fields of machine learning and *logic programming*. ILP research draws heavily on the results of logic programming research which is concerned with understanding the representation and interpretation properties of logic programs. It benefits from the combination of the formal foundations of logic programming and the experimental approach of machine learning. Chapter 3 includes an introduction to the basic principles and techniques of ILP and logic programming and chapter 4 describes several ILP and other learners.

1.3 Role of the Learner

MINERVA is a component of a learning agent: figure 1.1 places MINERVA within a possible architecture for an intelligent agent. MINERVA is responsible for developing the knowledge structure which may be used by other knowledge-based components of the agent. The knowledge is modelled as a *theory* — a set of facts held to be true or false.

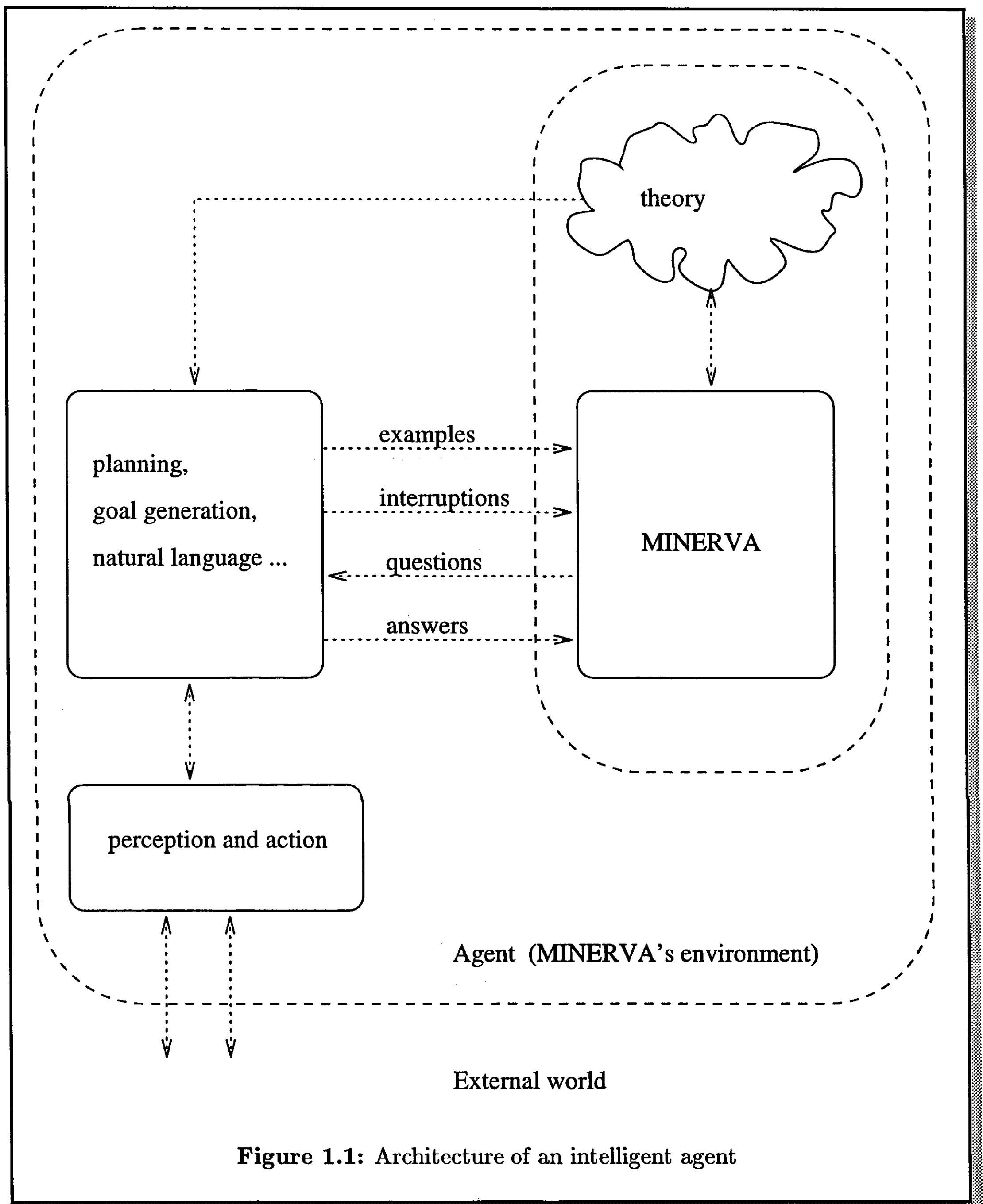


Figure 1.1: Architecture of an intelligent agent

MINERVA includes an *interpreter* which translates from the internal representation of a theory as a *program*.

MINERVA receives input, or *observations* in a formal language and performs experiments by asking *questions* in the same formal language. We say that the observations, responses to questions, and interruptions come from the *environment*. In a practical implementation they will be sourced from the external real world, provided by teachers,

natural events and the actions of the agent. They will be interpreted through the agent's sensors and reasoning components before presentation to MINERVA in the formal language. Similarly the formal language of experiments produced by MINERVA will be interpreted by the agent's reasoning components to formulate real world experiments, observations, and occasional natural-language questions.

Goals for learning are externally generated: by the environment or by another component of the agent and communicated to MINERVA by way of observations and interruptions. These other reasoning components of the agent could have access to the theory developed by MINERVA to aid their reasoning.

1.3.1 The learner in the environment

Provided that the environment presents occasional examples of concepts, learning proceeds autonomously through experiments designed by MINERVA. These experiments are self-directed observations of the environment and, subject to time constraints, MINERVA will perform as many as necessary to aid the search for inductive hypotheses. The experimental results (answers to questions) are assumed to be inexpensive to obtain as they are satisfied by observations of the environment in which MINERVA is embedded.

In this work the environment is modelled by a novel software tool called SAMPLER, suitable for interaction with an arbitrary co-operative learner. SAMPLER is primarily responsible for drawing facts from a target theory to present as examples to a learner; for providing the observations which correspond to the results of a learner's experiments; and for enforcing time limits on learning. These actions can be modulated by user-controlled parameters.

Figure 1.2 illustrates the role of SAMPLER in modelling the environment for MINERVA and chapter 7 describes SAMPLER in more detail. The empirical evaluation of MINERVA's learning performance in chapter 8 uses SAMPLER for environmental modelling.

1.4 Key Features

The major contribution of this thesis is a strategy for revision, incorporating both generalization and specialization, that is suitable for a learner having the following key features. Taken together these features distinguish MINERVA from other learning algorithms.

1. First order predicate calculus knowledge representation

2. Domain independence

3. Incremental revision prompted by incremental input

4. Learning from background knowledge

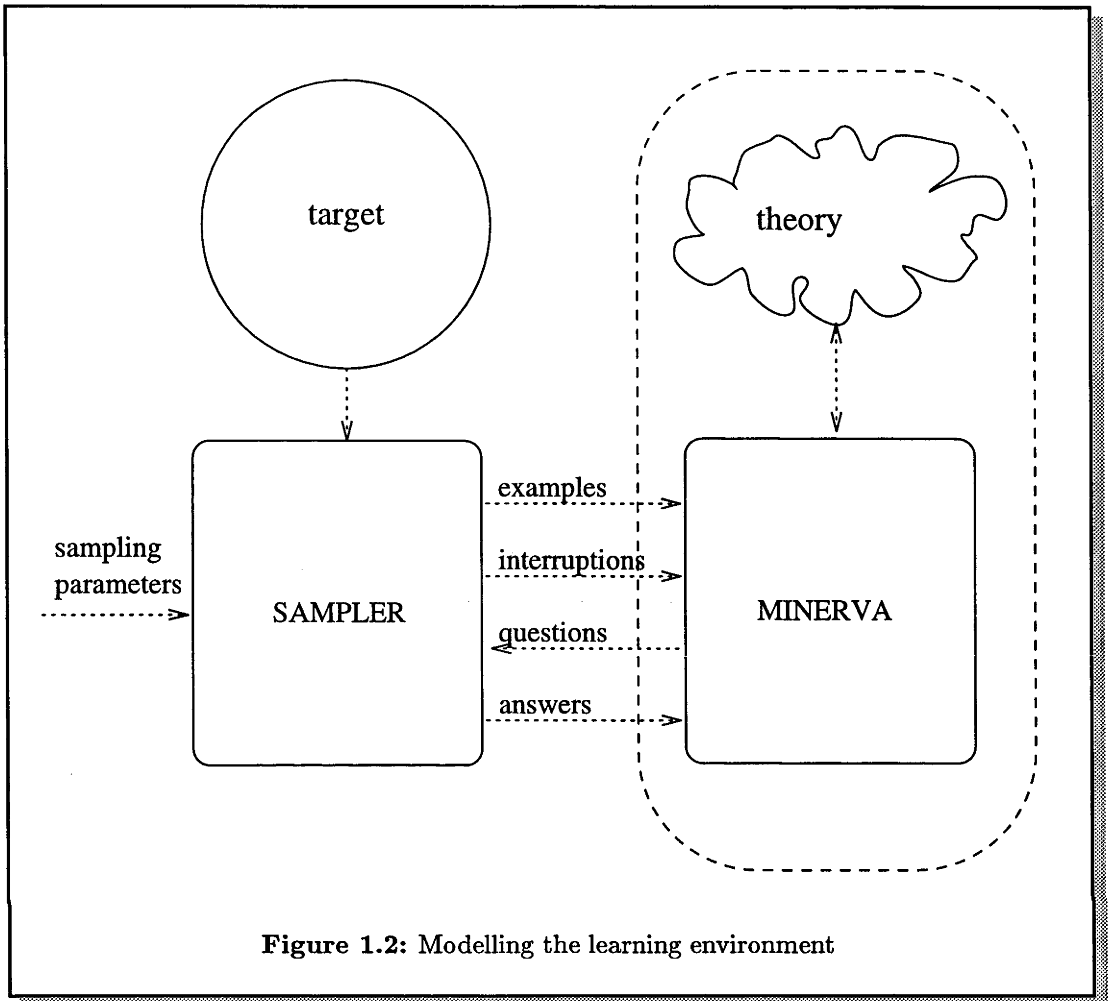


Figure 1.2: Modelling the learning environment

5. Multiple interspersed concept learning
6. Simple input language
7. Active experimentation
8. Interruptibility for anytime learning
9. Theoretical foundation

1.4.1 First order predicate calculus knowledge representation

In keeping with the ILP framework the theory acquired by MINERVA is represented by a logic program. The term "logic program" is applied to a wide range of representations in ILP, loosely meaning a set of universally quantified Horn clauses. The program structure of MINERVA is particularly expressive compared to other ILP learners. In particular, MINERVA supports function symbols, negation (not restricted to ground atomic predicates), non-determinate clauses, existential variables in clauses and direct

and indirect recursion. There are some syntactic restrictions on full normal programs: there may be only one negative literal in the body of a clause and that literal has a predicate symbol which occurs in no other clause body; and every variable in the head of a clause also occurs in the body. Nevertheless, the language of MINERVA is considerably more complex (and expressive) than is customary. The expressive power of MINERVA's representation is the source of three problems addressed in the thesis: an increased space of possible inductive hypotheses; undecidability of logical entailment; and the logical and computational difficulties associated with negation-as-failure.

A precise characterisation of MINERVA's knowledge representation language is given in chapter 5.

1.4.2 Domain independence

Like many other learners MINERVA is domain-independent. There are no prior assumptions about the terms of the language of observation: it is generated entirely by the environment, all terms having equivalent status. The environment is only assumed to be able to provide input of the syntactic form required by the learner and to truthfully answer questions presented in the same syntactic form.

1.4.3 Incremental revision in response to incremental input

Most experimental work in machine learning, and particularly in the ILP framework, assumes that the input is presented as a *batch*. That is, the input to a learner is a *finite set* of observations or examples. This enables a learner to make learning decisions with simultaneous consideration of their effect on the full input.

This model is appropriate when the goal of the learning algorithm is to discover patterns in the input, but it is inappropriate for a long-lived learner operating in a complex environment. We would like MINERVA to be able to jump to inductive hypotheses from partial information, even a single example, so that the agent can use the partial theory to make decisions and to guide further learning. This requires modelling input as a (possibly infinite) *sequence* of examples, allowing repetitions.

The term *incremental* is usually applied to this notion: of input being a sequence rather than a set. However, it is clear that a batch learner can be made incremental by simply allowing it access to one example at a time, having it keep a record of every example, and having it re-learn a theory from the complete example set each time a new example is supplied. Similarly, an incremental learner can be viewed as a batch learner by giving it all the input at one time and allowing it to order it internally.

Finite memory availability for a long-lived learner prevents it from permanently recording all observations. Time constraints suggest it cannot revise a theory to account for an error by removing the entire theory already constructed and starting again from the observations made thus far. Such a learner is forced to value the "epistemic entrenchment" of the theory it has already made even when it is known to be faulty, and to

adjust that theory to account for the observed errors.

It is a tenet of this thesis that an incremental learner working with time limitations must incrementally revise a partial theory when an error is exposed. This requires diagnosis to blame a part of the theory for causing the error, followed by revision to correct the error. A suitable strategy for revision, incorporating both generalization and specialization, is a major contribution of this thesis. It is described in chapter 5.

1.4.4 Learning from background knowledge

The term *background knowledge* refers to information about the learning problem which is not directly represented in the input. The need for learning mechanisms to take account of background knowledge has been apparent for a long time. This recognition leads immediately to the desire to empower a learning agent to learn its own background knowledge [Sammut 1981b, Stepp, Whitehall and Holder 1988, Russell 1989].

MINERVA implements this notion by having a uniform knowledge representation for any initially-provided background knowledge and the knowledge it acquires by learning. Inductive hypotheses constructed by MINERVA are assimilated into its theory and become available for use in further learning as background knowledge for the next learning task. MINERVA uses ILP techniques, particularly *inverse resolution* to support learning from background knowledge. Chapter 9 presents theoretical results, especially regarding soundness and completeness properties, that underpin the inverse resolution operator used in MINERVA.

1.4.5 Multiple interspersed concept learning

Unlike most learners, MINERVA can learn a theory about more than one concept. Examples of each concept may be interspersed in the input, although some sequence orders will enable better results than others. MINERVA constructs concept descriptions in terms of other concepts represented by input examples.

Some of the special difficulties of multiple concept learning are discussed by De Raedt, Lavrač and Džeroski [1993].

1.4.6 Simple input language

In common with most ILP and many other incremental learners the input to MINERVA is a sequence of simple facts, each classified as true or false. This places minimal demands on the environment: other incremental learners require prior declarations of the names of all concepts and objects, mode or other bias declarations, relevance annotations, strong ordering constraints, or answers to "difficult" questions. This information can be very helpful for learning when it is available, but MINERVA makes few demands on its environment in order to support autonomous learning. Nevertheless learning with a benevolent teacher is also possible and usually beneficial.

MINERVA does require that the environment is consistent at all times. The truth status of any fact must never change. A consequence of this is that *noise* in the input is not supported: MINERVA makes the *perfect data assumption* [Brazdil and Clark 1990] in order to enable deeper investigation of other aspects of the revision problem. Despite this, the design of MINERVA lends itself naturally to the incorporation of noise handling mechanisms.

1.4.7 Active experimentation

Rather than passively waiting for examples of concepts to come along, we would like a learner to actively experiment. A learner endowed with this capability is known as *active*.

The role of experimentation in childhood learning was identified by Piaget [Ginsburg and Opper 1979]. Experimentation is also fundamental to scientific theory formation [Klahr 1994]. Indeed, Popper's [1969a] philosophy of science demands that scientific inquiry proceeds by attempting to discover counter-examples to unlikely but explanatory hypotheses.

Experiments can be used to distinguish between alternative hypotheses, each of which is satisfactory with respect to experience. They can also be used to diagnose and correct mistakes in what has already been learnt. Established techniques for debugging logic programs are useful for this purpose.

Many learning systems assume the existence of an *oracle* or a *teacher* which can answer questions asked by the learner. Alternatively, these questions could be seen to represent the carrying out of an experiment or the making of a particular observation by an autonomous learner, in which case the answer is assumed to be given by the environment.

MINERVA asks questions to aid diagnosis and revision, described in chapter 5. We prefer the experimental/environmental view of the questions because for a learner embedded in an environment such directed observations could be inexpensively answered. In practice they could be answered by a combination of a teacher and the environment: the capabilities required of the question-answerer and the example-giver are the same. The questions asked by MINERVA are of a very simple form: is some fact true or false? The fact is expressed in the same language in which the environment has expressed examples. The environment must give either a yes or no response and that response must be consistent with earlier examples and answers.

However, experimentation cannot normally be exhaustive. The environment naturally imposes restrictions on the time and material resources available. Some difficulties with constructing experiments, including availability of materials and suitability of the current state of the environment, are beyond the scope of the environmental model used here. These issues are addressed by Cheng [1991] and Hume and Sammut [1991a].

Many learners ask questions which are more expressive than those of MINERVA. These

questions tend to be more demanding of the environment. The questions of MINERVA are at least as easy as those of any other active learner. This means that the information gained from answers is limited and the learning task is harder for MINERVA than for other learners which ask more expressive questions.

Angluin [1990] shows that theoretically, allowing a learner to ask questions improves the learning time and example complexity performance of a learner. Indeed, if questions for which the learner can correctly guess the answer are regarded free of cost, example complexity performance is remarkably improved [Goldman and Sloan 1994]. As explained by Shapiro [1981], asking questions need not affect the theoretical *completeness* of the learner provided that the environment is assumed to supply all possible examples eventually (or, more precisely, an *enumeration* of the environment).

1.4.8 Interruptibility for anytime learning

MINERVA is designed to be a learning component of an agent which usually functions to satisfy goals unrelated to learning. When the agent is unable to satisfy a goal because of an observed failing of its knowledge, MINERVA is invoked. MINERVA considers methods to correct the error in a best-first manner, allowing it to be interrupted at any moment without warning. It is unimportant whether the interruption is generated externally by an environmental occurrence or teacher command, or whether the agent's own goals require a shift of attention. The interruption is assumed only to happen unexpectedly and beyond the control of MINERVA.

MINERVA's support for interruption in this way is unique amongst ILP learners. Generally, intelligent agent components designed to improve results with increased time availability are called *anytime algorithms* [Dean and Boddy 1988, Poole 1993].

The major challenge of anytime learning is the allocation of time in the best possible manner. MINERVA's consideration of alternative revisions supports incremental improvement: easily evaluated revisions are investigated before more complex alternatives and information gained during the evaluation directs the following investigation. The best revision determined at the time of interruption is made and MINERVA awaits further input. Chapter 6 describes how this is done.

1.4.9 Theoretical foundation

The key components of MINERVA's learning strategy are analysed in a formal manner in the thesis, especially in chapter 9. Drawing on results from both machine learning and logic programming research, particular attention is paid to the formal meaning of generalization and to establishing a completeness property for the generalization strategy employed in MINERVA.

1.5 Structure of the Thesis

The succeeding chapter illustrates the behaviour of MINERVA by way of an extended annotated example of a learning sequence.

Basic concepts from machine learning and logic programming, needed for discussing the learning problem in depth, are introduced in chapter 3. Chapter 4 surveys other relevant research work to provided a context for the work of this thesis.

Chapter 5 outlines the architecture of MINERVA and deals in depth with interpretation, diagnosis, experimentation and redundancy. Chapter 6 describes the integration of these techniques under a heuristic search control strategy in MINERVA.

A tool for environmental modelling called SAMPLER is described in chapter 7. This tool is used in chapter 8 to empirically evaluate the performance of MINERVA: the evaluation procedure is described and the results of empirical tests reported and analysed.

Chapter 9 provides a supplementary theoretical definition and analysis of key components of MINERVA underlying the generalization strategy.

Chapter 10 concludes the work with a comment on its achievements and the opportunities it creates for further progress in machine learning.

2

Learning Illustrated

2.1 Introduction

In this chapter the learning behaviour of MINERVA is illustrated by an extended example based on family relationship concepts. The example represents a complete, unedited, learning sequence from a starting point devoid of background knowledge. It is intended to give the reader an idea of the power and range of the capabilities of MINERVA before presenting them in more depth. Some formal terminology is used informally in this chapter before definition in the succeeding chapter. Some readers might prefer to read chapter 3 before this one.

Figure 2.1 describes the family tree for the three families of the example. A generally descending line connects a mother or a father to each of his or her offspring. The families are not conventional, but they have been chosen to demonstrate particular features of MINERVA in a small domain.

The order of presentation and timing of the examples is also deliberate. Normally these would be controlled by the environment but the user-guided presentation given here enables a demonstration of MINERVA's major features. Normally, too, the hypotheses would be verified by experiments in the environment rather than questions posed to a user; all experiments are indicated here to aid the explanation.

Notation

When MINERVA is idle, awaiting input, the prompt mark ">" is indicated. Examples presented to MINERVA as observations are written as an atom prefixed by "+" for a positive example (true atom) or "−" for a negative example (false atom), and

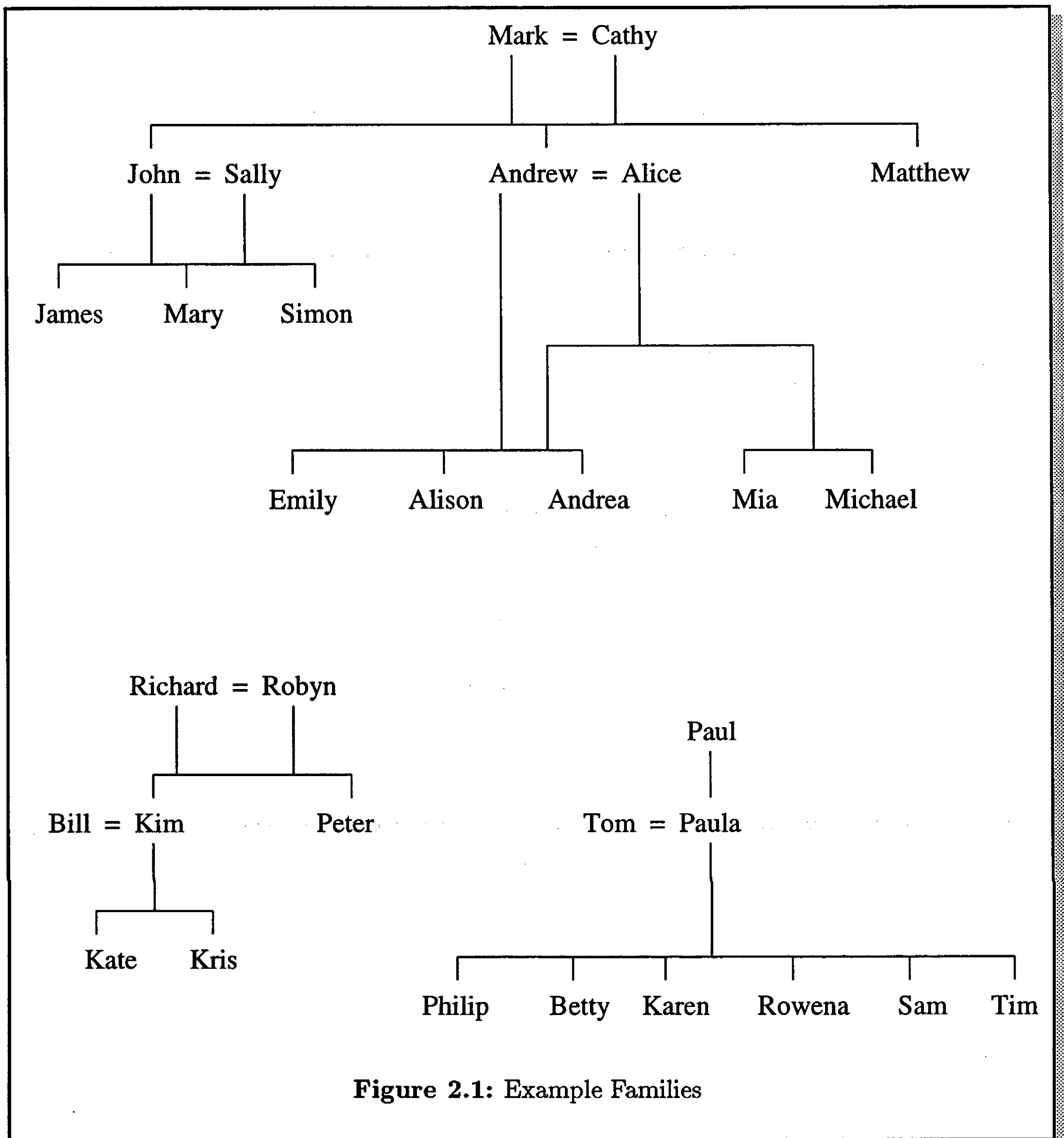


Figure 2.1: Example Families

terminated with a full stop. When MINERVA asks a question it is written as an atom preceded by a question-mark ("*?*") and followed by a full stop. The answer provided by the environment appears as "*y*" for yes or true, or "*n*" for no or false. Sometimes a hypothesis being considered by MINERVA is noted for illustrative purposes within the dialogue as a clause preceded by the mark "......", although it would not normally be displayed by MINERVA. When the deliberations of MINERVA are not permitted to continue to completion but are interrupted, this is indicated by an exclamation mark ("*!*").

2.2 Discovering New Facts

Initially the program representing MINERVA's theory is empty. The first few examples observed provide no opportunity for learning anything beyond the examples themselves, which are simply adopted as unit clauses in the program.

> *+person(john).*
> *+person(mary).*
> *+person(peter).*
> *+person(james).*
> *+person(simon).*
> *+person(cathy).*
> *+father(john, mary).*

The last example does provide an opportunity for the first tentative induction step. MINERVA already knows something about both *john* and *mary*, and this background knowledge provides the opportunity for generalization of those people to others like them. Knowing that *john* is one of several instances of *person*, the feature *john* is replaced by the more general feature *person*, and MINERVA asks a question to test the hypothesis *father(X, mary)←person(X)*.

? father(cathy, mary). n

Because the answer was negative, the alternative hypothesis which replaces *mary* by any *person*, *father(john, X)←person(X)*, seems immediately more interesting and is tested next.

? father(john, cathy). n

With another negative response, MINERVA returns attention to the first hypothesis, testing it further.

? father(james, mary). n

Again, attention is turned to the second hypothesis. This time there is a positive response, so work on this hypothesis continues further. The encouragement of the positive response is not outweighed by the succeeding two negative responses and the testing of the hypothesis continues to completion.

? father(john, james). y
? father(john, john). n
? father(john, peter). n
? father(john, simon). y

There is nothing further to be done with that hypothesis although time remains, so MINERVA tests the other hypothesis further in case there is more to be discovered.

? father(mary, mary). n
? father(peter, mary). n
? father(simon, mary). n

Now all possible generalizations of the example are fully tested and MINERVA chooses the best one to assimilate into the program. In addition to the best hypothesis, MINERVA adopts unit clauses for each of the newly discovered facts which are *not* covered by the preferred hypothesis. In this case MINERVA chooses simply to adopt the example itself as well as the newly discovered facts. Here is the program at this point.

person(john)←
person(mary)←
person(peter)←
person(james)←
person(simon)←
person(cathy)←
father(john, mary)←
father(john, simon)←
father(john, james)←

2.3 Learning from Background Knowledge

Now we'll allow MINERVA to quickly build up its theory with some more facts. Although MINERVA can and does develop inductive hypotheses from these examples, we do not allow the time for developing the hypotheses and so the facts themselves are adopted as unit clauses.

> *+father(mark, john).*
!
> *+father(mark, andrew).*
!
> *+father(andrew, emily).*
!
> *+father(andrew, alison).*
!

Now MINERVA uses the background knowledge it has built up to learn a clause describing a new concept, *grandfather*. Here, as before, a negative answer for the first hypothesis test causes the focus of attention to shift to an alternative hypothesis.

> *+grandfather(mark, alison).*
...... *grandfather(X, alison)← father(X,Y)*
? grandfather(andrew, alison). n
...... *grandfather(mark, X)← father(X,Y)*
? grandfather(mark, andrew). n

The next hypothesis tried is a specialization of the first one which has been rejected: *grandfather(X, alison)←father(X, john)*. But this hypothesis can only account for the same fact as the input example, so there is no need to test it. Some other similarly uninteresting hypotheses are considered by MINERVA without generating any questions until:

...... *grandfather(mark, X)*$\leftarrow$ *father(andrew, X)*
? grandfather(mark, emily). y

The positive answer here confirms every fact covered by this hypothesis, so the hypothesis is generalized further, using the background knowledge *father(mark, andrew)*$\leftarrow$.

...... *grandfather(X, Y)*$\leftarrow$ *father(X,Z), father(Z,Y)*
? grandfather(mark, james). y
? grandfather(mark, mary). y
? grandfather(mark, simon). y

These answers confirm all the facts covered by that hypothesis. Further hypotheses are investigated by MINERVA but no further questions are required to evaluate them. Eventually MINERVA chooses to adopt the latter hypothesis.

2.4 Returning to an Earlier Concept

Earlier, MINERVA learnt some facts about the concept *person*. Here MINERVA observes some more examples of persons, and finds a way to describe them in terms of other concepts learnt in the intervening period. Several hypotheses are investigated by MINERVA but shown here is the only one which gives rise to some questions and which is eventually adopted.

> *+person(emily).*
...... *person(X)*$\leftarrow$ *father(Y,X)*
? person(alison). y
? person(andrew). y

2.5 Interruption

Now we aim to build up some more of the theory to prepare for demonstration of some more interesting behaviour. The search by MINERVA for hypotheses to cover each of the following examples is prematurely terminated by external interruption, so each example is simply added as a fact to the program. We can assume that the interruption corresponds to an inability or unwillingness of the environment to answer the question, or the need of the agent in which MINERVA is embedded to focus resources elsewhere.

> *+mother(sally, james).*
!
> *+mother(sally, mary).*
!
> *+married(john, sally).*
!
> *+married(andrew, alice).*
!
> *+mother(alice, mia).*

```
!
> +mother(alice, andrea).
!
```

2.6 Learning a Symmetric Relation

The next example prompts a simple symmetry hypothesis to be generated using the background clause *married(andrew, alice)←*.

```
+married(alice, andrew).
...... married(X,Y)← married(Y,X).
? married(sally, john). y
!
```

MINERVA would continue to generate and test other hypotheses if not for the interruption here. Instead, the hypothesis *married(X,Y)←married(Y,X)* is adopted. This hypothesis is an example of a clause which can contribute considerably to the conciseness of the program representation of a theory — for every married couple henceforth only one side of the relationship need be presented as an example and MINERVA immediately recognizes the dual fact — but which can create difficulties for interpretation of the program by standard logic program interpreters. MINERVA's interpreter has no difficulty with it.

The following sequence gives the first example of error diagnosis in action. In this case missing answer diagnosis is employed, but we will suspend the explanation of diagnosis until we have a more complex program and hence a more comprehensive example. For the present the reader should note that the symmetric relation is easily handled by the diagnoser, although conventional declarative diagnosis techniques would not terminate in diagnosing this missing answer. Again, we interrupt learning prematurely here to keep it brief. In the following, the questions asked before the first inductive hypothesis is indicated by "......" are employed in the diagnosis stage of learning.

```
> + mother(robyn, kim).
> + mother(robyn, peter).
!
> + married(richard, robyn).
? married(robyn, richard). y
...... some uninteresting hypotheses
!
> +married(cathy, mark).
? married(mark, cathy). y
...... some uninteresting hypotheses
!
> +mother(cathy, matthew).
!
```

2.7 Learning Exceptions

So far, negative answers to questions have caused inductive hypotheses to be rejected outright. In the following sequences we see two ways that MINERVA can modify a hypothesis to exclude counter-examples discovered during experimentation (or even previously observed). First, the second-last hypothesis in the next sequence is specialized to generate the last one.

> *> + father(richard, peter).*
> *...... father(richard, X)← person(X)*
> *? father(richard, cathy). n*
> *...... father(richard, X)← mother(Y,X)*
> *? father(richard, andrea). n*
> *...... father(richard, X)← mother(robyn, X)*
> *? father(richard, kim). y*

Next we see how counter-examples can be handled by inventing an exception predicate. This is done in preference to rejecting a hypothesis when the next best hypothesis is so poor that working around the exceptions results in a better hypothesis. The ability to adopt a hypothesis even in the presence of counter-examples gives MINERVA some resilience to *noise* in the environment and to errors in its background knowledge. In the next sequence the previous hypothesis, having been fully tested is generalized further using the background clause *married(richard, robyn)←*.

> *...... father(X,Y)← married(X,Z), mother(Z,Y)*
> *? father(andrew, andrea). y*
> *? father(andrew, mia). n*

A negative answer to this last question causes attention to be turned to other hypotheses. After investigating others, without need for further questions, MINERVA returns to test this further:

> *? father(mark, matthew). y*

Now it is fully tested and one counter-example was discovered. But there are still some other less promising hypotheses to be investigated in the available time.

> *...... father(richard, X)← mother(Y,Z), mother(Y,X)*
> *? father(richard, james). n*
> *!*

At interruption, MINERVA has not exhausted all possible inductive hypotheses, but the best one so far, comprising two clauses, is adopted.

$$father(X,Y) \leftarrow married(X,Z),\ mother(Z,Y),\ \sim father^{0}(X,Y)$$
$$father^{0}(andrew,\ mia) \leftarrow$$

MINERVA has invented an exception predicate $father^{0}$ to describe the exception to the clause.

2.8 Removing Redundant Clauses

Two more examples will enhance the illustration of redundancy detection.

> *+mother(sally, simon).*
> *!*
> *+mother(cathy, john).*
> *!*

Now MINERVA has constructed the following program to represent a theory of families. The clauses marked by "*" are redundant in the sense that every fact they contribute to the theory is also covered by another clause.

```
*  person(john)←
*  person(mary)←
*  person(peter)←
*  person(james)←
*  person(simon)←
   person(cathy)←
*  father(john, mary)←
*  father(john, simon)←
*  father(john, james)←
*  father(mark, john)←
   father(mark, andrew)←
   father(andrew, emily)←
   father(andrew, alison)←
   grandfather(X,Y)← father(X,Z), father(Z,Y)
   person(X)← father(Y,X)
   mother(sally, james)←
   mother(sally, mary)←
   married(john, sally)←
   married(andrew, alice)←
   mother(alice, mia)←
   mother(alice, andrea)←
   married(X,Y)← married(Y,X)
   mother(robyn, kim)←
   mother(robyn, peter)←
   married(richard, robyn)←
   married(cathy, mark)←
   mother(cathy, matthew)←
   father(X,Y)← married(X,Z), mother(Z,Y), ~ father0(X,Y)
   father0(andrew, mia)←
   mother(sally, simon)←
   mother(cathy, john)←
```

MINERVA can detect and remove such clauses in the program, in response to an external instruction to do so, called *sleep*. MINERVA will check as many clauses of the program

as time allows — sleeping may be interrupted at any time.

> *sleep*

The program representing the theory of families is simplified by removing each redundant clause from the program. These comprise several unit clauses about *person* and *father* as the facts they describe are also covered by other clauses in the program.

2.9 Diagnosis

MINERVA has a short-term memory for facts distinct from the memory for a program. When examples are observed or questions are answered the facts and their validity are stored in the finite fact memory in a first-in-first-out allocation scheme. These facts are typically (although not always) consistent with the theory when MINERVA is idle. The purpose of the fact memory is simply to reduce the number of questions required in diagnosis and experimentation — a question is not asked if the answer is found in the fact memory. The fact memory is only short-term because the facts are also represented in the theory, usually in a more compact form, and so they do not contribute to the knowledge of the learner. The size of the fact memory affects the number of questions asked of the environment by MINERVA, but otherwise does not affect learning performance.

Here we illustrate diagnosis — when an error in the theory is not due to an error in the definition of the concept of the example but rather to an error in a concept on which the definition depends. This example particularly demonstrates *contradiction backtracing* diagnosis, invoked for an error in a negative example. First, two more examples make ready for the diagnosis example.

> *+mother(alice, michael).*
!
> *+stepfather(andrew, michael).*
!

At this point we have that *grandfather(mark, michael)* is true in the theory, using some unit clauses and the rules

$$grandfather(X,Y) \leftarrow father(X,Z),\ father(Z,Y)$$
$$father(X,Y) \leftarrow married(X,Z),\ mother(Z,Y),\ \sim father^0(X,Y)$$

> *−grandfather(mark, michael).*

Contradiction backtracing diagnosis commences by verifying each of the antecedent facts which contribute to the false conclusion of the first rule: *father(mark, andrew)* is confirmed to be true in the fact memory and *father(andrew, michael)* is checked with a question.

? father(andrew, michael). n

But the second rule implied that the answer should be "y". So now the diagnosis procedure checks the antecedents of that rule. *married(andrew, alice)* and *mother(alice, michael)* are each confirmed true in the fact memory. *father*^{0} cannot be asked about because it is an internally invented concept and has no meaning in the environment. MINERVA must conclude that either the rule is wrong or there should be another exception to the rule: *father*^{0}*(andrew, michael)*.

2.10 Generalizing an Exception

Because of the utility of the rule in question in accounting for several true facts in the theory, in this case MINERVA chooses to retain the rule but to record another exception to it. So *father*^{0}*(andrew, michael)* is assumed missing from the theory and MINERVA goes on to generalize the concept as for any learning example. Indeed, MINERVA finds and adopts a good generalization without any need to ask further questions: *father*^{0}*(X, Y)←stepfather(X, Y)*.

```
!
```

Now MINERVA will learn some more about stepfathers, firstly by an example of a *stepfather*.

```
> +stepfather(andrew, mia).
!
```

At this point the unit clause corresponding to the example is adopted, and as a consequence *father(andrew, mia)* is false in the theory. MINERVA may also be prompted to generalize *stepfather* by a negative example of a *father*. In this case only one question of the diagnosis phase is apparent because the other answers are available in the fact memory.

```
> +mother(kim, kate).
!
> +mother(kim, kris).
!
> +married(bill, kim).
? married(kim, bill). y
!
> −father(bill, kris).
? stepfather(bill, kris). y
...... stepfather(bill, X)← mother(Y,X)
? stepfather(bill, andrea). n
...... stepfather(X, kris)← married(X,Y)
? stepfather(andrew, kris). n
...... stepfather(bill, X)← mother(kim, X)
? stepfather(bill, kate). y
...... stepfather(X,Y)← married(X,Z), mother(Z,Y)
? stepfather(andrew, andrea). n
```

? stepfather(john, james). n
!

The best hypothesis found before interruption is *stepfather(bill, X)←mother(kim, X)*. This hypothesis is adopted because it is useful in the present environment — if MINERVA later learns of other children of *kim* and *bill* it may be revised or removed.

2.11 Missing Answer Diagnosis

Until now the examples of diagnosis have focused on the identification of a false rule. Here missing answer diagnosis aims to find whether the missing atom of *grandfather* is due to a missing atom in a concept on which it depends. Only one clause of the program, *grandfather(X, Y)←father(X,Z), father(Z, Y)* could account for the missing atom through missing atoms in sub-concepts.

> *+grandfather(richard, kris).*
? father(kim, kris). n
? father(peter, kris). n

There is no substitution for the variables in the clause that would have the antecedents being true facts in the theory and implying the desired consequent. There are, however, two possibilities for the first antecedent: *father(richard, kim)* and *father(richard, peter)* are true in the theory. But questions show that the second antecedent cannot be satisfied in each case. MINERVA is unable to ask an existential question of the form "Is *richard* the *father* of anyone else?" and so is forced to assume that this rule is not appropriate for concluding *grandfather(richard, kris)*, and thus that atom is diagnosed as uncovered. MINERVA proceeds to generalize it.

...... *grandfather(X, kris)← married(X, Y)*
? grandfather(andrew, kris). n
...... *grandfather(richard, X)← mother(Y,X)*
? grandfather(richard, andrea). n
...... *grandfather(richard, X)← mother(kim, X)*
? grandfather(richard, kate). y
!

The clause *grandfather(X, Y)←father(X,Z), mother(Z, Y)* would be considered, and eventually adopted, if MINERVA were permitted to continue further but at this time the less general clause *grandfather(richard, X)←mother(kim, X)* is the best found and is adopted upon interruption.

2.12 Replacing a Redundant Rule

An inductive hypothesis which is adequate at some time in the learning sequence may be adequate or even ideal with respect to the language and knowledge available to

MINERVA at the time of its adoption. Later, when more is known, a more general hypothesis may be more appropriate. Here MINERVA demonstrates its ability to replace the true hypothesis about *grandfather* just learnt, by a more general one at a later time. First we introduce another family to MINERVA, noting the ease with which MINERVA is able to learn more about *person*, interspersed with other concepts. Indeed, MINERVA finds a hypothesis about person expressed in terms of another concept which has only become available in the meantime.

```
> +father(paul, paula).
> +mother(paula, philip).
!
> +mother(paula, betty).
!
> +mother(paula, karen).
!
> +person(karen).
...... person(X)← mother(Y,X)
? person(betty). y
? person(kate). y
? person(kris). y
? person(mia). y
? person(michael). y
? person(philip). y
!
```

Now we give another *grandfather* example. The question in the diagnosis stage indicates an examination of the *grandfather* clause in terms of a *father* of a *father*.

```
> +grandfather(paul, karen).
? father(paula, karen). n
...... grandfather(X, karen)← father(X,Y)
? grandfather(andrew, karen). n
...... grandfather(paul, X)← mother(Y,X)
? grandfather(paul, andrea). n
...... grandfather(paul, X)← mother(paula, X)
? grandfather(paul, betty). y
? grandfather(paul, philip). y
...... grandfather(X,Y)← father(X, Z), mother(Z,Y)
...... grandfather(X,Y)← person(Y), father(X,Z)
? grandfather(andrew, cathy). n
!
```

MINERVA adopts the hypothesis *grandfather(X,Y)←father(X,Z), mother(Z,Y)*. The program is now:

```
  person(cathy)←
  father(mark, andrew)←
  father(andrew, emily)←
  father(andrew, alison)←
  grandfather(X,Y)← father(X,Z), father(Z,Y)
  person(X)← father(Y,X)
  mother(sally, james)←
  mother(sally, mary)←
  married(john, sally)←
  married(andrew, alice)←
  mother(alice, mia)←
  mother(alice, andrea)←
  married(X,Y)← married(Y,X)
  mother(robyn, kim)←
  mother(robyn, peter)←
  married(richard, robyn)←
  married(cathy, mark)←
  mother(cathy, matthew)←
  father(X,Y)← married(X,Z), mother(Z,Y), ~ father^0(X,Y)
* father^0(andrew, mia)←
  mother(sally, simon)←
  mother(cathy, john)←
  stepfather(andrew, michael)←
  father^0(X,Y)← stepfather(X,Y)
  mother(alice, michael)←
  stepfather(andrew, mia)←
  mother(kim, kate)←
  mother(kim, kris)←
  married(bill, kim)←
  stepfather(bill, X)← mother(kim, X)
* grandfather(richard, X)← mother(kim, X)
  father(paul, paula)←
  mother(paula, philip)←
  mother(paula, betty)←
  mother(paula, karen)←
  person(X)← mother(Y,X)
  grandfather(X,Y)← father(X,Z), mother(Z,Y)
```

The redundant clauses — those that are marked "*" including the earlier clause for *grandfather* — may be removed.

```
> sleep
```

2.13 Simplifying Exceptions

There is one more feature of MINERVA to highlight here. We have seen that exceptions to a clause may be recognized and collected as they are discovered. Sometimes, the exceptions may become so numerous and complex that the rule combined with the exceptions becomes more complex than the observational facts it accounts for!

In the next sequence MINERVA observes many counter-examples to the *father* rule until it is eventually replaced. If the counter-examples are themselves *stepfathers*, this will not add complexity because they are simply represented as previously unknown instances of *stepfather*. Instead they must be counter-examples to *father* which are not instances of *stepfather*. Perhaps they are adult children of *paula* who do not care to acknowledge a relationship with *paula*'s present husband, *tom*. In order to clarify the procedure here, assume that the short-term fact memory is empty at this point. This enables us to see the reasoning of MINERVA by the questions asked.

> *+married(paula, tom).*
? married(tom, paula). y
!
> *−father(tom, philip).*
? mother(paula, philip). y
? stepfather(tom, philip). n
? father(john, james). y
? father(john, mary). y
? father(john, simon). y
...... *father*^{0}*(tom, X)*← *mother(X,Y)*
!

In this dialogue, MINERVA diagnosed a missing exception to *father* but then checked some of the other *father* facts covered by the excepted clause. Those facts were true, so without checking all of them, it seems better to add another exception and then to attempt to generalize the exception. In this case, no good generalization was found and the exception hypothesis *father*^{0}*(tom, philip)* ← is adopted. Now we define some more children of *paula* so that there will be a sufficient number of exceptions. For each, the unit clause hypothesis is adopted.

> *+mother(paula, rowena).*
!
> *+mother(paula, sam).*
!
> *+mother(paula, tim).*
!

Here, each counter-example causes MINERVA to check one more fact the rule accounts for, before deciding to add yet another exception.

> *−father(tom, rowena).*
? stepfather(tom, rowena). n
? father(andrew, andrea). y

```
!
> −father(tom, sam).
? stepfather(tom, sam). n
? father(mark, matthew). y
!
> −father(tom, tim).
? stepfather(tom, tim). n
? father(mark, john). y
!
```

The next counter-example, together with some more counter-examples found while checking more covered facts, is enough to tip the balance.

```
> −father(tom, karen).
? mother(paula, karen). y
? stepfather(tom, karen). n
? father(tom, betty). n
? father(richard, kim). y
? father(richard, peter). y
>
```

MINERVA decides to remove the *father* clause that has been so troublesome and each of its exceptions:

$$father(X,Y) \leftarrow married(X,Z),\ mother(Z,Y),\ \sim father^{0}(X,Y)$$
$$father^{0}(X,Y) \leftarrow stepfather(X,Y)$$
$$father^{0}(tom,\ philip) \leftarrow$$
$$father^{0}(tom,\ rowena) \leftarrow$$
$$father^{0}(tom,\ sam) \leftarrow$$
$$father^{0}(tom,\ tim) \leftarrow$$

They are replaced by unit clauses to cover each of the true facts for which the clause alone was responsible for including in the theory. That is, each clause defining $father^{0}$ is removed and the following clauses are added.

father(andrew, andrea)←
father(john, james)←
father(john, mary)←
father(john, simon)←
father(mark, john)←
father(mark, matthew)←
father(richard, kim)←
father(richard, peter)←

2.14 Summary

The example has demonstrated the major features of MINERVA as a learning algorithm. The representation of knowledge as a logic program offers a domain-independent description language and permits the application of the concepts and tools of logic programming. The incremental revision process is demand-driven by the incremental presentation of observations. Experiments are heavily used for diagnosis of errors and evaluation of inductive hypotheses. Inductive hypotheses are developed in the context of the background knowledge acquired in earlier learning steps. At all times the learning process is "interruptible"— the theory is corrected in the best manner known at the point of interruption, although more time could allow a better revision. Some revisions are made by recording exceptions to rules, and exceptions themselves may be described by general rules.

3

Foundation Concepts

3.1 Introduction

In this chapter the concept learning problem is introduced, at first by identifying what a concept is and what it means to learn a concept in terms of a concept description. Basic terminology and techniques of concept learning are introduced with particular emphasis on the tools of inductive logic programming and logic programming on which it is founded. Special attention is given to techniques for interpreting logic programs and diagnosing errors in programs as these techniques will underpin components of MINERVA.

3.2 Inductive Concept Learning

Viewing a *concept* as a set of objects, *positive examples* are a set of objects that are concept members and *negative examples* are a set of objects that are not concept members. A solution to the inductive concept learning problem is a *description* of the concept that determines whether or not some other objects are also members. A *learner* is an algorithm designed to solve the concept learning problem.

An object is described by some *attributes* or *features* of the object: symbols in the language of observation. Sometimes a learner is expected to construct descriptions of multiple concepts. In this case an example names the concept to which the object does or does not belong and the names are also symbols of the language of observation.

A set of named concepts is called a *theory*. A description of multiple concepts is called a *theory representation*. The reader's attention is drawn to the terminology — some writers would call the concept description a "theory" and the concepts them-

selves "instances", "extension", or "semantics", in conflict with the practice of logicians [Gärdenfors 1988, page 24].

The concept learning problem is inherently *inductive*. Induction is often described as a process of inference from some specific instances to a general description which is satisfied by those instances as well as others. In contrast with deductive inference it is not truth preserving. An inductive learner, through construction of concept descriptions describing some concept members, acquires an *epistemic attitude* [Gärdenfors 1988] about other objects that have not been observed. The epistemic attitude represents a belief, or opinion, about the concept membership of those objects. Accordingly, a concept description, or a part of a concept description, is often called a *hypothesis*.

For any concept learning problem there is a special theory, usually unknown to the learner, called the *target*. The target determines whether or not objects are concept members, providing the classification of objects as positive or negative examples thus providing an ideal model for the theory representation developed by the learner.

Most experimental concept learning research deals with *batch* learning where the input is a finite set of examples. This thesis is concerned with *incremental* learning where the examples are presented as a sequence which may be infinite. Although the aim of a batch learner is to construct a single good theory in response to the input, an incremental learner aims to construct a sequence of theories, each of which is good with respect to the initial sub-sequence of input.

Unless every concept member and non-member of the theory held by a learner can be verified, an incremental learner will sometimes find that an example contradicts the current theory — a *failing example*. The learner is forced to *revise* the theory to remove the contradiction. Even some batch learners proceed by revising a given incorrect theory to account for a batch of failing examples. The process of revision usually comprises a *diagnosis* component by which particular parts of the theory representation are blamed for the error, and a subsequent hypothesis formulation component by which corrections to those parts are made. Many revising learners employ the techniques of *declarative debugging* known as *contradiction backtracing* and *missing answer diagnosis*, first introduced by Shapiro [1981]. Inductive learners typically formulate corrective hypotheses by a process of *generalization* — expanding a concept to include more objects, and *specialization* — contracting a concept to include fewer.

3.2.1 Learning in the Inductive Logic Programming framework

Inductive Logic Programming [Muggleton 1991] provides a logical framework for the investigation of incremental inductive learning. In this framework the theory representation for theories developed by induction is the language of logic programs. In this context, a theory representation is a *program* and, loosely, the theory it represents is a Herbrand model for the program.

In logic programming terminology, the target identifies an *intended interpretation* for the *language of observation* in which examples are expressed. Hence a sentence which

is true in the target is true in the intended interpretation (for which we say simply that the sentence is *true*) and one false in the target is false in the intended interpretation (which we call *false*). In particular, a positive example is a true atom and a negative example is a false atom.

In the ILP framework, a predicate is a relation on features, that is, a relation on terms expressed using the constants and non-unary function symbols in the language of observation. A predicate symbol is identified with a concept by the naming of examples, so "concept" and "predicate" are used as synonyms.

The task for any induction mechanism is simplified if the knowledge representation language has a simple syntax, as do logic programs. Representational power is not sacrificed: logic programs are computationally complete [Lloyd 1987b, page 53] and so are capable of representing undecidable concepts. It is easy in the ILP framework to syntactically restrict the language in various ways to avoid particular difficulties. For example, by avoiding the use of function symbols, decidability is guaranteed.

As with logic programming, the formal semantics and tools of the first-order predicate calculus provide a powerful foundation for the development of formal theory for inductive logic programming.

The fragment of the calculus with which logic programming is concerned has lent itself to the development of theoretically sound and practically feasible deduction methods. These methods are implemented in interpreters and compilers for the PROLOG programming language. The problem of interpretation of a program, that is, determining whether an object is a member of a concept in a theory, is well understood in this framework. The tools and terminology of the mature field of research into logic programming are available as a starting point for the development of the new field of inductive logic programming.

In the following section some basic terminology and results of logic programming are introduced, prior to introducing the basic concepts of ILP that are founded on them. Deeper presentations of logic programming and ILP concepts which are particularly relevant to this work follow.

3.3 Notation

In this work use of the language and notation of logic programming is based on Lloyd's [Lloyd 1987b, pages 4–10]. Definitions of the terms are given in the succeeding section.

Single upper case letters are used to denote clauses or sets of literals, (C, D, H, I, K, S) and single lower case letters to denote constants (a, b, c, d, e), variables (u, v, w, x, y, z) function symbols (f, g, h), and predicate symbols (p, q, r, s, t). P is used to denote a program, A an atom or a unit clause and L a literal. Lower case Greek letters $(\alpha, \beta, \delta, \gamma, \epsilon, \theta, \phi, \sigma, \mu)$ denote substitutions. Occasionally these symbols are sub- or super-scripted.

Where mnemonic symbols are more appropriately used in examples, constant, function, and predicate symbols are sequences of lower case letters and variables are distinguished as upper case letters. This complies with the usual PROLOG notation convention.

The logical symbols are used as follows: $\sim$ for negation, $\wedge$ or sometimes "," for conjunction, $\vee$ for disjunction, $\leftarrow$ for implication, $\models$ for logical consequence, $\forall$ for universal quantification and $\exists$ for existential quantification. The quantifiers are used to close a formula by quantifying every unbound variable in the formula. The scope of a quantifier extends as far to the right as possible.

The symbol $-$ is used in the text to denote set difference. That is for sets S and T, $S - T = \{A | A \in S \textit{ and } A \notin T\}$.

3.4 Preliminary Concepts of Logic Programming

In this section formal definitions of basic logic programming concepts are given. These will provide a foundation for the introduction of the basic concepts of ILP. Some of the definitions are taken unaltered from Lloyd [1987b]. Where precise definitions are commonly understood and not critical to the formal reasoning of this work, the definition is not repeated here but the reader is referred to Lloyd.

3.4.1 Terms

Lloyd's definitions are used for *constant*, *function symbol*, *variable*, and *predicate symbol*.

A term is either a variable, a constant symbol, or a composite term. A *composite term* $f(t_1, \ldots, t_n)$ is comprised of *function symbol* f of *arity* $n > 0$ and n *arguments*, $t_1, \ldots, t_n$, each of which is a term. "Function symbol" is used exclusively to refer to function symbols of arity greater than zero, and so does not encompass constant symbols.

An *atom* $p(t_1, \ldots, t_n)$ is comprised of a *predicate symbol* p of *arity* $n \geq 0$ and n arguments $t_1, \ldots, t_n$, each of which is a term. A *literal* is an atom or a negated atom. A *positive literal* is an atom and a *negative literal* is a negated atom.

A term s *occurs* in a term t when $s = t$ or when s occurs in an argument of t. A term s *occurs* in an atom A when A has at least one argument and s occurs in an argument of A. A term t occurs in a set of atoms or set of terms if t occurs in any element of the set.

A term is *ground* if no variable occurs in the term.

3.4.2 Clauses and programs

A *normal clause* C, is an expression $A \leftarrow B_1, \ldots, B_n$ where A is an atom and each B_i $(i = 1, \ldots, n)$ is a literal. A normal clause is a *definite clause* when each B_i is an atom. "$C_\odot$" denotes the *head* of clause C, the atom A. "$C_\otimes$" denotes the *body* of clause C, $B_1, \ldots, B_n$. Occasionly a clause body will be regarded as a *set* of literals as is common practice in ILP; the intended meaning is clear from context. A clause is implicitly universally quantified, so clause C is equivalent to $\forall C$.

A *normal goal*, G, is an expression $\leftarrow B_1, \ldots, B_n$ where each B_i $(i = 1, \ldots, n)$ is a literal. A normal goal is a *definite goal* when each B_i is an atom. "$G_\otimes$" denotes the *body* of the goal G, $B_1, \ldots, B_n$. The *empty goal* has no literals in the body.

A *unit clause* is a clause with no literals in the body. A *rule* is a clause which is not a unit clause. The literals in the body of a rule are called *antecedents*. A *fact* is a ground literal.

A *definite program* is a set of definite clauses. A *normal program* is a set of normal clauses. A *logic program* or a *program* can be either of these.

Let L be a literal. Then L is *about* the predicate symbol of L. Let C be a clause. Then C is a *about* the predicate symbol of $C_\odot$ and C *defines* the predicate symbol of $C_\odot$. Let P be a program and p be a predicate symbol. Then the maximal subset of P in which every clause is about p is called the *definition* of p.

3.4.3 Substitutions

As defined by Lloyd [1987b], a *substitution* θ is a finite set of the form $\{v_1/t, \ldots, v_n/t_n\}$, where each v_i is a variable, each t_i is a term distinct from v_i and the variables $v_1, \ldots, v_n$ are distinct. The variables $v_1, \ldots, v_n$ are called the *input* of the substitution and the terms $t_1, \ldots, t_n$ are called the *output*. θ is called a *variable-pure* substitution if the t_i are all variables. θ is called a *ground* substitution if the t_i are all ground terms. The substitution $\{\}$ is called the *identity* substitution.

An *expression* is a literal, clause or goal, or a conjunction, disjunction or set of literals, clauses or goals.

Let E be an expression and let $\theta = \{v_1/t, \ldots, v_n/t_n\}$ be a substitution. To *apply* θ to E giving $E\theta$, for each $v_i/t_i \in \theta$ simultaneously, replace every occurrence of v_i in E by t_i.

Let $\theta = \{u_1/s_1, \ldots, u_n/s_n\}$ and $\sigma = \{u_1/s_1, \ldots, u_n/s_n\}$ be substitutions. To *compose* θ and σ giving the substitution $\theta\sigma$, construct $\{u_1/s_1\sigma, \ldots, u_m/s_m\sigma, v_1/t_1, \ldots, v_n/t_n\}$ and then delete any $u_i/s_i\sigma$ $(i = 1, \ldots, m)$ for which $u_i = s_i\sigma$ and delete any v_j/t_j for which $v_j \in \{u_1, \ldots, u_m\}$.

Let E be an expression. Then E is *functor-free* if no function or constant symbol occurs in E. E is *function-free* if no function symbol occurs in E. E is *ground* if no variable

occurs in E. Let θ be a substitution. Then $E\theta$ is an *instance* of E. If $E\theta$ is ground then $E\theta$ is a *ground instance* of E. If, for every $v/t \in \theta$, t is ground then θ is a *ground substitution.*

Let E and F be expressions. Then E is a *variant* of F if there are substitutions θ and σ such that $E = F\theta$ and $F = E\sigma$. Also θ is a *renaming substitution* for E.

Let E and F be atoms. Then substitution μ is a *most general unifier* (*mgu*) for E and F if $E\mu = F\mu$ and for any substitution θ such that $E\theta = F\theta$ there exists a substitution γ such that $\theta = \mu\gamma$.

Basic algebraic properties of substitutions are given by Lloyd [1987b, page 21].

3.5 Preliminary Concepts of Inductive Logic Programming

The basic concepts of ILP are built on those of logic programming. In this section the theoretical foundations of ILP that are important to MINERVA are introduced: generality models, generalizing operators, and representation change by flattening and predicate invention.

3.5.1 Generality models

Formalised induction requires a formal notion of generality. A generality relation imposes an ordered structure on hypotheses, potentially improving the efficiency of a search for a satisfactory hypothesis. This principle was demonstrated by Mitchell's [1982] pioneering work on concept learning in *version spaces*.

The generality relation usually induces a quasi-order on language objects. In this work the term "more general" is informally used to mean either strictly more general or equivalent in generality. Recalling that the variables in clauses are implicitly universally quantified, it is assumed henceforth that clauses are *standardised apart*, that is, the variables are renamed so that the clauses share no variables.

In concept learning, a concept $\mathcal{C}_1$ is a *generalization* of $\mathcal{C}_2$ if $\mathcal{C}_2 \subseteq \mathcal{C}_1$. In the context of logic representations, generality is naturally understood as implication: a formula is more general than another one it implies. That is, $\mathcal{F}$ is more general than $\mathcal{G}$ if $\mathcal{F} \rightarrow \mathcal{G}$. But this logical notion of generality does not capture the idea of concept descriptions: when generality is used to compare formulae, they should be about the same concept. In a program, the clauses in a predicate definition are about that predicate.

In the following, some alternative notions of generality are presented. Although some apply to pairs of sets of clauses, only the generality relation on pairs of single clauses is considered here.

3.5.1.1 θ-subsumption

The θ-subsumption relation is weaker than implication, but has the advantage of being decidable and well understood because of its role in theorem-proving. It was investigated for induction by Plotkin [1970] in the context of sets of literals but here we give a definition suitable for definite clauses.

Definition 3.1 (θ-subsumption, $\succeq$) *Let C and D be definite clauses. Then $C \succeq D$ (read C θ-*subsumes *D) if there exists a substitution θ such that $C_{\odot}\theta = D_{\odot}$ and $C_{\otimes}\theta \subseteq D_{\otimes}$.*

That is, a clause is more general than another by θ-subsumption if it can be turned to the other by dropping literals from the body and applying a substitution.

Note that if $C \succeq D$ then $C \rightarrow D$, but the converse does not hold for self-recursive clauses.

Example 3.1 Let C be the definite clause $p(f(x)) \leftarrow p(x)$. Let D be the definite clause $p(f(f(x))) \leftarrow p(x)$. Then $C \rightarrow D$ but not $C \succeq D$. □

3.5.1.2 Relative subsumption

θ-subsumption relates the generality of clauses in isolation. When learning in the presence of background knowledge represented by a logic program, the effect of that background knowledge on generality should be taken into account. For this purpose, Plotkin [Plotkin 1971a, Plotkin 1971b] defines *relative subsumption* on disjunctive sets of literals relative to background knowledge represented as a conjunction of disjunctive sets of literals. Buntine [1988] describes it: $\mathcal{C}$ is more general than $\mathcal{D}$ *relative* to $\mathcal{P}$ if there is a substitution θ such that $\mathcal{P} \models \forall(\mathcal{C}\theta \rightarrow \mathcal{D})$.

3.5.1.3 Generalized subsumption

Buntine [1988] proposes an alternative model called *generalized subsumption*, more appropriate and particularly designed for comparison of definite clauses with respect to definite programs. Unlike relative subsumption, clauses related by generalized subsumption are about the same concept. The underlying idea is that, with the aid of the program, the more general clause can be used to prove instances of the head of the less general clause at least whenever the less general clause could. The notion of *cover* is crucial. The definition for cover given here is Buntine's (rephrased from Shapiro), reformulated to refer to a normal clause rather than a definite clause for later convenience.

Definition 3.2 (Cover) *Let C be a normal clause and I be an interpretation. Then C* covers *ground atom A in I if there is a substitution θ such that $C_{\odot}\theta = A$ and $\exists(C_{\otimes}\theta)$ is true in I.* [Buntine 1988, page 156]

The cover *of C in I is the set of atoms A_i such that C covers A_i in I.*

Often we refer to the *cover* of a clause with respect to a program rather than an interpretation. In this case we loosely refer to the cover of a clause in the interpretation computed by an interpreter for the program. Ideally, for a definite program we mean the atoms covered by the clause in the interpretation for the language of the program together with the clause, that is the least Herbrand model for the program. For a normal program we refer ideally to the interpretation for the language of the program and the clause that is a minimal normal Herbrand model for the completion of the program (see [Lloyd 1987b, corollary 14.8]).

This dual use of the term is common practice but in formal reasoning we are careful to state the interpretation of interest. We can now give a formal definition of generalized subsumption.

Definition 3.3 (Generalized subsumption, $\succeq_P$) *Let C and D be definite clauses and let P be a definite program. Then $C \succeq_P D$ (read C is more general than D with respect to P by* generalized subsumption*) if for any Herbrand interpretation I (for the language of at least the symbols of P, C, and D) such that P is true in I and for any atom A such that D covers A in I then C covers A in I.* [Buntine 1988, definition 4.1]

Buntine [1988] also gives an alternative definition which he proves to be equivalent. We will use this theorem later to establish generalization properties.

Theorem 3.1 (Testing for $\succeq_P$) *Let C and D be definite clauses with distinct variables and P be a definite program. Let θ be a substitution grounding the variables in D using distinct new constants not occurring in C, D or P and let D' be the set of unit clauses $\{(A\leftarrow)|\ A \in D_{\otimes}\theta\}$. $C \succeq_P D$ if and only if, for some substitution σ, $C_{\odot}\sigma$ is identical to $D_{\odot}$ and $P \cup D' \models \exists(C_{\otimes}\sigma\theta)$.* [Buntine 1988, theorem 4.2]

In practice this means a more general clause under generalized subsumption may be converted to a less general clause by applying substitutions, adding atoms to its body, or by resolving it with some clause from the background knowledge.

Both generalized subsumption and relative subsumption with respect to empty programs coincide with θ-subsumption. Generalized subsumption is a special case of relative subsumption, requiring that the head of the less general clause is an instance of the head of the other clause. In particular, this means that only clauses with the same predicate symbols in the head may be related by generalized subsumption.

Because of this generalized subsumption is also a weaker model of generality than the logical model. Further, like relative subsumption and θ-subsumption, it does not have a clause to be more general than another which can be derived from it recursively.

Example 3.2 Let P be the program $\{p(a)\leftarrow\}$, let C be the clause $p(f(x))\leftarrow p(x)$, and let D be the clause $p(f(f(x)))\leftarrow p(x)$. Then $P \models (C\rightarrow D)$ but not $C \succeq_P D$. □

Following Buntine [1988] we call the search space for an inductive hypothesis induced by generalized subsumption, a *hierarchy* of clauses. The hierarchy *rooted* at the bottom by a given clause I for program P is a structured organisation of all clauses H such that $H \succeq_P I$. If there is a link upwards from clause D to clause C in the hierarchy then $C \succeq_P D$. The top element of such a hierarchy is the unit clause with the head having the same predicate symbol as $I_{\odot}$ and each argument being a distinct variable, since this is the most general clause that is more general than I.

3.5.2 Generalization

Having chosen a suitable model of generality, the question of how to construct generalizations arises. The concept of an inverse substitution is central. Loosely, an *inverse substitution* is a mapping of term occurrences to variables such that when applied to an expression, there is a substitution which may be applied to recover the original expression. Nienhuys-Cheng and Flach [1991] give a more formal definition and analyse the algebraic properties of inverse substitutions.

3.5.2.1 Least generalization

When inducing general rules from specific examples, a useful approach is to construct a single clause which is more general than each of them. If such a clause is minimally more general than each of them, it is called a *least general generalization* or *most specific generalization*. Any other clause more general than each is a generalization of the least general generalization.

Plotkin describes how to calculate the least general generalization by θ-subsumption and by relative subsumption [Plotkin 1970, Plotkin 1971b]. Buntine [1988] describes how to calculate the least general generalization by generalized subsumption. Least general generalizations by relative subsumption and generalized subsumption do not always exist.

3.5.2.2 Generalizing by θ-subsumption

An operator that generalizes according to θ-subsumption is sometimes called *truncation* [Rouveirol and Puget 1990a, Muggleton and Buntine 1988]. Jung [1993] shows how a truncation operator that is complete for θ-subsumption firstly duplicates atoms in the clause body an arbitrary number of times, then applies some inverse substitution and finally removes arbitrary atoms from the resulting clause body.

3.5.2.3 Generalizing by implication

Recent work has been directed at generalizing definite clauses according to implication, to overcome the deficiency of θ-subsumption in capturing the power of self-recursive

clauses. Suitable generalizing operators are variously called *inverse implication*, *recursive anti-unification* and *sub-unification* [Muggleton 1992, Lapointe and Matwin 1992, Idestam-Almquist 1993, Muggleton and Page 1994]. These operators are designed for generalization of a clause isolated from background knowledge. Muggleton [1992] and Lapointe and Matwin [1992] suggest mechanisms for taking background knowledge into account when it is represented by a program of ground unit clauses. It is not obvious how the operators would be extended to generalize in the context of definite clauses.

3.5.2.4 Generalizing by generalized subsumption

Inverse resolution provides a domain-independent technique for generalization of clauses with respect to background knowledge represented as a definite program.

It describes a suite of generalization operators which are based on the idea of inverting the well known deductive inference rule called *binary resolution* [Robinson 1965]. Muggleton and Buntine [1988] originally proposed the *V operators* called *absorption* and *identification*, the *W operators* called *intra-construction* and *inter-construction*, and also the *truncation* operator. We restrict out attention to the absorption operator, first introduced by Sammut [1981b], and central to the learning strategy of MINERVA.

A number of different forms of it are used — here we present it in its most general form for definite clauses, regarding clause bodies as sets of atoms. Later we will identify a more useful specific version.

Informal Definition 3.4 (Absorption) *Let I be a definite clause (the* input *clause) and B be a definite clause (the* background *clause). Then for each substitution θ such that $B_{\otimes}\theta \subseteq I_{\otimes}$, absorption generates every clause*

$$(I_{\odot} \leftarrow (I_{\otimes} - B_{\otimes}\theta) \cup \{B_{\odot}\theta\} \cup S)\Theta$$

where Θ is an inverse substitution and S is any subset of $B_{\otimes}\theta$.

The head of the new clause is the head of the input clause. To construct its body from the body of the input clause, remove some (possibly none) of the atoms which also occur in the body of the instantiated background clause and insert the instantiated head of the background clause. Then replace some terms by new variables as an inverse substitution. In some versions of the operator the inverse substitution is constrained [Muggleton and Buntine 1988].

Example 3.3 (Absorption) Let I be the input clause

grandfather(mark, alison)$\leftarrow$ *father(mark, Y), father(Y, alison)*

and B be the background clause

parent(U, V)$\leftarrow$ *father(U, V)*

Then before applying the inverse substitution we have the the set of clauses generated by absorption:

grandfather(mark, alison)$\leftarrow$ *father(mark, Y), parent(Y,alison)*
grandfather(alison, mark)$\leftarrow$ *father(mark, Y), parent(Y,alison), father(Y,alison)*
grandfather(alison, mark)$\leftarrow$ *father(Y,mark), parent(alison, Y)*
grandfather(alison, mark)$\leftarrow$ *father(Y,mark), parent(alison, Y), father(alison, Y)*

Taking the first clause and applying an inverse substitution here are just some of the results:

grandfather(X,Z) $\leftarrow$ *father(X, Y), parent(Y,Z)*
grandfather(X,Z) $\leftarrow$ *father(X, U), parent(Y,Z)*
grandfather(V,Z) $\leftarrow$ *father(X, U), parent(Y,Z)*
grandfather(X,Z) $\leftarrow$ *father(U, Y), parent(V, W)*
grandfather(mark, alison) $\leftarrow$ *father(mark, Y), parent(Y, alison)*
grandfather(X, alison) $\leftarrow$ *father(X, Y), parent(Y, alison)*
grandfather(X, alison) $\leftarrow$ *father(mark, Y), parent(Y, alison)*

The first of these could be both a correct and a useful generalization if the background program also contains the clause *parent(U, V)*$\leftarrow$*mother(U, V)*. □

Absorption performs a binary resolution step in reverse: consider any clause constructed by absorption of input clause I and background clause B. Then the *resolvent* of the new clause and B is either I or a clause that θ-subsumes I. Therefore I may be replaced by the new clause in any definite program which also contains B, and I remains a consequence of the program.

Intuitively, when there are other clauses in the definition of the concept that B is about, the new clause gives a more general definition than I for the concept that I is about. When a non-empty inverse substitution Θ is used, the new clause gives a more general definition for the concept of I because some constants or functors or have been replaced by variables. Formally, the new clause is more general than the input clause with respect to any program containing the background clause, under generalized subsumption [Taylor 1993, Jung 1993].

The inverse resolution framework does not specify which of all possible clauses should be chosen for absorption. Nor does it state which substitution to choose when more than one is possible, which atoms of the body of the background clause to retain in the body of the new clause, nor which inverse substitution to apply. These choices are left to the control strategy of a learner using absorption to generalize. A survey of the various approaches to these issues by a number of ILP learners is made by Ling and Narayan [1991].

3.5.2.5 Generality of clauses generated by absorption

The generality of different clauses constructed by absorption from the same two clauses cannot always be compared. Recall that for absorption, θ is constrained to satisfy

$B_{\otimes}\theta \subseteq I_{\otimes}$. If the domain of θ is restricted to the variables in the body of B, then the constructed clause is more general than otherwise (because only the minimal necessary substitution has been applied). However, when there is more than one literal in the body of C to which some literal in the body of B could be mapped by the substitution, there remains choice for θ, and the resulting clauses are of incomparable generality.

The choice of which literals in the body of I to retain in the new clause (that is, the choice of which literals of $B_{\otimes}\theta$ to remove) also affects the generality of the new clause. For any given θ, a clause which retains a subset of the literals which another retains is more general, and those which retain different literals are incomparable. For any given θ and given choice of which literals from $B\theta$ to include, different choices for Θ result in different clauses which all θ-subsume the one generated by an empty Θ, but which are not necessarily comparable in generality. These issues are dealt with in depth by Nienhuys-Cheng and Flach [1991].

3.5.2.6 Absorption extremities

Ignoring for the present the inverse substitution Θ, a most general (by θ-subsumption) solution for absorption of a particular input clause I with a background clause B using a particular suitable substitution θ such that $B_{\otimes}\theta \subseteq I_{\otimes}$ is given by

$$I_{\odot}\leftarrow((I_{\otimes} - B_{\otimes}\theta) \cup \{B_{\odot}\theta\})$$

Similarly, the least general (by θ-subsumption) solution is given by Muggleton [1991]:

$$I_{\odot}\leftarrow(I_{\otimes} \cup \{B_{\odot}\theta\})$$

Note that the least general solution is equivalently general to I with respect to any program which contains B, so least general absorption does not strictly generalize. The *saturation* operator of Rouveirol [1991b] is equivalent to exhaustive iterated least general absorption, where the clause produced by one application of least general absorption becomes the input clause for the successive application.

3.5.2.7 Restriction

One way of incrementally generating each of the clauses in between the most general and the least general is to start from the most general one but to keep track of the atoms of $B_{\otimes}\theta$, called the *optional atoms*. The other clauses may be successively generated from such a clause and the optional atoms by an operation called *restriction* by Sammut and Banerji [1986].

Definition 3.5 (Restriction, *restrict***)** *Let H be a definite clause and S be a non-empty set of atoms (the* optional atoms*) such that each atom in S does not occur in H. Then restrict(H, S) is the set of pairs of clauses, H' and atoms, S' such that there is an $A \in S$ such that H' is $H_{\odot}\leftarrow H_{\otimes} \cup \{A\}$ and S' is $S - \{A\}$.*

Now given a particular input clause, background clause and substitution and assuming a null inverse substitution, most general absorption and restriction may be coupled to generate each clause of absorption in a general to specific order. Indeed this is the generalization strategy of MARVIN [Sammut 1981b]. First, most general absorption is applied to give a clause H and the set of optional atoms S is recorded. Later, H is specialized by restricting it with the optional atoms S to give pairs H',S'. Later again, each H' is restricted with its S' to give more clauses H'' paired with optional atoms S''. This can continue until all the optional atoms are exhausted and hence every clause has been generated.

3.5.2.8 Flattening

So far, in the treatment of absorption we have avoided the inverse substitution step. That is because, through a representation change, we can convert the difficult problem of inverting a substitution on a clause to the simpler one of deleting antecedents in a different representation of the clause. The appropriate representation change, called *flattening*, is described by Rouveirol and Puget [1990b].

Flattening replaces function symbols and constants with variables and new predicate symbols. The new predicate symbols are defined in the program by associated unit clauses that do contain function symbols. The flattened clause, supplemented with the symbol-defining clauses, is logically equivalent to the original one [Rouveirol 1994].

For our purposes flattening is defined as follows. For simplicity, within the scope of the definition constant symbols are regarded to be function symbols of arity zero.

Definition 3.6 (Flattening, Flat predicate, Symbol-defining, $flat$) *Let C be the definite clause $p(t_1,\ldots,t_n)\leftarrow C_1,\ldots,C_m$ where p is a predicate symbol of arity $n \geq 0$ and each t_i is a term and each C_j is an atom. The definite clause $flat(C)$ is given by*

$$p(v_1,\ldots,v_n)\leftarrow(\bigcup_{i=1}^{n} flatt(t_i,v_i)) \cup (\bigcup_{j=1}^{m} flatl(C_j))$$

where each v_i is a variable and $flatl$ and $flatt$ are defined as follows.

Let $p(t_1,\ldots,t_n)$ be an atom with predicate symbol p and each t_i is a term. Then the set $flatl(p(t_1,\ldots,t_n))$ is given by $\{p(v_1,\ldots,v_n)\} \cup \bigcup_{i=1}^{n} flatt(t_i,v_i)$ where each v_i is a variable.

Let $f(t_1,\ldots,t_n)$ be a term with functor f of arity $n \geq 0$ and argument terms t_i and let v be a variable that uniquely stands for the term $f(t_1,\ldots,t_n)$ throughout. Then $flatt(f(t_1,\ldots,t_n),v)$ is the set $\{f_n(v,v_1,\ldots,v_n)\} \cup \bigcup_{i=1}^{n} flatt(t_i,v_i)$ where each v_i is a variable and f_n is a flat predicate symbol *of arity $n+1$ which cannot occur in C (or any program containing C). Let v be a variable. Then $flatt(v,v)$ is the empty set.*

The symbol-defining *clauses, S, associated with $flat(C)$ are defined as follows. For each flat predicate symbol f_n occurring in $flat(C)$, there is a unit clause in S of the form $f_n(f(v_1,\ldots,v_n),v_1,\ldots,v_n)\leftarrow$ where each v_i is a distinct variable.*

We use *deep* to refer to a clause or program which does not have any flat predicate symbols occurring in it. We use *flat atom* to refer to an atom with a flat predicate symbol. We say that when $flatt(f(t_1,\ldots,t_n),v)$ is the set $\{f_n(v,v_1,\ldots,v_n)\} \cup \bigcup_{i=1}^{n} flatt(t_i,v_i)$ in the definition above, then $f_n(v,v_1,\ldots,v_n)$ *stands for* $f(t_1,\ldots,t_n)$ and also v *stands for* $f(t_1,\ldots,t_n)$. Notice that any atom that is an instance of the head of a symbol-defining clause is unambiguously determined by the term that is the first argument of the atom and that this term is never a variable. This property will be useful later.

Example 3.4 (Flattening) Let C be the deep clause $p(f(x,a),b) \leftarrow q(x,f(y,b))$.

$$flat(C) = p(u,v) \leftarrow f_2(u,x,w),\ a_0(w),\ b_0(v),\ q(x,z),\ f_2(z,y,v)$$

The associated symbol-defining clauses are

$$f_2(f(x,y),x,y) \leftarrow$$
$$a_0(a) \leftarrow$$
$$b_0(b) \leftarrow$$

The flat atom $f_2(z,y,v)$ in $flat(C)$ stands for $f(y,b)$ as does the variable z. □

Alternative forms of flattening, such as Ling and Narayan's [1991] *logical traces* and others discussed by Rouveirol [1994] are slight variations of the form presented here. A distinguishing feature of this form is that every occurrence of a term in the deep clause, is stood for by the same variable in the flattened clause.

The flattening operation is easily inverted by the inverse operation called *unflattening*. This is done by a syntactic manipulation of the flattened clause, removing each flat atom and unifying the first argument of the atom with a term comprised of the function symbol corresponding to the flat predicate symbol, applied to the remaining arguments. Alternatively, the unflattening operation may be viewed logically as performing several deductive steps of resolution with each of the notional unit symbol-defining clauses [Rouveirol 1994].

Deleting some flat atoms from a flattened clause corresponds to inverting a substitution in the corresponding deep clause [Rouveirol 1994]. Notice however that this kind of operation cannot invert a substitution which unifies variables in the deep clause.

Absorption, especially in the flat representation, is dealt with formally and in depth in chapter 9. The chapter establishes a new result for the completeness of absorption as a generalizing operator for generalized subsumption and also extends absorption to generalize normal clauses.

3.5.3 Predicate invention

In the ILP framework the invention of new terms in the theory representation language is often called *predicate invention*, although more broadly in machine learning research it is usually known as *constructive induction* [Michalski 1983]. The inverse resolution

framework provides operators for predicate invention, called *inter-construction* and *intra-construction*. The non-determinism of these operators is even greater than for absorption. Although recognition of the need for invented predicates is widespread and many attempts have been made to identify useful invented predicates, by for example De Raedt and Bruynooghe [1992], Muggleton [1994], Banerji [1991], and Stahl and Weber [1994], the problem remains largely unsolved.

When the theory representation permits, predicate invention can be useful for representing exceptions to rules of the program in a manner called *closed world specialization* [Bain and Muggleton 1990]. If a clause covers a false atom, a negative literal antecedent is added to the clause. Using the substitution that unifies the false atom with the clause head, the instance of the complement of the antecedent is an exception atom. When a clause covering the exception atom is also added to the program, the original clause no longer covers the false atom. When the negative literal is about an invented predicate that cannot occur in the language of observation, it is called an *exception predicate*. In principle, such predicates can then also be used as antecedents in other clauses.

Minimally, the exception predicate can be defined by unit clauses — one for each exception to the rule. This is equivalent to the techniques employed in KARDIO [Mozetic 1987] and sometimes in MOBAL [Wrobel 1994] — although they actually modify the interpreter rather than the program. The batch learner GCWS [Bain 1991b] invents an exception predicate and generalizes its definition thus defining it by non-unit clauses. Occasionally in MOBAL a positive invented predicate literal is added to a clause and a batch-learning technique is used to learn a definition for the invented predicate.

3.6 Interpreters

We have discussed ways to construct clauses of a program representing a theory. Now we turn to the question of computing the theory itself. In ILP the theory representation is usually a definite program, in which case the theory represented usually corresponds to a *least Herbrand model* [Lloyd 1987b, page 36] for the program. In practice, the meaning or theory represented by a logic program is computed by an *interpreter*. For this purpose we make use of the pure PROLOG interpreters described by Lloyd [1987b]: an *SLD-refutation procedure* for definite programs and an *SLDNF-refutation procedure* for normal programs.

Definition 3.7 (Derived, Selected, Resolvent) *Let* G' *be* $\leftarrow A_1, \ldots, A_m, \ldots, A_k$ *and* B *be* $A \leftarrow B_1, \ldots B_q$. *Then* G *is* derived *from* G' *and* B *using mgu* θ *if* A_m *is an atom, called the* selected *atom in* G'; θ *is an mgu of* A_m *and* A; *and* G *is the goal* $\leftarrow(A_1, \ldots, A_{m-1}, B_1, \ldots, B_q, A_{m+1}, \ldots, A_k)\theta$. G *is called a* resolvent *of* G' *and* B.
[Lloyd 1987b, page 40]

Definition 3.8 (SLD-derivation) *Let* P *be a set of definite clauses and* G *a definite goal. An* SLD-derivation *of* $P \cup \{G\}$ *consists of a (finite or infinite) sequence* $G_0 =$

$G, G_1, \ldots$ *of goals, a sequence* $C_1, C_2, \ldots$ *of variants of clauses of* P *and a sequence* $\theta_1, \theta_2, \ldots$ *of mgus such that each* G_{i+1} *is derived from* G_i *and* C_{i+1} *using* θ_{i+1}.

Each C_i *is a suitable variant of a clause in* P *so that* C_i *does not have any variables which already appear in the derivation up to* G_{i-1}. [Lloyd 1987b, page 41]

Definition 3.9 (SLD-refutation) *An* SLD-refutation *of* $P \cup \{G\}$ *is a finite SLD-derivation of* $P \cup \{G\}$ *which has the empty goal as the last goal in the derivation.* [Lloyd 1987b, page 41]

As a precise understanding is not crucial to this work, the reader is referred to [Lloyd 1987b] for formal but lengthy definitions of *SLD-tree*, *SLDNF-resolution*, *SLDNF-derivation*, *SLDNF-refutation*, and *SLDNF-tree*.

Our informal use of the term *proof* may be loosely taken to mean SLDNF-refutation although in practice an interpreter other than an SLDNF-refutation procedure may be used to construct it. We can then say that some atom or literal L is *provable* if $\forall L$ is true in the theory represented by a program, although it may not be valid in the target.

3.6.1 Negation as failure

Negation in logic programs is interpreted by the *negation as failure* rule of inference. The *completion* of a program, P, written $comp(P)$, provides a formal justification for it. Lloyd [1987b] gives a constructive definition attributed to Clark.

Two important soundness results about the completion are reproduced after preliminary definitions. The definitions for *answer* and *computed answer* equally apply to the special case of SLD-refutations of definite goals.

Definition 3.10 (Answer, Correct answer) *Let* P *be a normal program,* G *a normal goal. An* answer *for* $P \cup \{G\}$ *is a substitution for variables in* G. *Let* θ *be an answer for* $P \cup \{G\}$. θ *is a* correct answer *for* $comp(P) \cup \{G\}$ *if* $\forall(G_{\otimes}\theta)$ *is a logical consequence of* $comp(P)$. [Lloyd 1987b, page 80]

Definition 3.11 (Computed answer) *Let* P *be a normal program and* G *a normal goal. A* computed answer θ *for* $P \cup \{G\}$ *is the substitution obtained by restricting the composition* $\theta_1, \ldots, \theta_n$ *to the variables of* G*, where* $\theta_1, \ldots, \theta_n$ *is the sequence of substitutions used in an SLDNF-refutation of* $P \cup \{G\}$. [Lloyd 1987b, page 86]

Theorem 3.2 (Soundness of NAF) *Let* P *be a normal program and* G *a normal goal. If* $P \cup \{G\}$ *has a finitely failed SLDNF-tree, then* G *is a logical consequence of* $comp(P)$. [Lloyd 1987b, theorem 15.4]

Theorem 3.3 (Soundness of SLDNF-resolution) *Let P be a normal program and G a normal goal. Then every computed answer for $P \cup \{G\}$ is a correct answer for $comp(P) \cup \{G\}$.* [Lloyd 1987b, theorem 15.6]

These results require a safeness condition which constrains both the *computation rule* and the structure of the program and goal. By a *computation* of $P \cup \{G\}$ is meant an attempt to construct an SLDNF-derivation of $P \cup \{G\}$. We also speak of a computation of $\{G\}$ when P is understood by context. A computation of $P \cup \{G\}$ *flounders* if at some point in the computation a goal is reached which contains only non-ground negative literals [Lloyd 1987b, page 88]. By a *safe computation* is meant a computation which does not flounder.

One structure constraint on programs and goals ensuring safe computations is called *allowedness*; although other solutions are known [Foo, Rao, Taylor and Walker 1988, Edmondson 1988, Chan 1988]. For definite clauses alone, allowedness is sometimes called *range-restriction* [De Raedt 1992]. Our definition is slightly more restrictive than that of Lloyd [1987b, page 89], but the following result holds nevertheless.

Definition 3.12 (Allowed) *Let P be a normal program, C be a normal clause and G a normal goal. C is* allowed *if every variable that occurs in C occurs in a positive literal of $C_\otimes$. G is* allowed *if every variable that occurs in G occurs in a positive literal of $G_\otimes$. P is* allowed *if every clause in P is allowed. $P \cup \{G\}$ is* allowed *if every clause in P is allowed and G is allowed.*

Theorem 3.4 (Safeness and Groundness of Allowed Computations) *Let P be a normal program and G a normal goal. Suppose that $P \cup \{G\}$ is allowed. Then no computation of $P \cup \{G\}$ flounders and every computed answer for $P \cup \{G\}$ is a ground substitution for all variables in G.* [Lloyd 1987b, proposition 15.1]

3.6.2 Interpreter deficiencies

Ideally, we would like an interpreter to be sound, complete, efficient, terminating, and not restrictive on the language of programs. But there are many well-documented flaws in SLDNF-refutation procedures. When interpreters are intended for computer programming it is assumed that an intelligent user would understand the flaws and would take account of these in designing the programs. Hence the programmer chooses a representation which takes account of the *procedural* nature of an interpreter. Indeed, the programmer may even include predicates which have no declarative meaning but only procedural side-effects.

In MINERVA as in most ILP work the program represents only a declarative representation of knowledge and the "procedural" reading of the program should be irrelevant. The use of *cut* to prune infinite derivations is not usually available in ILP because of its lack of declarative meaning (except in the work of Bergadano, Gunetti and Trinchero [1993]). The ordering of clauses in the program and literals within a clause do not affect

the declarative meaning of a program (and hence the theory) and should be irrelevant to both the interpretation evaluated by the interpreter and the programs which can be learnt. Yet the ordering chosen can effect the soundness, completeness, and termination properties of the interpreter.

Unification, as introduced by Robinson [1965] is a basic element of SLD-resolution. Although most PROLOG interpreters implement a more efficient but unsound variation of the standard unification algorithm by omitting the *occurs check* [Lloyd 1987b], the consequent difficulties are easily rectified by implementing a sound unification algorithm instead.

But other difficulties are not so easily dismissed. Logical consequence even in the language of definite programs with function symbols is not decidable, so there is no sound interpreter which always terminates. Moreover, SLD-trees may have infinite branches even when the language is decidable.

Termination of an SLD-refutation procedure can be ensured by preventing it from indefinitely searching branches of an SLD-tree. This is done either by prematurely truncating the search of branches or by ensuring that there are no infinite branches — by varying the computation rule or restricting the structure of programs. Often some combination of these techniques is used.

Let us discuss some of the solutions of the literature. We begin by considering definite programs alone, then move on to the additional problems of normal programs. This will provide the background for the explanation in chapter 5 of the interpreter used in MINERVA.

3.6.2.1 Definite program interpreters

Recursion

Infinite branches of the SLD-tree arise through the use of recursion in programs. In particular, through the dependency relation of predicates.

Definition 3.13 (Depend) *A predicate symbol p* depends *on a predicate symbol q in a normal program if q occurs in the body of a clause in the definition of p or if p depends on some predicate symbol, r and q occurs in the body of a clause in the definition of r.*

In a *hierarchical* definite program no predicate symbol depends on itself and all SLD-derivations are finite [Lloyd 1987b, page 99] so any SLD-refutation procedure terminates. But hierarchical structure forbids recursion and so prevents representation of infinite theories, typical of number and list concepts. Furthermore some recursive clause structures, such as the transitive, symmetric and biconditional definitions noted by Covington [1985a] that are particularly problematic for termination, are particularly convenient for an ILP theory representation.

Clause and literal ordering

In logic programming advice, sometimes conflicting, is given to novice programmers to order clauses and atoms within clauses in a particular way [Bratko 1990, Clocksin and Mellish 1981, Sterling and Shapiro 1986]. These heuristics come without guarantee. Their partial success depends on the programmer's knowledge of the expected instantiation of goals at the place a clause is used in a derivation.

Cameron-Jones and Quinlan [1993] propose an ordering method integrated with learning for function-free definite clauses. For these clauses termination of the standard SLD-refutation procedure is guaranteed. The method is designed to be included in the ILP batch learner FOIL [Quinlan 1990] which learns by selecting one literal at a time to add to the right of the body of an increasingly specialized clause. It relies on input/output mode declarations for predicates and analysis of the input prior to learning to discover plausible ordering relations on the terms of the language. The ordering information is used to prevent antecedents which would violate the ordering from being added to the clause as it develops during learning.

The FOIL method is designed only for the restricted language of definite clauses without function symbols. It does not work for mutually recursive predicate definitions. Because it relies on detecting a static ordering on the language terms, it is inappropriate for a learner where the language is not declared in advance and the input is only available incrementally. When the program is to be revised incrementally, this kind of technique undesirably constrains the syntax of any newly acquired clause according to the syntax (as distinct from the declarative meaning) of the existing program. If this effect is to be alleviated, then existing clauses in the theory may need to be restructured on observation of each new example.

The approach is closely related to the techniques in the logic programming literature which rely on prior mode declarations and analysis of a complete program to determine partial orderings on terms which are used to restructure the program. The formally analysed method of Plümer [1990] also supports function symbols.

Depth bound

The most frequently used technique in ILP is to impose a finite *depth bound* on the length of SLD-derivations or the depth of the proof tree (definition 3.19) and to terminate with failure when that limit is reached. This method is used for example in the incremental ILP learners MIS [Shapiro 1982], MPL [De Raedt et al. 1993], CIGOL [Muggleton and Buntine 1988] and ITOU [Rouveirol 1991b]. Modifying the SLD-refutation procedure in this way guarantees termination and will sometimes enable SLD-refutations to be found which would otherwise be missed. Unfortunately it also compromises the completeness of the SLD-refutation procedure; although some otherwise missed refutations will be found, some otherwise found refutations will be missed (namely those of length greater than the bound). Further, the procedure may waste a lot of time investigating fruitless branches of the SLD-tree until the bound is reached.

Fair search

The standard SLD-refutation procedure with *unfair* depth-first search of the SLD-tree may descend an infinite branch of the SLD-tree leaving finite success branches unexplored, thus missing answers. If the language is decidable (for example, if there are no function symbols) then any *fair* search of any SLD-tree (such as breadth-first search) guarantees that no answers will be missed, at the expense of memory efficiency. But even then the procedure will not terminate when there are infinite derivations in the SLD-tree.

In iterative deepening search a depth bound is initially imposed to bound the search but when search at a given bound is exhausted the bound is incrementally increased and the search recommenced. Stickel [1984] proposes replacing the depth-first search strategy of standard SLD-refutation procedures with an iterative deepening strategy. He argues that the cost of searching at depths less than n before first finding a refutation of length n will "probably not be unacceptably high relative to the cost of just searching at level $n+1$". He also suggests a minor refinement: a derivation may be failed before reaching the length of the current bound when the sum of the length of the derivation so far and the number of atoms in the current goal exceeds the depth bound.

The iterative deepening approach can overcome the problems of an unfair search strategy, but will not terminate on SLD-trees with infinite branches. It seems to have no advantage over the fixed depth bound method whenever the initial goal is ground and so only one answer is sought. When multiple answers are sought to a non-ground atomic initial goal, the iterative deepening approach enables return of answers in an "easiest" first order — those that are computed in the shortest derivations are computed earliest.

Varying the computation rule

Sometimes a computation via a particular computation rule generates an SLD-tree with infinite branches although the SLD-tree via a different computation rule would be finite. Improvements to termination properties made by re-ordering literals in clauses to suit the standard computation rule may be mimicked or bettered by allowing a more flexible computation rule instead. Naish shows how the computation rule may be directed by *when* or *wait* declarations and how these can often be automatically generated [Naish 1985, Naish 1986]. Like the automatic reordering approaches, this method requires intensive analysis of a program which may have to be repeated every time a program is revised. It has the advantage that it can also be used for normal programs, but the method will not always result in terminating computations even for a decidable language.

Loop checks

Also in the logic programming literature, the termination problem is addressed by pruning away infinite branches of the SLD-tree as they are explored. This is done by supplementing the standard depth-first SLD-refutation procedure with a *loop check*. At each step of a derivation the partial derivation up to that point is examined and that information is used to determine whether to continue investigation of the present branch of the SLD-tree, or to prune the branch at that point and to consider the derivation as *failed* at that node. A loop check is *sound* if does not cause answers to be missed. It is *complete*, if it guarantees that every search (with a given computation rule and search rule) for SLD-derivations finitely terminates. Using this terminology, the depth bound method already discussed is a complete but unsound loop check.

It is not possible for a loop check to be both sound and complete for definite programs with function symbols because the language is not decidable. Indeed, Bol, Apt and Klop [1991] show that even in the absence of function symbols there cannot be a sound and complete loop check which uses only the information in the SLD-derivation.

Inventing a loop check is not easy. Covington's [1985a] loop checks are shown to be unsound by Poole and Goebel [1985]. The amended loop check presented by Covington [1985b] and another by Nute [1985] can also be shown to be unsound by referring to the counter-example of Bol et al. [1991, page 46].

Bol et al. [1991] present a range of alternative loop checks designed for languages without function symbols. They analyse the relative *strength* of many of them; a loop check is stronger than another if it detects all loops detected by the other. They also prove soundness for many of them. The strongest loop check is called the *instance of an atom* loop check but it is not sound.

The strongest sound loop check is named *subsumes instance of resultant multiset (SIR_M)* check, and is employed in MINERVA. The definition describes the set of SLD-derivations which are pruned away by checking.

Definition 3.14 ($\mathbf{SIR}_M$ loop check) *Let L be a set of SLD-derivations. Then remsub(L) is the subset of L defined by $\{D \in L \mid L$ does not contain a proper sub-derivation of $D\}$.*

The subsumes instance of resultant multiset (SIR_M) *check is the set of SLD-derivations remsub($\{D|D$ is the sequence of goals $\leftarrow G_0, \leftarrow G_1, \ldots \leftarrow G_k$, mgus $\theta_1, \ldots \theta_k$, and program clauses $C_1, \ldots, C_k$ such that for some i with $0 \leq i < k$, there is a substitution σ with $G_i\sigma \subseteq_M G_k$ and $G_0\theta_1, \ldots, \theta_i\sigma = G_0\theta_1, \ldots \theta_k\}$).* [Bol et al. 1991, pages 39 and 60]

Implementing this loop check requires checking as each successive goal in a derivation is generated, say $\leftarrow G_k$ at step k, whether G_k is a superset of an instance of some earlier goal say, $G_i\sigma \subseteq G_k$ for $i < k$. If so, it must be confirmed that the kth-step instantiation of the original goal ($G_0\theta_1, \ldots \theta_k$) is identical to the instance of original goal at the ith step by σ, $G_0\theta_1, \ldots, \theta_i\sigma$. If these conditions are met then the current branch of the

SLD-tree is treated as a failed derivation.

The SIR_M loop check is shown to be sound for function-free definite programs. Although it is apparently sound for all definite programs, it is not complete even for function-free programs as demonstrated by the following example.

Example 3.5 (SIR_M loop check) Let P be the program

$$
\begin{array}{l}
s(zero, one)\leftarrow \\
s(one, two)\leftarrow \\
s(two, three)\leftarrow \\
l(U,W)\leftarrow l(U,V),s(V,W) \\
l(X,Y)\leftarrow s(X,Y)
\end{array}
$$

Following is an initial fragment of an infinite derivation for $P \cup \{\leftarrow l(X,Y)\}$ showing respective sequences of goals, program clauses, and *mgus* which cannot be pruned by the SIR_M loop check.

goals	program clauses	*mgus*
$l(X,Y)\leftarrow$		
$l(X,V_1),s(V_1,Y)\leftarrow$	$l(U_1,W_1)\leftarrow l(U_1,V_1),s(V_1,W_1)$	$U_1/X, W_1/Y$
$l(X,V_2),s(V_2,V_1),s(V_1,Y)\leftarrow$	$l(U_2,W_2)\leftarrow l(U_2,V_2),s(V_2,W_2)$	$U_2/X, W_2/Y$
$l(X,V_3),s(V_3,V_2),s(V_2,V_1),s(V_1,Y)\leftarrow$	$l(U_3,W_3)\leftarrow l(U_3,V_3),s(V_3,W_3)$	$U_3/X, W_3/Y$
$\vdots$	$\vdots$	$\vdots$

□

Bol, Apt and Klop show that the loop check is complete for some classes of function-free definite programs all of which permit some recursion. These include the so-called *svo* programs in which each variable of each clause occurs in its body at most once. When a computation rule which selects left-most atoms is used, *cycle-restricted* programs are also included. In clauses of a cycle-restricted program, of the atoms in the body only the rightmost may have a predicate symbol that depends on the predicate symbol of the head. Independently of the computation rule, the loop check is also complete for programs of definite function-free *nvi* clauses.

Definition 3.15 (nvi) *Let C be a definite clause. Then C is non-variable introducing* (nvi) *if every variable in $C_\otimes$ also occurs in $C_\odot$.* [Bol et al. 1991, definition 5.10]

3.6.2.2 Complexity checks

Smith, Genesereth and Ginsberg [1986] address the problem of non-termination in languages with function symbols. They describe a problem called *divergent inference* which occurs when subgoals in a derivation become increasingly more complex with deeper terms.

Example 3.6 (Divergent inference) Let P be $\{n(zero)\leftarrow, n(X)\leftarrow n(s(X))\}$. Then the computation of $P \cup \{\leftarrow n(s(zero))\}$ does not terminate and in each successive goal of the derivation the function symbol is applied once more. □

Let us define what we mean by the *depth* of a term.

Definition 3.16 (Term Depth) *Let t be a term. If t is a variable or a constant then the term depth of t is zero. Otherwise, if t is the term $f(t_1, \ldots, t_n)$ for some function symbol f and terms t_i ($i = 1, \ldots, n$), then the term depth of t is one more than the maximum term depth of each t_i ($i = 1, \ldots, n$).*

Provided that for every rule in the program, the depth of each term in the head is equaled or exceeded by the depth of a term in the body, Smith et al. [1986] claim that a derivation may be terminated with failure at any goal in which occurs a term with a depth exceeding the depth of all terms in unit clauses in the program. In the example, the computation of $P \cup \{\leftarrow n(s(X))\}$ could terminate with failure immediately. But this restriction is very strong: a clause like $n(s(X))\leftarrow n(X)$ could not occur in such a program.

The related *h-conform* approach of De Raedt [1992] aims to solve the same problem, bounding derivations by bounding the term depth of goals in a derivation. It is, in a way, the converse of the restriction of Smith, Genesereth and Ginsberg, being based on *term embedding*. Our definition for term embedding is a more general one than De Raedt's [1992] similar *complexity of a variable in a term*.

Definition 3.17 (Term Embedding) *Let t be a term. Then the* term embedding *of t in t is zero. Let t and $f(t_1, \ldots, t_n)$ be distinct terms such that t occurs in $f(t_1, \ldots, t_n)$. Then the* term embedding *of t in $f(t_1, \ldots, t_n)$ is one more than the maximum of the term embedding of t in each of $t_1, \ldots, t_n$ that t occurs in. Let t be a term and $p(t_1, \ldots t_n)$ be an atom that t occurs in. Then the* term embedding *of t in $p(t_1, \ldots t_n)$ is the maximum term embedding of t in each of $t_1, \ldots, t_n$ that t occurs in.*

A definite program is *h-conform* if clauses are *nvi* (definition 3.15), allowed, and the term embedding of every variable in the head of a clause exceeds or equals the maximum term embedding of the same variable in the antecedents of the clause. A clause like $n(s(X))\leftarrow n(X)$ can occur in such a program, but $n(X)\leftarrow n(s(X))$ cannot.

Unfortunately termination guarantees for SLD-refutation on such programs are confined to ground goals, and the lack of support for *nvi* clauses is debilitating. De Raedt gives termination guarantees for such programs interpreted by a forward-chaining interpreter.

Non-SLD-interpreters

A common approach in the logic programming literature replaces the standard SLD-refutation procedure by an OLDT-refutation procedure [Plümer 1990, Smith et al.

1986]. Although it always terminates for programs with finite Herbrand models this is at the expense of space efficiency. Similarly, forward-chaining interpreters like the mapping T_P of van Emden and Kowalski [Lloyd 1987b, page 37], can be guaranteed to terminate in this circumstance, but the space requirements make them suitable only for small theories.

3.6.2.3 Normal program interpreters

When negative literals are permitted in the bodies of clauses the problem of non-termination is compounded. The following simple example is due to Lloyd [Lloyd 1987b, page 98].

Example 3.7 Let P be the program $\{(r \leftarrow p), (r \leftarrow \sim p), (p \leftarrow p)\}$. Then the identity substitution is a correct answer for $comp(P) \cup \{\leftarrow r\}$ but there is no SLDNF-tree. There is an infinite branch corresponding to the computation of $P \cup \{\leftarrow p\}$. The search for the other branch does not terminate because there is neither a finitely failed SLDNF-tree nor an SLDNF-refutation for $P \cup \{\leftarrow p\}$. □

Soundness results for SLDNF-derivation depend on a safe computation rule. In this section we assume a safe computation rule is used rather than the standard left-most selection computation rule more commonly implemented in PROLOG interpreters.

Language restrictions

If a program is hierarchical then the SLDNF-tree for any goal via a safe computation rule is finite and so all computations terminate [Lloyd 1987b, page 99]. But the hierarchical structure is very restrictive: there can be no recursive predicate definitions.

Cavedon [1991] and Apt and Bezem [1990] investigate classes of normal programs which do allow limited use of recursion. In particular, the *acyclic* allowed class of programs is quite general and has many attractive properties, including that SLDNF-resolution is complete for such programs and allowed goals [Cavedon 1991]. Unfortunately there remains no guarantee that computations will terminate, not even when the set of correct answers is finite.

The attractive properties for these program classes are conditional on confirming that a program is a member of the class. For example, for the acyclic class this is done by building a dependency graph of the ground atoms in the Herbrand base with respect to the clauses in the program, to develop an ordering on the ground atoms. This approach has the same difficulty for an incremental ILP learner as the related approach of FOIL described previously. In addition to the computation overhead of maintaining the graph, incremental learning performance becomes increasingly dependent on the ordering of input examples, since a hypothesised clause which fails to meet the acyclic condition in the context of the program at some time may satisfy the condition at a later time when some extant clauses are removed or altered.

Loop checks

Again we turn to the need for a mechanism which can avoid non-termination on-the-fly, as derivations are being computed. The loop check algorithms developed for SLD-resolution do not easily extend to SLDNF-resolution. Certainly they may be applied without alteration to any part of an SLDNF-derivation that is an SLD-derivation as in the first example below. But it will not always work, as in the second example.

Example 3.8 In the program of the previous example (3.7) the SLD-derivation for $P \cup \{\leftarrow p\}$ would terminate with failure and the resulting SLDNF-tree for $P \cup \{\leftarrow r\}$ would then have two finite branches. One of these would be an SLDNF-refutation with the identity answer and the other a failure branch, so the correct answer would be produced and the search would terminate. □

Example 3.9 Let P be $\{p \leftarrow \sim p\}$. $comp(P)$, representing the formula $p \wedge \sim p$, is inconsistent. Let Q be $\{(p \leftarrow \sim q), (q \leftarrow \sim p)\}$. $comp(Q)$ is disjunctive, representing the formula $(p \vee q) \wedge \sim (p \wedge q)$. The searches for each of the SLDNF-trees for $P \cup \{\leftarrow p\}$ and for $Q \cup \{\leftarrow p\}$ do not terminate, and there is no SLD-derivation of length greater than one constructed during the search and hence no opportunity to prune the search with the SLD-derivation based loop checks. □

Whether the sound SLD-derivation loop checks could be extended to work in this situation would be an interesting research question. Perhaps some of the loop checks of Stickel [1984] could be used, but these are sometimes at odds with the semantics of the program completion. Given the difficulty of the discovery of sound loop checks for SLD-derivations, it is unlikely to be a trivial matter.

Even extending the unsound depth bound loop checks for normal programs is difficult. The interpretation of negation as "failure to prove" is replaced by "failure to prove within the depth bound". When a negative literal is selected, if the depth bound for refutations of the complementary atom is less than the initial depth bound, then inconsistent "boundary" behaviour could arise. On the other hand, if it is restarted to the initial depth bound, so that the depth bound uniformly limits the depth of search of branches in SLDNF-trees, then the technique does not guarantee termination.

Extending the iterative deepening approach to interpretation of normal programs inherits these difficulties. When a negative literal is selected, what initial depth bound is appropriate for the interpretation of its complementary atom? Every negative literal in a goal can be proved at a sufficiently small depth bound, so an iterative deepening approach could have every negative goal succeed immediately.

Non-SLDNF interpreters

It is also not obvious how to extend the OLDT or forward-chaining interpreters to deal with negation. Certainly it would require a different interpretation of the meaning of

negation. An extension to GOLEM [Bain 1991a] which uses a forward chaining interpreter is able to evaluate negative literals in goals in this way but only with respect to a definite program.

Some of the difficulties with negation stem from the concept of negation as failure: a closed world assumption. An alternative theory representation could allow explicit negation and interpret its theories with a classical theorem-prover. Recalling that a program is intended to represent a theory, a three-valued logic is suggested: a ground atom could be true, false or unknown. De Raedt [1992] presents some introductory work on this idea. MOBAL [Morik, Wrobel, Kietz and Emde 1993] includes an interpreter for a theory representation which permits negative literals in the body of each clause as well as permitting the head to be a negative literal, but having no function symbols. Stickel [1984] discusses a theorem prover for an even more general clausal theory representation based on SLD-refutation technology and including some loop checking capability.

Later, in chapter 5, the ideas presented here are used to define an interpreter for MINERVA.

3.7 Declarative Diagnosis

When an example is observed that is not provable it may indicate an error in the definition of the predicate of the example. But the definition of the example predicate may be correct and complete — the error may be due to incorrect or incomplete definitions of predicates on which the example predicate depends. Diagnosis determines which predicate definition is either incomplete or incorrect. More specifically, it determines either an *uncovered atom*: an atom which is not covered in the target by any clause of the program; or *false clause* of the program together with a false ground atom which it covers in the target.

In this section we survey the literature from which MINERVA's diagnosis procedure derives.

3.7.1 Diagnostic tools

A method for finding an erroneous predicate definition in a definite program, called *declarative diagnosis*, was first introduced by Shapiro as *algorithmic debugging* [Shapiro 1981, Shapiro 1982, Sterling and Shapiro 1986]. The techniques have since been enhanced in various ways in an endeavour to produce practical debuggers for logic programs [Lloyd 1987a, Lloyd 1987b, Naish 1992]. Declarative diagnosis needs an *oracle* to answer questions posed by the diagnoser about the intended interpretation of the language of the program. The role of oracle is played by a teacher ready to answer questions, by the environment when the asking of a question is viewed as the performance of an experiment, or by a programmer when the diagnosis is used as a program debugging tool.

What kind of questions are needed to perform declarative diagnosis? Depending on the intended application, different forms of questions may be more or less reasonable. The "easiest" question is a *membership query* which asks whether an object in the language of observation is a member of a particular concept.

Definition 3.18 (Membership query, Answer) *A membership query asks for the classification of a ground atom in the language of observation. For ground atom A, the* answer *to the membership query about A, is the boolean value (*true *or* false*) assigned to A in the target.*

The design of a diagnosis algorithm for a time-parsimonious active learner must take account of a need to minimise the number of questions asked and to make those questions sufficiently "easy" for the environment to answer. Some other kinds of queries sometimes asked in diagnosis include a *satisfiability query* which asks whether there exists a true instance of a given atom; an *instance query* which asks for true instances of a generally non-ground atom; or an *incompleteness query* which asks whether a given set of universally quantified atoms are the only true instances of an atom [Naish 1992].

There are two components of declarative diagnosis for definite programs, *contradiction backtracing* and *missing answer diagnosis*. Contradiction backtracing is invoked to deal with errors of commission which occur when for definite program P and atom A there is an SLD-refutation for $P \cup \{\leftarrow A\}$ and $\exists A$ is false. It finds a false clause in P. Missing answer diagnosis is invoked to deal with errors of omission which occur when there is no SLD-refutation for $P \cup \{\leftarrow A\}$ and A is true. It finds an *uncovered* atom, sometimes called a missing answer, of P.

Shapiro [1982] shows how the techniques can be combined to support debugging of normal programs. If there is no SLDNF-refutation of $P \cup \{\leftarrow \sim A\}$ and A is false then contradiction backtracing should be applied to the SLDNF-refutation of $P \cup \{\leftarrow A\}$. On the other hand, if there is an SLDNF-refutation of $P \cup \{\leftarrow \sim A\}$ and A is true then missing answer diagnosis should be applied to $P \cup \{\leftarrow A\}$.

3.7.2 Contradiction backtracing

The contradiction backtracing algorithm searches through the clauses in an SLD-refutation, making membership queries for each atom in a clause until a ground instance of a clause is found with a false head and true antecedents. Such a clause covers the false ground atom that is the head instance. Alternative forms of the algorithm [Shapiro 1982, Sterling and Shapiro 1986, Lloyd 1987a, Lloyd 1987b] differ mainly in the order in which the clauses of the refutation are searched. Shapiro argues that the divide and query (binary search) approach asks the fewest questions in the worst case, but Lloyd points out good reasons for employing the top-down strategy attributed to Av-Ron: it is easier to incorporate heuristics to aid the search in the top-down approach, and even without them the number of questions asked may be fewer. Lloyd shows that the number of questions asked by either strategy is quite similar in an "average" case.

The top-down strategy, as implemented in the contradiction backtracing diagnoser in MINERVA, is best explained in terms of its search of the *and*-tree corresponding to the successful SLD-refutation, called an *SLD-proof tree*. Here we define a more general *proof tree* which will be of use later on; an SLD-proof tree is a just a proof tree with no negative literals occurring in it. For simplicity of explanation we assume that the initial goal of any refutation comprises only one literal. This is generally the case in ILP because it is invoked in response to an error discovered from an atomic example.

Definition 3.19 (Proof Tree) *Let the SLDNF-refutation of $P \cup \{\leftarrow A_0\}$, where P is a normal program and A_0 is a literal, be the sequence of goals $G_0, G_1, \ldots, G_n$, the sequence of clauses and ground negative literals $C_1, C_2, \ldots, C_n$, and the sequence of substitutions $\theta_1, \theta_2, \ldots, \theta_n$. Let θ be the composition $\theta_1\theta_2\ldots\theta_n$. The* proof tree *of $A_0\theta$ is the tree with root $A_0\theta$, and A_0 is the selected literal in G_0.*

If atom A'_j ($j \in \{0, \ldots, n-1\}$) is a node of the tree then there is an atom A_j such that $A_j\theta = A'_j$ and A_j is the selected atom in some goal of the refutation, say in G_k. The node A'_j has a child node for each literal in the body of the clause $C_{k+1}\theta$. On the other hand, if negative literal $\sim A'_j$ ($j \in \{0, \ldots, n-1\}$) is a node of the tree then there is a finitely-failed SLDNF-tree for $P \cup \{\leftarrow A'_j\}$ and the node has no children.

The top-down strategy searches the SLD-proof tree commencing with the initial atomic goal instance. The atom is assumed to be false because otherwise there is no need to invoke contradiction backtracing. A membership query is asked about each child atom (siblings in the tree), in an unspecified order, until a false child is found. The proof tree of the false child is then examined in the same fashion. If a node has no false children then the corresponding clause is false.

It is possible that the atoms in a proof tree are not all ground. In this case, Shapiro suggests that variables remaining in the proof tree may be consistently replaced by arbitrary ground terms. Alternatively, the oracle could be asked to supply a false ground instance of a non-ground atom.

Lloyd [1987b] gives soundness and completeness results for contradiction backtracing. Paakki [1994] suggests that some improvements may be made using extra-logical information about the program predicates: information unavailable to MINERVA.

3.7.3 Missing answers

Missing answer diagnosis is concerned with diagnosing an error in a definite program P, that arises from finite failure of every branch of the SLD-tree for $P \cup \{\leftarrow A\}$ where A is a true atom. The problem is not that some clause in the program is false, but rather that some concept definition is incomplete.

Missing answer diagnosis is important for a learner with a program for which more than one concept definition is potentially incomplete. This would generally be the case for an incremental multiple-concept learner. By assuming that incompleteness is due

only to the definition of the concept of the example, an opportunity for learning would be missed.

Example 3.10 (Missing answer diagnosis) Let P be the program

$$grandfather(X, Y) \leftarrow father(X, Z), mother(Z, Y)$$
$$father(paul, paula) \leftarrow$$

Let A be the atom *grandfather(paul, karen)*, presented as a positive example. Without missing answer diagnosis a learner might add the clause *grandfather(paul, karen)*← to P. But if *mother(paula, karen)* is also true, then the first clause in P already covers A in the target. On the other hand, missing answer diagnosis could determine that *mother(paula, karen)* is an uncovered atom. Adding *mother(paula, karen)*← to P would also make A provable. □

Missing answer diagnosis asks questions about the clauses in a program until a true atom is found for which there is no clause in the program that covers the atom in the target.

The search for the uncovered atom commences with an atom that is an unprovable positive example. Each clause with a head which unifies with the atom is checked to determine whether it covers the atom. This is done by asking questions about the antecedents of the clause instance made by unifying the head with the atom. If the questions determine that a clause does cover the atom then there must be an antecedent which is not provable and that atom becomes the focus of a recursive application of the procedure. Otherwise, there is no covering clause for the atom and so it is diagnosed uncovered.

The missing answer diagnosis algorithms [Shapiro 1982, Sterling and Shapiro 1986, Lloyd 1987a, Naish 1992] differ in the order and number of the questions asked, the form of questions asked, and the class of logic programs to which they are applicable. Naish [1992] provides a detailed analysis, addressing question-efficiency, soundness and completeness of many variations of these algorithms.

The questions posed by a missing answer diagnoser are typically more complex than the membership queries required for contradiction backtracing. Missing answer diagnosers ask a combination of instance queries, satisfiability queries and incompleteness queries. Incompleteness and satisfiability queries can be used to reduce the number of instance queries required or to obviate them entirely, but like instance queries they are very demanding of the oracle. By using incompleteness queries alone an incomplete definition can be located but not an uncovered atom; the diagnoser could determine an atom for which some unknown instance is uncovered.

As noted by Naish, Shapiro's [1982] MIS assumes that there are always a finite number of true ground instances of any atom. This means that although only instance queries are made, the oracle need only give ground answers (possibly all of them). Without this assumption, which restricts the class of programs in which the algorithm finds an uncovered atom, it is necessary to empower the oracle with the ability to answer

with universally quantified (non-ground) instances. Even then, when the program language permits function symbols, a finite number of universally quantified atoms may be insufficient [Naish 1992].

Of Naish's [1992] diagnosers, N2 requires fewest membership and instance queries. Unlike some others, it is not distracted by checking the truth of all covered atoms encountered in the search. This distraction may be merited when the diagnoser is used for computer programming assistance but is not appropriate for MINERVA because the focus on the failing example is lost. If distracted in this way the diagnoser would either ask unnecessary questions or diagnose a false clause which, when corrected, could not repair the original problem for which the diagnoser was invoked. The missing answer diagnoser of MINERVA is based on N2.

3.8 Summary

This chapter has introduced the basic terminology and some advanced tools of the fields of machine learning, logic programming and inductive logic programming, setting a context for a description of MINERVA. Of particular importance are the concepts of generalizing by absorption, interpreting programs by SLDNF-refutation procedures, and diagnosing errors by contradiction backtracing and missing answer diagnosis.

4

Incremental Learners and Theory Revision Systems

4.1 Introduction

This chapter surveys concept learning and theory revision systems. In the first sections, a few of the better known and influential ILP learners are described. They all use first order definite program theory representations, although the form of the clauses is restricted in various ways. Afterwards a number of other learners are introduced. These are not cast in the ILP framework but have made contributions to the theory revision problem in machine learning.

In the succeeding section more detail on the learners is given in the context of discussion of general approaches to common theory revision problems embedded in those systems. Finally, the chapter briefly identifies the relationship of inductive concept learning to a body of work dealing with rational changes of belief.

4.2 Incremental ILP Learners

4.2.1 MARVIN

Sammut's MARVIN [Sammut 1981a, Sammut 1981b] is an active relational, incremental ILP learner which is a forerunner to MINERVA. MARVIN's success in learning complex theories relies on the use of the absorption operator for generalization and a very *helpful teacher* [Salzberg, Delcher, Heath and Kasif 1991]. Given a positive input example, MARVIN constructs a starting clause comprising the example in the head and some other

ground facts in the body which are effectively selected by the teacher (by deliberate ordering). The constants in the starting clause are replaced by variables and the clause is then generalized by absorption with some clause from the background program. The generalized hypothesis is tested by a method similar to that of MINERVA and, if it is found to cover some false atom, is restricted by adding back some of the optional atoms and testing again. The language of hypotheses is functor-free, although the ground facts of the program may use constant symbols.

The success of MARVIN is demonstrated by its ability to learn a range of concepts including list concepts, number concepts and blocks-world problems. In order to achieve this, MARVIN depends on the teacher to provide examples in a strict order, requiring of the teacher a good understanding of both the target and MARVIN. If the order is not adhered to, MARVIN may learn false clauses and there is no capability for correcting them. Later, Sammut and Banerji [1986] suggest the use of contradiction backtracing for diagnosing false clauses in MARVIN.

4.2.2 MIS

Shapiro's incremental MODEL INFERENCE SYSTEM (MIS) [Shapiro 1981, Shapiro 1982], a contemporary of MARVIN, is based on a complete search of the clauses of the logical language of the theory representation. When the current theory does not include a positive example MIS uses missing answer diagnosis to find an uncovered atom. It then searches an enumeration of the clauses in the language of hypotheses for a clause which covers the atom to add to the developing program. Clauses which cover negative examples are found by contradiction backtracing and removed. After a clause is removed, a check is made that all positive examples so far remain covered and new covering clauses are added as necessary. After a clause is added, a check is made that all negative examples so far remain uncovered, and other clauses are removed if necessary.

Completeness of the procedure relies on the ordered search through the enumeration of clauses. Clauses added to the theory are chosen from *refinements*, that is minimal specializations of previously removed clauses. The refinements of the empty program are all the atoms of the language with a most general instantiation. The refinements of a clause are formed by unifying pairs of distinct variables, replacing variables by constants or function symbols with a most general instantiation or by adding a new atom in its most general instantiation to the body. In order for refinements to be evaluated, the complete alphabet of the hypothesis language must be declared to the learner in advance of any learning.

Clauses considered for addition are tested for their ability to cover a missing atom by the *adaptive* strategy which is similar to the experimental approach of MINERVA. The interpretation in which a clause must cover the missing atom is the current theory combined with the complete set of known true facts (from observations and responses to previous questions).

A major deficiency of MIS as an active incremental learner is its requirement that the complete alphabet of the language of observation is declared in advance of any

learning. Ling [1991] describes a variant which can support a growing language of constant symbols but which requires initial declaration of the other symbols of the language. Another shortcoming stems from its use of the θ-subsumption model of generality. As noted by Buntine [1988], the use of a general to specific search strategy coupled with the failure to use background knowledge to structure the search requires exploration of larger than necessary search spaces.

The major contribution of MIS to ILP is the introduction of the declarative diagnosis algorithms (called *algorithmic debugging* by Shapiro). Related techniques have been employed in many other revising learners.

4.2.3 CIGOL

Muggleton and Buntine's [1988] CIGOL identified the first-order inverse resolution operators. CIGOL generates hypotheses to cover positive examples initially by least general generalization (by θ-subsumption), and then by further generalization using absorption. Complexity-based heuristics are used to order plausible hypotheses, which are then tested by a method similar to the method of MINERVA and further confirmed by the user before admission to the theory. Intra-construction is occasionally used to invent new predicates, and the definition of the proposed invented predicate is confirmed by the user before admission to the theory.

There is no mechanism for correcting errors. Although CIGOL's hypothesis language supports function symbols, restrictions imposed on the use of the inverse resolution operators effectively restrict the hypothesis language in other ways.

4.2.4 CLINT

CLINT [De Raedt, Bruynooghe and Martens 1991, De Raedt and Bruynooghe 1992, De Raedt 1992] is an incremental learner in the ILP framework that addresses some of the shortcomings of MIS. CLINT orders the clauses in the hypothesis language by a sequence of syntactic constraints on the clauses, rather than by generality. The clauses of the hypothesis language are definite functor-free and allowed — although these constraints could be relaxed in an extended sequence [De Raedt 1992].

A suitable clause for covering each new positive example is selected from the earliest possible language in the sequence. There may be many such clauses. A clause is suitable if it covers the example and covers no known negative examples. This clause is then generalized by deleting atoms from the body (of which there may be very many) and testing. The testing technique is similar to that of MARVIN.

Covered negative examples, including any found during the testing procedure, prompt contradiction backtracing to locate a false clause that is then retracted. The positive examples which were previously covered by that clause are treated as new examples for generalization. Finally, every positive and negative example with a concept description which refers to the concept of the removed clause is treated again as a new example.

When supplemented with CIA, the CLINT-CIA learner can also invent new predicates by recognizing common second-order schemata patterns of existing clauses. New predicates and their definitions are confirmed by the oracle before being assimilated.

The efficiency of learning by CLINT is highly dependent on the order of examples presented. At worst, CLINT may be forced to reconsider almost every example already observed whenever a new example is observed. Because of assumptions about the form of the initial background knowledge (the definition of so-called basic predicates) and the language of hypotheses, CLINT cannot learn its own background knowledge. The predicate invention technique (combined with the use of contradiction backtracing where the teacher may be asked about the truth of invented atoms) demands much of the teacher. Nevertheless CLINT is demonstrated to learn some complex concepts, it does identify in the limit and it is claimed to require fewer examples than MIS.

4.3 Batch ILP Learners

Significant empirical success in ILP has been achieved for single-concept batch learning by Muggleton and Feng's [1990] GOLEM and Quinlan's FOIL and their derivatives. GOLEM is a specific to general algorithm that is based on constructing a number of approximate least general generalizations (by relative subsumption) of some of the positive input examples. For constructing the generalizations the background theory is approximated by a set of ground atoms which are consequences of the program derivable in a limited number of resolution steps. Each least general clause is then successively generalized further by dropping literals which do not help to discriminate any negative examples, and by using the information of user-supplied *mode* declarations. A hill-climbing algorithm is used to determine the grouping of examples for constructing combined least general generalizations. The clauses of the first-order hypothesis language of GOLEM are restricted by a syntactic property known as *i-j determinism*. Nevertheless results reported for GOLEM include accurate learning of list concepts and qualitative physics problems.

In contrast, FOIL [Quinlan 1990, Quinlan 1991] is a general to specific learner which proceeds by successively constructing clauses by a covering method. Atoms are added to the body of an increasingly specialized clause by an *information content* heuristic which measures the contribution of an atom to discriminating positive and negative examples covered by the clause. Background knowledge, in the form of a ground unit clauses, is used to supply the language for the atoms to add and to evaluate the cover of a clause under construction. The hypothesis language is functor-free.

Nevertheless, FOIL has become something of a *de facto* standard in ILP with which other learning systems are compared. It has become a basis for many other learning systems which attempt to address its shortcomings [Džeroski and Bratko 1992, Pazzani, Brunk and Silverstein 1991, De Raedt et al. 1993].

MPL of De Raedt et al. [1993] addresses the neglected problem of batch learning of multiple concepts in a functor-free language. The authors analyse the special difficulties

of learning definitions for multiple, interrelated concepts and argue that it is not easy to adapt a single-concept learner like FOIL to solve this problem. MPL assumes there is a partial program correctly defining "basic" predicates to which definitions for "learned" predicates must be added. The general to specific learning algorithm is based on FOIL but provides better support for mutually-dependent multiple predicate definitions by interleaving the learning of clauses for different concepts. Clauses once assimilated may be removed by later revision. Although some successful learning has been demonstrated by MPL, the concepts involved were few and simple, and the complete example set quite small.

The relational batch learners Richards' FORTE [Richards and Mooney 1991, Richards and Mooney 1995] and Wogulis's [1991] AUDREY approach very similar problems of domain-independent revision. They use different heuristic operators to revise an initial incorrect theory according to a batch of examples. Like for MPL, predicates are partitioned into "basic" ones for which user-given definitions are assumed correct and others which require correction.

4.4 Other Revising Learners

The problem of theory revision in inductive learning has been addressed in frameworks other than ILP: the fields of knowledge base refinement, explanation-based learning, operator planning and scientific exploration. In this section several reported learning systems of these frameworks are introduced in preparation for discussing their approach to some of the common problems in theory revision.

Although the learners surveyed have distinct representation languages for theories, an attempt is made to describe them in terms of an equivalent logic program representation. Sometimes this requires ignoring some aspects of the revision process. Thus the learners are described as *propositional*, meaning they learn ground clauses, or *relational*, meaning they learn definite function-free clauses, or *first-order*, meaning they learn definite clauses with function symbols. When representations permit negated literals in the body of a clause, they are described as *normal*. Some of the propositional and relational learners have special domain-dependent mechanisms for handling particular first-order concepts, such as integers.

4.4.1 Knowledge base refinement

Most work in theory revision has concentrated on the problem of *knowledge base refinement* for so-called *expert systems* or in the *explanation based learning* (EBL) framework. Learners aim to correct faulty theories previously constructed by experts by revision with respect to batches of empirical examples.

This work often assumes that the sentences in which the theory is expressed are supplemented with numeric labels attached to each called rule *certainty factors*. The factor is used as a preference criteria in deductive inferencing, intended to represent some-

thing like confidence, belief, or probability of the rule. Revision algorithms often focus on modifying certainty factors, although some of these algorithms also correct logical errors in the rules.

Most of these learners use normal propositional representations, supplemented with some specially-handled first-order components. They include PROHC [Wilkins and Tan 1989], SEEK [Politakis 1985] and KRUST [Craw and Sleeman 1991]. Davis's [1979] TEIRESIAS, the learning component of the well known MYCIN expert system, is also included.

Mozetic's [1987] learning component of KARDIO is an active, revising learner of relational theories which represent the functional and structural properties of the human heart. It was designed to aid the construction of an expert system for the diagnosis of heart disorders.

We also discuss three EBL revising learners which use a propositional theory. This restricted representation simplifies some aspects of revision. Hamakawa's [1991] THEORY REFINEMENT ALGORITHM (TRA) is presented only in broad outline and ignores many of the difficult problems of induction encountered by its approach. On the other hand, Cain's [1991] DUCTOR and Mooney and Ourston's [1991] EITHER are well-developed systems.

There are also a number of relational revising learners of the EBL framework. OCCAM [Pazzani 1989] is a relational revising learner which is also concerned with correcting a "compiled" theory to reflect changes made in the underlying "domain" theory. We consider only its approach to revision of the domain theory. COAST [Rajamoney 1989] is a revising learner for theories represented using *Qualitative Process Theory* [Forbus 1984] with an EBL approach. The work focuses on identifying and using *exemplars* as case histories for choosing between alternative revision hypotheses.

4.4.2 Operator planning

A number of revising learners are concerned with incrementally revising theories of operators. The theories describe the preconditions and postconditions of operators that can be used to achieve planning goals. Although these systems do not use logic programs as their representation, their representations could be translated to normal logic programs.

Typically, these learners make EBL assumptions, that is, some finite set of concepts are considered to be defined by indisputable ground facts, and all other concepts are defined in terms of these.

Active revision of a relational knowledge of *operators* and *inference rules* is performed by the PRODIGY system [Carbonell and Gil 1987, Minton, Carbonell, Knoblock, Kuokka, Etzioni and Gil 1990, Carbonell and Gil 1990, Gil 1991].

The ENTROPY REDUCTION ENGINE (ERE) [Kedar, Bresina and Dent 1991] makes EBL assumptions and uses an additional declarative partial model of the domain, "domain

constraints" to revise a relational theory representation of operators.

TL [Hayes-Roth 1983] uses a heuristic approach to correct errors in its relational theory representation of operators and inference rules. Some of the revision techniques deal with correcting knowledge which is logically inconsistent: we ignore these techniques because we assume a consistent theory.

4.4.3 Science

The problem of theory revision has been addressed in work on the philosophy of science for many years. Popper [1969a] acknowledges the vital role of errors in the acquisition of knowledge, including scientific knowledge. His theory justifies a trial-and-error approach to science, and demands that science properly proceed by attempting to discover counter-examples to hypotheses. The source of hypotheses is not important, but of those which can explain observations, those which make improbable (with respect to other knowledge) and testable predictions should be preferred. From this work we derive the necessity to endow an incremental learner with the ability to ask questions (or to perform experiments).

Kuhn [1970] identified two forms of revision in scientific theories: minor corrections of "normal science" and major revolutions of "paradigm shift" periods. Shapiro [1981] believes that the revolutions require a change in the language of hypotheses, that is, a capability for *constructive induction*.

There are several learners which use this scientific scenario to investigate theory revision.

A relational revising learner called ABE [O'Rorke, Morris and Schulenburg 1989] is claimed by its authors to be capable of "world model revision" rather than "routine theory revision". However there is no mechanism for determining which form of revision is more appropriate for some input, nor is there any capability for constructive induction. Like COAST, the theory is expressed with relational Qualitative Process Theory, and the method is illustrated by modelling the Chemical Revolution of the 18th century.

KEKADA [Kulkarni and Simon 1989] is also an active learner which attempts to discover scientific hypotheses to explain observed chemical reactions. It attempts to model the behaviour of an experimental scientist by applying heuristic strategies derived from observation of human scientists. Many of the heuristics are strongly domain-dependent and, because the revision process is tightly integrated with other aspects of the system (such as generating goals and making decisions) it is difficult to compare the revision techniques with other revision systems. We note here that the strategies "divide and conquer" and "determine if all the independent entities are necessary to produce the surprising phenomena" are similar respectively to the missing answer diagnosis and delete antecedents generalization techniques of other revising learners.

4.5 Approaches to Revision Problems

In this section a range of approaches to the main aspects of learning by theory revision is presented. We discuss approaches to diagnosis, generalization and specialization which are implemented in the learners introduced in previous sections.

4.5.1 Diagnosis

What should an inductive learner do when it observes a failing example? It could discard the entire theory it has developed to date and recommence learning from scratch. It could assume that the example is an isolated anomaly and minimally correct its theory to account for it. Alternatively, it could analyse its theory to detect the reason for the fault and then attempt to learn from the mistake.

Many revising learners including KRUST, TRA, FORTE, DUCTOR, SEEK, KARDIO and COAST make little distinction between diagnosis and error correction, assigning blame to particular parts of the program according to the neatness with which they can be fixed. This neatness is typically measured by preference for improvements in accuracy over a large example set, preference for using certain revision operators, preference for the syntactic form of the revised theory or preference for accuracy over specially selected examples. Often these preferences are exhibited as strong assumptions about the nature of the error. For example, the assumption that there is only one atom missing that would enable a successful proof is made by OCCAM, AUDREY and PROHC.

Some batch learners, including MPL, EITHER, AUDREY, TRA and SEEK prefer to assume a clause as false if it is used in a large number of proofs of failing negative examples.

Other learners, including ERE and, in some circumstances, PRODIGY and CLINT, assume that an error is due to the definition of the concept of a failing example. This is a reasonable assumption for single concept learners and, in some cases, for learners of operators, but does not generally hold. It is an especially poor assumption when learning multiple concepts with the aid of background knowledge which has been learnt the same way.

Contradiction backtracing is used in a simple restricted form by PRODIGY, and also more generally by TEIRESIAS, MIS and CLINT. Missing answer diagnosis is used by TEIRESIAS and MIS.

4.5.2 Generalization

Some revising learners do not generalize. Having diagnosed a missing atom, the corresponding unit clause or a covering clause given by the oracle is simply added to the theory. This is the approach taken by TEIRESIAS, ABE, and KARDIO. The atom is not always known to be true so in a sense some inductive learning is achieved.

A more commonly used approach is to generate or select a starting clause, and then

to generalize that clause. The starting clause may be the example itself, a diagnosed missing atom, or some existing clause which is assumed to require generalization in order to cover the missing atom.

Many generalization operators apply only to non-unit clauses, in which case a starting clause must be non-unit. The diagnosed missing atom (or the unit example itself) forms the head of the clause and some *relevant* literals comprise the body. The problem of determining relevance for this purpose was noted by Plotkin [1971b] but left unanswered. It is called the *situation-identification problem* by Charniak and McDermott [1986].

4.5.2.1 Relevance

It is common practice (and necessary) for propositional learners such as DUCTOR, KRUST, OCCAM, TRA, TEIRESIAS, SEEK and EITHER to expect the relevant facts to be associated with an example in the input. That is, the burden of identifying relevance is placed on the teacher or environment. Unusually for relational learners, FORTE and COAST also work in this way. The relational incremental learner MARVIN collects the relevant facts in a dialogue between the learner and a teacher. For all these learners relevant facts are deemed to be those associated with the example in input. Sometimes the generalization technique also supports the consideration of any facts that can be derived from the given facts using the current theory as background knowledge.

PRODIGY partitions operator concepts into observational ones and derived ones. The relevant facts associated with an example for learning are all the observational facts known to be true in the environment at the time of the example. This technique assumes that relevance is determined by the environment. The learners of operator theories TL and ERE do similarly.

Other learners assume that relevant facts must be selected from those already true in the theory as background knowledge. One technique for doing this is based on least general generalization such as is used by GOLEM and another is the closely related *saturation* operator [Rouveirol 1991b]. Essentially these techniques suggest that every true atom of the theory is relevant. Some atoms may be discarded by making certain syntactic simplifications or assuming that two or more examples of a concept are to be covered by the same clause. The difficulties with the technique stem primarily from the typically large number of facts in the theory, and hence a very large constructed clause.

The technique used by CLINT is closely related. All facts of the growing theory are deemed relevant, but a predefined ordering of syntactic restrictions is used to prefer certain subsets of the facts over others. A larger subset is used only when it is shown that a smaller subset is not satisfactory.

Another technique used, for example, in FORTE, assumes that all the relevant facts are transitively *connected* [Plotkin 1971b, Vere 1977, Richards and Mooney 1992] to the example in a graph of atoms which are connected when they share terms. The length

of the connection path may also be limited. This assumption sacrifices completeness for a more restricted notion of relevance which is often sufficient.

4.5.2.2 Generalization operators

Many learners, including KRUST, TRA, FORTE, ERE, OCCAM, DUCTOR, SEEK, TL, AUDREY, EITHER and PROHC, use heuristic techniques to select one of an available set of generalization operators. These include adding a missing atom, least general generalization of the missing atom and a background clause or another missing atom, inverse resolution operators (identification and absorption), and deleting antecedents from a clause. These learners differ mainly in the range of operators they consider and the heuristic functions they use to select one. Some learners, including KARDIO, TRA, EITHER, and PROHC, incorporate well-known single-concept batch learning techniques to induce rules to cover a large number of missing atoms of the same concept. MPL uses a variant of FOIL's technique to add literals one at a time to the body of an increasingly specialized covering clause with an interesting twist: the predicate at the head of the clause is not determined until the body of the clause is complete.

MIS does not generalize: it works through an enumeration of the clauses in the predefined language, adding any which minimally specialize an earlier removed clause and which cover the missing atom.

CLINT successively deletes antecedents from a least general covering clause (of a restricted syntactic form) until a suitable hypothesis is found, or if necessary successively relaxes constraints on the language of the covering clause.

4.5.3 Specialization

The major difficulty with specialization is managing to avoid over-specialization and so uncovering some true facts.

Many learners use a simple approach to specialization: they always remove a clause believed to be false. These learners, including MIS, KARDIO, AUDREY, ABE, MPL, and CLINT, focus their efforts on generalization: any true atom uncovered by specialization is learnt again as input to generalization. This approach means that it is easy to achieve completeness by having a complete generalization technique, but it is wasteful of both the resources (and justifications) for originally constructing the clause which is removed and the utility of the clause for achieving other goals. It also required the learner to keep a record of all the examples it observes.

Other learners, including TEIRESIAS, EITHER, FORTE, TRA, DUCTOR, TL, SEEK, KRUST, ERE, COAST, and PRODIGY, use heuristic techniques to select one of a number of specialization techniques available to them. These techniques include removing blamed clauses and adding some antecedents to blamed clauses. The antecedents to add are chosen from the antecedents of examples, those derivable from the antecedents of examples, by asking the oracle, or by experimentation.

Another simple technique for specialization is used by KARDIO, depending on the mode of operation. Ground facts are collected as exceptions to clauses and a simplified SLDNF-refutation inferencing procedure is used. As pointed out by Ling [1991], any learner for which this is the only way of specializing a false clause cannot identify in the limit.

4.6 The AGM Logic of Belief Revision

A theory of rational changes of belief that we call *AGM logic* has been developed primarily by Alchourrón, Gärdenfors and Makinson [Gärdenfors 1988]. A substantial and growing body of literature deals with this model of theory revision which is closely related to the study of non-monotonic logics. It is concerned with providing a logical model for the theory revision which should rationally occur in response to a single observational sentence. In our model this corresponds to incremental learning. But, as noted by Gärdenfors the theory is not intended for modelling the belief changes of *inductive* learners.

In the AGM logic a *theory*, representing knowledge or beliefs, is a set of propositional statements closed under a consequence relation with certain properties. The theory describes an *epistemic attitude* to any sentence expressed in the language of observation. Any observational sentence is either *accepted* (is in the theory), *rejected* (its negation is accepted) or *indeterminate* (neither the sentence nor its negation is in the theory). In contrast, the concept learning approach typically supports only two epistemic attitudes: acceptance and rejection.

The Gärdenfors *revision postulates* prescribe constraints on the revision function by which a theory is updated to account for new observations. The postulates are motivated by a criterion of minimal change that is expressed as the *conservativity principle*:

> When changing beliefs in response to new evidence, you should continue to believe as many of the old beliefs as possible.

Although the revision postulates are concerned with three revision functions, they are reducible to a single function, *contraction* by which a sentence previously accepted in the theory becomes indeterminate. The contraction functions which minimise the change to a theory, according to the conservativity principle, are called *maxichoice* functions. Any contraction function of this class is distinguished by a preference partial ordering on theories based on the notion of *epistemic entrenchment*.

Maxichoice revisions are rejected because they generate revisions that are too big. It is shown that any theory $\mathcal{K}$ thus revised is maximal, that is, for all propositions A either $A \in \mathcal{K}$ or $\sim A \in \mathcal{K}$. This criticism is applied by Wrobel [1993] to the so-called non-monotonic learning recommendations of Bain and Muggleton [1990], underlying specialization by invention of exception predicates. However, the criticism is not relevant to the ILP model of knowledge because theories already have this property prior to revision due to the closed world assumption.

In our ILP model of learning a maxichoice revision function has a more serious drawback. In response to a new example contradicting the theory it would add a positive example as a unit clause or a negative example as an exception to a clause which covers it. Indeed, the conservativity principle conflicts with the aim of learners to construct theory representations which have more consequences than just the observed examples. It even prevents a learner from learning a correct theory when the target is infinite. However, learners sometimes assume that revising for a failing positive example should be inductive and prompt generalization but that revising for a failing negative example, that is specialization, should be minimal. For these systems, the AGM approach to revision for negative examples is appropriate and sometimes approximately used. We shall see that MINERVA also takes this approach when there is insufficient time to examine better alternatives.

The AGM model justifies the importance of the notion of *epistemic entrenchment*, which is based on the utility or informational power of a sentence in a theory. The entrenchment of a sentence (or clause) is not the same as the believed probability of the sentence's correctness, rather it assumes that some information is more important or more useful than other information. Gärdenfors identifies a major paradigm shift or a scientific revolution [Kuhn 1970] with a radical change on the ordering of epistemic entrenchment in a theory.

There is a correspondence between the epistemic entrenchment of sentences in a theory of the AGM model and heuristics determining a revision function on a corresponding theory representation as a program [Nayak, Pagnucco, Foo and Kwok 1995]. This justifies the use of heuristic measures of clause value in revising learners. It is interpreted in MINERVA as an evaluation of clausal hypotheses based on both cover and complexity.

5

The Learning Algorithm

5.1 Introduction

This chapter describes MINERVA, an algorithm designed to achieve autonomous learning by incremental induction and revision. The chapter commences with a strategic outline noting the assumptions made in the design of MINERVA, followed by a procedural description of the top level algorithm, providing a framework for the integration of the learning components. The first learning component, the revision procedure incorporating generalization and specialization is described but details about generalization are left to chapters 6 and 9. Four inter-dependent lower-level components of MINERVA— interpretation, diagnosis, experimentation, and redundancy detection — are discussed in detail, together with schematic code descriptions of their implementation.

5.2 Design Principles

5.2.1 Strategy outline

MINERVA is a learning algorithm designed for embedding in an information-processing centre of an intelligent agent. Learning goals are generated elsewhere in the agent and communicated to the learner by the simple mechanism of symbolic examples which prompt revision of the learner's program.

5.2.1.1 Questions

MINERVA's task is to learn as much as it can from any example presented to it. This involves an intensive analysis of the example coupled with question-asking for more information. Assuming that question-asking is a time-consuming process for the learner, it must be done in an intelligent way to minimise the number of questions and to get the most informative results from them. The questions asked must be reasonable for the environment to answer. The language of questions should therefore be the same as the language of examples. Questions phrased in language terms known only internally to the agent are not permissible.

5.2.1.2 Interruption

At some point the agent embedded in an environment must interrupt the learning process to attend to other needs. So that we do not need to incorporate goal generation into the learning model, it is assumed that the interruption is unpredictable and becomes apparent to the learner without prior warning. When interrupted the learner can only take the minimal necessary time to save state. Because of the possibility of interruption at all times during learning MINERVA must follow a strategy of taking action which seems most likely to provide the best improvement in the theory in the short term, while aiming to make the best improvement in the theory in the long term if sufficient time is made available. This requirement suggests a best-first search strategy for revisions, informed by a heuristic metric of goodness. The metric should allow for comparison of alternative revisions which may be at different stages of investigation and development.

5.2.1.3 State memory

When interrupted, the learning state is saved only by immediately applying the revision which is considered best at the time of interruption. No record is kept of the justification for the chosen revision nor of the alternative revisions which might have been made were more time available. Although this might seem to be an unfortunate waste of useful information, the approach is merited by its simplicity and space economy. Only the theory representation and a small number of additional facts are stored between presentation of examples. The approach is justified by assuming that a goal-generation component of the learning agent could periodically inspect the progress of the learner and allow it to continue working until satisfied with the revision proposed. Alternatively, if there is an externally-generated interruption in the mean-time, the present learning problem is assumed to be less important than other demands for the resources of the agent. If necessary, shortcomings of the best revision at the time may be corrected at a later time by presentation of another illustrative example to the learner.

5.2.1.4 Short-term fact memory

As well as the program representing the current theory, MINERVA maintains a short-term facts database. The facts database comprises a small set of true ground literals — classified true and false atoms. The facts are collected as input examples or questions and their answers.

MINERVA is a long-lived learner gradually learning a potentially infinite theory, so there may be a large or unbounded number of such facts made available to MINERVA. As MINERVA learns, its theory should approach the target, with the program representing the theory in a more compact and predictive form than any set of ground literals. If MINERVA is learning well, the facts should all be true in the theory.

The facts database is used solely to reduce the number of questions asked in diagnosis and experimentation and does not affect learning performance in other ways. Therefore a simple implementation of the facts database having a predefined fixed size and a first-in-first-out space allocation scheme is sufficient. Because in practice target domains often exhibit an uneven distribution of positive and negative examples in input, the finite memory space is partitioned into spaces each for positive and negative literals, each limited by an upper bound on the number of such literals stored. The bounds are parameters to MINERVA, adjusted according to preference. But learning performance is not sensitive to their values except to the extent that higher values usually reduce the number of questions asked by MINERVA as the facts database can more often supply the answer.

5.2.1.5 Optimism

During learning MINERVA assumes that the current theory is correct everywhere except where an error is demonstrated by facts. This assumption respects the original reasons for the assimilation of clauses into the program, without requiring access to those reasons. The assumption reduces the number of questions which might otherwise be asked in diagnosis and testing of revision hypotheses. For example, if an input example demonstrates an error in the theory, an alternative diagnostic approach could ask questions to verify every atom of the theory. Nevertheless, a compromise is made between the aim to avoid re-visiting existing parts of the program and admitting that everything could be wrong. This compromise aims to give weight to the value of existing knowledge and even the structure of the existing program while admitting the possibility of error in them. In practice, this means that an inductive hypothesis is evaluated by its cover with respect to the current program rather than the target, and that a clause is only removed from the program when every atom it covers is either refuted or covered in some other way.

5.2.1.6 Accuracy

There is no provision for error in observed examples or answers to questions. It is assumed that the environment is completely accurate and that truth does not vary over time. In practice, though, MINERVA's learning behaviour can tolerate inaccuracies and time-varying truth because of its revision strategy and short-term fact memory. If an example contradicts the facts database it is ignored. An example which does not correspond to any fact of the database is treated as a new fact and revision is prompted only if it contradicts the theory. If a question is answered inaccurately then that inaccurate answer is regarded as the truth for diagnosis and evaluation of hypotheses, but the revision mechanisms allow for odd exceptions to a generally useful rule and so revised hypotheses are not unduly affected by small numbers of errors.

5.2.2 Redundancy

Because of the learning strategy's focus on individual examples as they are presented and on best incremental changes to the program, the structure of the program representing the current theory can become unnecessarily complex. From time to time MINERVA takes a break from other tasks to work on internal representational changes to the program. This procedure, called sleep, is done quietly, without asking questions. The times at which the learner sleeps are dictated externally, as is the amount of time available for the sleep. Regular sleep is not mandatory for learning, but we would expect learning performance to improve when opportunities for sleep are available.

5.2.3 The top-level algorithm

Figure 5.1 describes the outermost flow of control in MINERVA in a procedural notation. MINERVA commences with an initial program P and an initial fact database F provided externally. These would usually be empty. In succeeding code figures, the current program P and fact database F are treated as global variables and are not declared in the code fragments. Although changes to P in the code are explicitly noted, changes to F are not. Actually, F is updated whenever an example is observed or a question answered.

Input to the learner is a sequence of commands. Most of these commands would be positive or negative examples from which MINERVA is to learn. Interleaved, there could also be other control commands. Most of these are for inspection of the current state of MINERVA and the program but one command is important to learning: the sleep command instructs MINERVA to sleep until interrupted.

Immediately an example is observed by MINERVA there is potential for surprising interruption. The first task of MINERVA is to check whether or not the example is provable with the current program. If it is, the example is simply recorded in the facts database and the learner awaits the next command.

```
input parameters
   P: program        /* usually {} */
   F: facts database /* usually {} */
var
   R: revision
   C: command

begin repeat forever
   C := read
   case C
      example:
         par-begin
            R := null
            while not (P ⊢ C) do
               P := assimilate(R,P)
               R := best-revision(C,P)
            endwhile
         []
            wait-for-interrupt
            break
         par-end
         P := assimilate(R,P)
      sleep:
         P := sleep(P)
      other:
         other-command(C,P,F)
   endcase
end repeat forever
```

N.B. The **par-begin**, [], **par-end** notation delineates sections of code executed in parallel. Every section terminates whenever any one section terminates.

Figure 5.1: Top level algorithm

If an example is not provable then MINERVA proceeds to search for suitable revisions to the theory. Often the search for a revision poses questions. The answers are recorded in the facts database for future reference and immediately used to direct the search. When the search terminates the chosen revision is assimilated into the theory. Sometimes the chosen revision, although fixing an error, is not sufficient to make the example provable, in which case another revision is sought. Otherwise, MINERVA just idly awaits the next command. Often the search is terminated prematurely by an interruption, in which case MINERVA immediately assimilates the best satisfactory revision of the time and awaits the next command.

5.3 Revision

The major work of MINERVA lies in the revision procedure, described in figure 5.2. The revision procedure firstly makes a diagnosis, which is either a false clause and a substitution giving a false atom covered by the clause, or a true atom not covered by any clause in the program. MINERVA then proceeds to determine a suitable revision. A **revision** is a pair composed of some clauses in the program marked for retraction and some new clauses for assertion.

When a false clause is diagnosed, heuristic criteria are used to decide whether to remove the clause and replace it by an equivalent set of unit clauses or, alternatively, to specialize the clause by adding a negative literal about an exception predicate to the clause and subsequently generalizing the definition of the exception predicate. When an uncovered atom of an exception predicate is diagnosed then the same heuristic criteria determine whether to remove the clause to which it is an exception and replace it by its true exclusive cover or, alternatively, to generalize the exception predicate definition. The heuristic criteria are detailed in chapter 6. On the other hand, if a diagnosed uncovered atom is about an observational predicate then it is always generalized.

5.3.1 Specialization by exception

A false atom may be excluded from the theory of MINERVA either by removing a covering false clause and adopting unit clauses to cover each of the true atoms covered by the clause, or by excepting the clause. In MINERVA negative literals are introduced to the body of a clause only to describe *exceptions* to clauses. The exceptions are discovered by diagnosis or by experimentation as described later in this chapter. Notice that exceptions are defined with respect to the current theory, not to the target.

Definition 5.1 (Exception) *Let C be a normal clause and P be a normal program. If C covers an atom A with respect to P although A is false, then A is an* exception *to C.*

Exceptions are excluded from the cover of a clause in MINERVA by including a single negative literal about an *exception predicate* in the body of the clause. The definition of the exception predicate is generalized to cover the atom corresponding to the exception. Negative antecedents are used only to exclude exceptions in MINERVA's programs: there are no other negative literals in clause bodies. Furthermore, atoms about exception predicates occur in the body of at most one clause of a program — in a negative literal.

The predicate symbol of an exception predicate is an *invented predicate*. It is invented internally and cannot occur in the language of observation. Any other predicate symbol in a program is called by contrast an *observational* predicate. The arguments of the negative literal occurrence of an exception predicate are identical to the arguments of the atom which forms the head of the clause in which it occurs. Furthermore clauses

```
procedure best-revision
input parameters
   Ex: fact /* not P ⊢ Ex */
   P : program
output parameters
   Revision: revision
var
   Diag: diagnosis
   C, C': clause
   θ: substitution
   A: atom
   H: set of clause
begin
   Revision := null
   Diag := diagnose(Ex)
   if Diag = "(Cθ) is a false clause" then
      Revision := decide-fault(Cθ⊙,C)
      if Revision = "make an exception" then
         A, C' := invent-exception-predicate(C,θ)
         H := best-generalization(A)
         Revision := "assert H ∪ {C'} and retract {C}"
      endif
   else-if Diag = "(A) is an uncovered atom" then
      if A is about an exception predicate then
         C:= clause ∈ P s.t. A excepts C
         Revision := decide-fault(A,C)
         if Revision = "make an exception" then
            H := best-generalization(A)
            Revision := "assert H"
         endif
      else
         /* A is an observational atom */
         H:= best-generalization(A)
         Revision := "assert H"
      endif
   endif
end
```

Figure 5.2: Procedure best-revision

defining exception predicates are themselves definite clauses, allowing no negative literals in their body. We shall see later that this very limited form of predicate invention offers advantages for incremental program revision for MINERVA, especially when time is short, but enables judicious diagnosis and generalization of invented predicates.

Definition 5.2 (Exception predicate, excepted clause) *Let C be a definite non-unit clause $p(t_1, \ldots t_m) \leftarrow C_1, \ldots, C_n$ where p is a predicate symbol of arity m in the language of observation and $t_1, \ldots t_m$ are terms. Then C is* excepted *by replacing it*

by an excepted clause C' *given by* $p(t_1, \ldots t_m) \leftarrow C_1, \ldots, C_n, \sim p^C(t_1, \ldots t_m)$*, where* p^C *of arity* m *is an* exception predicate *for* C'*, and* p^C *occurs in the body of no other clause and can never occur in the language of observation. Every clause in the definition of* p^C excepts C.

Example 5.1 (Exception predicate) Consider the program:

mother(kim, kate)←
married(bill, kim)←
father⁰(bill, kate) ←
father(X,Y)← married(X,Z), mother(Z,Y), ∼ father⁰(X,Y)

Then $father^0$ is an exception predicate and the clause $father^0(bill, kate) \leftarrow$ excepts the excepted clause $father(X,Y) \leftarrow married(X,Z), mother(Z,Y), \sim father^0(X,Y)$. □

Although an exception predicate occurs only once as a negative antecedent, there may be any number of clauses in which it occurs as the predicate of the head. Note that excepting an allowed clause results in another allowed clause.

5.3.2 Generalization

If time is available, having diagnosed and excepted a clause, or having diagnosed a missing observational atom, MINERVA carries on to search for a generalization of the atom. The generalization procedure employs the basic mechanisms of flattening (definition 3.6), absorption (definition 3.4) and restriction (definition 3.5). The particular form of absorption, called *f-absorption*, is developed, defined and analysed in chapter 9; for the moment we can describe it as most general absorption working in the flat representation. F-absorption is applied iteratively — the output of one application becomes the input clause for another — to eventually generate a set comprising almost every clause that is more general than the initial missing atom.

The procedure works like this. Beginning with an atom that is covered by no clause in the current program, it is flattened it to yield a non-unit clause and considered to be the first candidate hypothesis. Regarding a candidate hypothesis as an input clause, a background clause from the program is selected. Any negative literals are deleted from the clause and it is flattened. Then, for each suitable substitution, f-absorption is applied. A record of the full set of optional atoms is attached to the clause and the new clause is included in the set of candidate hypotheses. Iteratively a hypothesis is selected from the set and restricted using the optional literals (until exhausted) to create other hypotheses which are also included in the set. Moreover, subject to experimental confirmation, the selected hypothesis is paired with a suitable flattened background clause and f-absorption is applied again. As it is generated, each clause is compared with those already generated and discarded if duplicated.

Each clause in the hypothesis set, except those that fail a number of syntactical constraints is a candidate for unflattening and then assimilation into the program as an

inductive hypothesis to cover the initial atom. The syntactical constraints are identified in section 6.5.3.

While building up the hypothesis space a heuristic measure is used to determine the allocation of resources to the incremental generation of the hypothesis set. It determines which hypothesis in the set should be developed further in the next step and whether that should be by f-absorption, restriction, or experimentation. It also supports exception predicate invention to manage errors in the background program during generalization. Of the hypotheses in the set it determines which is eventually assimilated into the program. The use of the heuristic to achieve these goals in MINERVA is described in chapter 6.

5.4 Interpreting a Program

The theory of MINERVA is internally represented as a program and an *interpreter* is used to determine the corresponding theory. When MINERVA observes an example the interpreter acting on the program determines whether the example is already provable, or included in the theory. Interpretation of the program is also vital for error diagnosis, evaluation of a hypothesis by determining its effect on the theory, and recognition of *redundant* clauses which may be removed.

The natural and customary choice for a logic program interpreter is one of the SLD-refutation or SLDNF-refutation procedures. These are the efficient, well-understood interpreters for PROLOG. Most ILP learners use an SLD-refutation procedure with a computation rule which always selects the leftmost atom in a goal together with a depth-first search rule, trying clauses in the program in the fixed order in which they appear: the "standard" definite PROLOG interpreter. It is supplemented with the negation as failure rule to give an SLDNF-refutation procedure but, for most ILP learners goals are evaluated with respect to definite programs so a simplified SLDNF-refutation procedure is sufficient. MINERVA learns normal programs, requiring a more sophisticated interpreter.

For MINERVA we prefer to keep within the conventional class of normal programs and to make use of the efficient and well-studied interpreters based on SLDNF-resolution. We have shown in section 3.6.2 that there is no standard interpreter which is sound and complete, guarantees termination and does not overly restrict the language of the theory representation.

In compromise then, we opt for a pragmatic approach simultaneously incorporating many of the partial solutions of the literature. It is descriptively complex but simply implemented and demonstrates in practice many of the best advantages without suffering unduly from the disadvantages of other interpreters. It depends on assumptions about the structure of programs and goals to be interpreted: first we describe the language of programs of MINERVA.

5.4.1 The language of programs

MINERVA learns normal clauses with function symbols. To enable successful interpretation of programs made up of such clauses, the clauses are restricted in the following ways.

Allowedness

Clauses are *allowed* (definition 3.12). As we shall see, this has the consequence that all initial goals required for interpretation in MINERVA are also allowed. Each clause has at most one negative literal. Because the clauses are allowed and any negative literal is always rightmost in the clause, the standard leftmost computation rule is safe — there is no need to implement a delaying computation rule to prevent floundering.

Self-recursion

There is at most one *self-recursive* literal in each clause: an antecedent about the same predicate as the clause itself. Further, although it does not affect the expressive ability of the language, this literal is constrained to occur as the rightmost positive literal in a clause. This goes some way towards making the program cycle-restricted because the most obvious predicate on which the predicate of the head depends — itself — can only occur according to the cycle-restricted constraint (see section 3.6.2.1).

Furthermore a self-recursive antecedent in a clause may not introduce a new variable into the clause, that is, its variables must also occur in other antecedents or the head of the clause. This restriction makes the clause partially *nvi* (definition 3.15), with regard to the particularly problematic self-recursive literals. Without this constraint, MINERVA tends to favour clauses which violate it and non-termination impedes successful learning in practice. Note however, that in general clauses may be non-*nvi*.

Term embedding bound

The language of MINERVA also imposes a constraint on the occurrence of function symbols in a clause: the term embedding (definition 3.17) of every variable in the head of the clause exceeds or equals the maximum term embedding of the same variable in the positive antecedents of the clause. This term embedding bound aims to address the problem of divergent inference (see section 3.6.2.2) that frequently arises in practice in MINERVA in the absence of the bound. Although this restriction is inspired by the observations of Smith et al. [1986] and De Raedt [1992] presented in section 3.6.2.2, it is a quite different solution to that which they suggest. It is similar to the restriction of De Raedt [1992] but it permits non-*nvi* clauses and does not depend on a forward-chaining interpreter.

This term embedding bound does not guarantee termination of an interpreter in the way

the alternatives can. But it does prevent a troublesome clause such as $n((X) \leftarrow n(s(X))$ from occurring in the program while permitting a clause such as $n(s(X)) \leftarrow n(X)$.

Negative literals

If a negative literal occurs in a clause then it is about an exception predicate, occurring in the body of no other clause. The arguments of the negative literal are identical to the arguments of the atom in the head of the clause, as this is sufficient to achieve its intended purpose to exclude exceptions. The clauses defining the predicate symbol of the literal are themselves definite clauses. Furthermore these clauses do not include any antecedent about the corresponding observational predicate symbol — such clauses are rarely useful inductive hypotheses.

These restrictions on the occurrence and definition of exception predicates are justified by the effect on diagnosis discussed later in section 5.5, but they also serve to alleviate the special difficulties for interpretation of recursive complementary literals introduced in example 3.9. They do not prevent those difficulties, but in practice mean that they occur less often than otherwise.

5.4.2 Interpreters

MINERVA employs two interpreters to interpret programs of this structure, both based on the standard SLDNF-refutation procedure. The *cover interpreter* is used to determine the cover of a hypothesis for evaluation and the *membership interpreter* is used for all other purposes.

In most respects the interpreters are alike. Both interpreters implement the standard, sound unification algorithm including the occurs check. Both interpreters use the standard computation rule, always selecting the leftmost literal in a goal, but this is a safe computation rule for the programs of MINERVA. The search rule of both interpreters uses the standard ordering rule, trying program clauses in the fixed order in which they occur in the program. Although this is actually a temporal ordering, it may be considered arbitrary for interpretation purposes. The interpreters differ by the search order of SLDNF-trees.

During the computation, whenever a ground atom is selected, at most one refutation for that goal is explored. When any refutation is found, search of the sub-tree rooted at the goal in which the atom was selected is terminated. This reduces redundancy in the search, and so sometimes avoids searching infinite branches of SLDNF-trees. This idea was proposed by Smith et al. [1986] and is shown to be sound by them.

The interpreters of MINERVA also employ the SIR_M loop check (definition 3.14), which is the strongest sound loop check of the many analysed by Bol et al. [1991]. The loop check is applied to SLD-derivations — those parts of an SLDNF-tree not involving selection of negative literals. It is not complete for the programs of MINERVA, even when there are only definite clauses. However, it can avoid many non-terminating and

redundant computations that would otherwise occur.

Because the loop check is not complete, to improve termination properties a fixed depth bound limiting the length of derivations is also enforced. The depth bound check incorporates Stickel's [1984] refinement mentioned in section 3.6.2.1: the search of a derivation is failed at a step when the depth bound is exceeded by the sum of the length of the derivation to that step and the number of literals in the goal at the step.

Because of the SIR_M loop check this fixed bound can be greater than when a depth bound is the only precaution against non-termination. The depth bound here is used as a catch-all for infinite or unbounded SLD-derivations not pruned by SIR_M. The depth bound is defined by a parameter to MINERVA.

But even the depth bound does not guarantee termination of the interpreter in the presence of negative literals in clause bodies of the program. Whenever a negative literal is selected, its computation, or search for an SLDNF-refutation, is bounded by the same depth bound on the length of derivations as for the initial goal. If, during that search, a negative literal is selected then the same initial depth bound is imposed again. Therefore the depth bound does not always bound the search and it remains possible that the computation does not terminate. This will occur in cases of complementary literal goals like in example 3.9, although those particular programs violate MINERVA's constraints on the occurrence of negative antecedents.

The two interpreters differ in respect of the search rule used. The membership interpreter is used to determine whether a given ground literal is included in the theory or to find answers to non-ground goals in diagnosis and redundancy checking. It uses the standard depth-first search rule, supplemented by the loop check mechanisms described. In schematic code fragments, use of the membership interpreter for program P and goal $\leftarrow G$ to determine an answer θ to $\{P \cup \leftarrow G\}$ is indicated by the notation $P \vdash G\theta$.

The *cover interpreter* is used to determine cover of an inductive hypothesis with respect to the current program. In this case answers to non-ground goals are sought and the determination of answers in easiest-first, subsets is appropriate for the context in which hypotheses are evaluated. Therefore the cover interpreter makes use of an iterative deepening search strategy, where the depth increment is defined by a parameter to MINERVA. Whenever a negative literal is selected by the cover interpreter it is in turn interpreted by the membership interpreter, thus avoiding boundary inconsistencies.

5.5 Diagnosis

A presentation of earlier work and a general description of the contradiction backtracing and missing answer diagnosis algorithms were given in section 3.7. In this section we describe how diagnosis is performed in MINERVA, firstly by a general description coupled with schematic code figures, and then by a more detailed description of particular design points, identifying where further improvements could be made.

```
procedure diagnose
input parameters
   L: literal
output parameters
   D: diagnosis
   Found: boolean
var
   A: atom
   Proof: proof tree
   C: clause
   θ, φ: ground substitution
begin
   Found := false
   if L is negative, ~ A then
      /* A is ground */
      if not question(A) and there is a Proof for P ⊢ A then
         D:= false-clause(Proof)
         Found := true
      endif
   else /* L is an atom, usually ground */
      A:= L
      while there is another φ s.t. Aφ is ground and question(Aφ)
               and not P ⊢ Aφ and not Found do
         while there is another clause C ∈ P s.t. there is a θ
               s.t. Aφ = Cθ⊙ and not Found do
            Found, D := diagnose-conjunction(Cθ⊗)
         endwhile
         if not Found then
            D := "(Aφ) is an uncovered atom"
            Found := true
         endif
      endwhile
   endif
   if not Found  then
      D:= "there is no error"
end
```

Figure 5.3: Missing answer diagnosis

Figures 5.3, 5.4 and 5.5 describe MINERVA's declarative diagnosis. Procedures **diagnose** and **diagnose-conjunction** implement missing answer diagnosis while procedure **false-clause** implements contradiction backtracing. Contrary to the usual presentation they are described in a procedural notation because this better illustrates the search order — the major point of variation in alternative forms of the algorithms. The reader who prefers a PROLOG-like declarative treatment is referred to the work cited in section 3.7.

In figure 5.3, diagnose is the top level procedure for declarative diagnosis, initially invoked with a parameter which is an example — a true ground literal. Whenever the

```
procedure diagnose-conjunction
input parameters
   S: sequence of literals S1...Slength(S)
output parameters
   D: diagnosis
   Found: boolean
var
   i, j: index of clause literals
   θ: substitution
begin
   Found := false
   if empty(S) then
      j:= 0
   else
      j:= least i in 1 ... length(S) s.t. there is no answer θ to P ⊢ (S1,...,Si) θ
   endif
   while j > 0 and not Found do
      while there is another answer θ to P ⊢ (S1,..., Sj-1) θ and not Found do
         Found, D := diagnose(Sjθ)
      endwhile
      j := j - 1
   endwhile
end
```

Figure 5.4: Missing answer diagnosis (continued)

literal is inconsistent with the theory the procedure sets the return variable Found to true and also returns either an uncovered atom or a false clause. When the literal is consistent with the theory Found is set to false and no error is diagnosed.

When the parameter is a negative literal for which the complement is both false and provable, the procedure false-clause is invoked to diagnose the error. Otherwise, when the parameter is a positive literal which is both true and not provable, then the body of each clause with a head which unifies with the literal is passed in turn to procedure diagnose-conjunction. The matching clauses are tried in the order they appear in the program. If diagnose-conjunction fails to find an error in any matching clause then the true and not provable parameter is the uncovered atom.

In diagnose-conjunction (figure 5.4) each literal of the conjunction is examined in turn; an instance of it passed recursively to diagnose. Procedure diagnose performs better when its parameter is ground, so the first literal examined is the rightmost one for which there is an answer for the conjunction of literals to its left. Each answer provides a possible but not provable instance of the failing literal: each instance is the subject of an invocation of procedure diagnose. If diagnose does not find an error in a literal of the conjunction, then the literal to its left is examined in the same way. If diagnose finds an error in one of the literals then diagnose-conjunction returns successfully, setting

```
procedure false-clause
input parameters
   Proof: proof tree
output parameters
   D: diagnosis
var
   Found: boolean
   Ci: literal
   A: atom
begin
   /* root(Proof) is a false atom provable by Proof tree */
   Found := false
   while there is another child, Ci, of root(Proof) and not Found do
      if Ci is a negative literal, ~ A then
         Found, D := diagnose(A)
      else /* Ci is an atom */
         if not question(Ci) then
            D := false-clause(Ci)
            Found:= true
         endif
      endif
   endwhile
   if not Found then
      D := "(Root ← {Ci | Ci is a child of Root}) is a false clause"
end
```

Figure 5.5: Contradiction backtracing diagnosis

Found to true. If it finds no error in any literal of the conjunction then it returns unsuccessfully, setting Found to false.

Procedure false-clause in figure 5.5 is invoked with a parameter which is a proof tree (definition 3.19) of an atom known to be false. The literals which are children of the atom in the proof are examined in a left to right order until either an error is found in one of them or there are no more. A negative literal is examined by a recursive invocation of diagnose on its complement, an atom. A positive literal is examined by a membership query, and, if it is false then the proof sub-tree rooted there is recursively passed to false-clause. If all children have been checked without diagnosing an error then the root of the proof parameter is a false atom covered by the instance of the false clause with the root as the head and its children comprising the body.

Declarative diagnosis plays an important role in the learning of MINERVA, but the performance and question-efficiency of the diagnosis are not critical factors to the performance of MINERVA as a whole. Some performance improvement that could be gained in some circumstances is therefore given up in favour of the advantages of clarity and simplicity.

Having given a skeletal description of diagnosis in MINERVA, in the remainder of this section the design decisions are justified and some additional details are fleshed out. The first-time reader might prefer to skip to section 5.6.

5.5.1 Questions

Frequently the procedures must verify some ground atoms. Procedure question firstly refers to the facts database but if the answer cannot be found there a membership query is asked. The atom and the answer are immediately stored in the facts database for future reference. If procedure diagnose is invoked with a non-ground atom parameter then the facts database is used by question to find a true ground instance of the atom if possible. But question can only access facts stored there — it cannot make queries to find other valid ground instances of the non-ground atom.

This is the major point of deviation of the missing answer diagnosis of MINERVA from Naish's [1992] N2. In N2 the ϕ of procedure diagnose would be found by asking an instance query, prohibited in MINERVA. MINERVA does not ask the question but instead treats it as having been answered negatively. This circumstance arises when there is no answer for the non-ground atom about which the question would have been asked, or when all answers have been tried but did not lead to a diagnosis. By assuming a negative answer, an uncovered atom of the atom's predicate or a predicate on which it depends may not be discovered. Instead, unless an error is diagnosed in a literal to the left of it, the procedures will determine that the corresponding instance of the corresponding clause head is an uncovered atom. This diagnosis could be in error: such an atom is a missing atom but not necessarily an uncovered one — it may indeed be covered by the clause. Fortunately in this case, the diagnosed atom is both true and missing from the theory, so the addition to the program of a covering clause would improve the theory. In the worst case, avoiding non-ground questions in this way amounts to avoiding missing answer diagnosis altogether.

In contradiction backtracing of MINERVA there is no need to ask for a false ground instance of a non-ground atom nor to arbitrarily apply ground substitutions to literals in the proof of a false atom as suggested by Shapiro [1982]. This is because the restriction to *allowed* clauses in the programs of MINERVA guarantees that every literal in a proof tree is ground.

5.5.2 Search order

Although the top-down version of contradiction backtracing is employed partly due to its suitability for incorporation of heuristics (see section 3.7), in MINERVA's false-clause questions about the atoms in the body of a rule are made only in a left to right order.

Missing answer diagnosis investigates clauses that unify with a missing atom in the order they appear in the program. In procedure diagnose-conjunction as in N2, the membership interpreter is used to find answers satisfying as many as possible of the

literals counting from the left. This often has the effect of finding a substitution θ which grounds the variables in S_j enabling better performance of diagnose.

But missing answer diagnosis in diagnose improves N2 by asking fewer questions in some circumstances. In N2 the atom A is verified by a question before determining whether A is provable. When A is ground and not provable the question is unnecessary, as the answer does not affect the subsequent behaviour of the diagnoser. Therefore in MINERVA, whenever A is ground in procedure diagnose the provable check is made first and the membership query is made only if it is indeed provable.

5.5.3 Invented predicates

When the diagnosis procedures require the verification of an atom of an invented predicate, a membership query cannot be used because the predicate does not even occur in the language of the target. Recall that an invented predicate describes an exception to a clause, occurring uniquely in the body of the clause as a negative, rightmost literal. The invented predicate symbol is applied to the same arguments as the corresponding observational predicate symbol at the head of the clause. Further, particularly to aid diagnosis, clauses defining an invented predicate do not themselves include exception predicates in the body.

Instead of asking about ground atoms of exception predicates MINERVA assumes that such an atom is true whenever the corresponding observational predicate instance is false and vice-versa. This assumption causes the diagnosis procedures to investigate the predicates on which a predicate depends before diagnosing an error in the exception predicate itself, so an error in an exception predicate is diagnosed only if no errors are found in the predicates on which it depends. When the corresponding observational atom is true but not provable only because the exception atom is provable, and when there are no contributing errors in the predicates of the definition of the exception atom, then the clause which covers the exception atom is concluded to be a false clause. On the other hand, when the corresponding observational atom is false and provable (and hence the exception atom is not provable), then if there is no contributing missing atom in the predicates defining the exception predicate, the exception atom is concluded to be an uncovered atom.

Thus a clause which is excepted will never be determined to be false by the diagnoser; an instance of the exception atom will be diagnosed missing instead. A clause defining an exception atom will be determined to be false by the diagnoser when that clause prevents a clause defining an observational predicate from covering a missing atom.

5.5.4 Non-terminating diagnosis

Because declarative diagnosis of a program is so closely tied to interpretation of the program, the diagnosis procedures themselves are subject to non-termination difficulties. In MINERVA's diagnosis procedures, as for the interpreters, the matter is dealt

with by the implementation of several measures.

Contradiction backtracing, in procedure false-clause, commences after a refutation has been found, so because it works through the corresponding finite proof tree it terminates provided that it does not recursively invoke diagnose.

The missing answer diagnoser is itself an interpreter of the program as can be seen by its reference to the program to find a unifying clause in procedure diagnose. Its top-down search for an uncovered atom follows the structure of a search for an SLDNF-refutation so recursion in the program can cause non-termination of the diagnoser.

In this case, however, an easy loop check can be used. On the initial call to diagnose and subsequent calls from false-clause, a loop check stack parameter is initialized to be empty. In diagnose a ground instance of the input parameter atom is found immediately before matching the atom with program clauses, preparatory to recursive invocation of procedure diagnose-conjunction. At this point the ground atom is compared with the ground atoms in the loop check stack representing antecedent invocations of the procedure. If the atom is not repeated there it is pushed onto the stack and the procedure continues as described in figure 5.5, the stack parameter being passed through to successive invocations of diagnose through diagnose-conjunction. On the other hand, if the atom is repeated, then the recursion is terminated at this point; if there is no alternative ground instance then Found is set to false and the current invocation of diagnose terminates.

In practice, this implements the strongest loop check of Bol et al. [1991]: the instance of an atom loop check. In this case it is applied to ground atoms selected in the search, and in this circumstance it coincides with other loop checks proven sound by them, so it is also sound.

Because of the term embedding bound (section 5.4.1) and the use of the facts database to instantiate non-ground atoms in diagnose, the term depth of atoms in the loop check stack is bound above by the maximum of the term depths of the first atom on the stack, the atoms of the facts database, and ground terms in the body of program clauses. There can only be a finite number of such atoms, so the loop check ensures that, provided the interpreter terminates and false-clause is not recursively invoked, diagnose always terminates. But mutually recursive calls of diagnose and false-clause can cause non-termination of the diagnoser. This problem, too, could be avoided by a more complex loop check mechanism which detects complementary literals, but this is not implemented because the interpreter is also non-terminating in such circumstances (see section 5.4.2).

5.5.5 Interrupted diagnosis

Diagnosis is sometimes time-consuming because of its frequent use of the interpreter. If learning is interrupted during diagnosis, MINERVA conveniently assumes that the missing atom or clause partially diagnosed at the time of interruption is to blame.

For missing answer diagnosis the blamed atom is the (positive) example initially, but later the most recent atom on which the diagnosis procedure was recursively invoked. Certainly such an atom is both true and not provable, so its addition to the program as a unit clause improves the theory. However, were missing answer diagnosis allowed to proceed further a better missing atom could be found — one whose addition to the program would put both itself and the earlier atoms into the theory.

If contradiction backtracing in procedure false-clause is interrupted before completion, the top level clause of the proof tree of the most recent invocation is assumed to be faulty. It may not be false but the atom at the root of the proof is nevertheless covered by it in the program.

Unless the clause defines an exception predicate, it is now *excepted* so that it no longer covers the exception. The excepted clause and the ground unit clause defining the exception predicate are added to the program, replacing the blamed clause. But if the clause already defined an exception predicate, then instead the observational atom corresponding to the exception atom is added to the program as a unit clause.

These revisions are made without generalization and so comprise a "quick fix" to the theory. If the learner is never given sufficient time to proceed beyond the diagnosis stage, the program tends to become a set of ground unit clauses representing a finite theory.

5.6 Experiments

Having constructed a plausible inductive hypothesis by generalization, an active learner like MINERVA should be able to evaluate the hypothesis by performing experiments. Ideally, the syntax of the questions should conform with those asked in diagnosis to avoid over-burdening the teacher. In MINERVA we find that such experimentation is possible without equipping MINERVA with any additional capabilities: hypothesis testing requires only an interpreter and membership queries.

The critical experiments to perform in considering acceptance of a hypothesis are those which verify the new facts which that hypothesis introduces into the theory. To avoid a surfeit of questions, attention is not given to the atoms which are already in the theory, covered by other clauses, even if they are (unknowingly) false. The exclusive cover of a clause with respect to a program comprises the atoms which are covered by the clause but are not covered by any other clause of the program. In the style of the earlier notion of cover, we firstly define exclusive cover in an interpretation, and then with respect to a program representing a theory.

Definition 5.3 (Exclusive cover) *A normal clause C* exclusively covers *ground atom A in an interpretation I if there is a substitution θ such that $C_{\odot}\theta = A$ and $\exists(C_{\otimes}\theta)$ is true in I and A is false in I.*

Definition 5.4 (Exclusive cover with respect to a program) *Let H be a normal clause, and P be a normal program. Then H* exclusively covers *ground atom A with respect to P if H exclusively covers A in every Herbrand model for comp(P).*

The exclusive cover of a hypothesis H in the interpretation which is the current theory, represented by normal program P, can be evaluated by a logic programming system by computing the answers for $P \cup \{\leftarrow H_{\otimes} \wedge \sim H_{\odot}\}$ and applying each answer to $H_{\odot}$. The soundness of this method, called the *exclusive cover test*, is established in the following.

Lemma 5.1 (Safeness and groundness of the exclusive cover test) *Let H be an allowed clause and P be an allowed program. Then every computation of $P \cup \{\leftarrow H_{\otimes} \wedge \sim H_{\odot}\}$, is safe and every computed answer is a ground substitution for the variables in H.*

Proof. If H is allowed then every variable in H occurs in a positive literal of $H_{\otimes}$ (definition 3.12), so the goal $\leftarrow H_{\otimes} \wedge \sim H_{\odot}$ is also allowed. The result follows from the safeness and groundness of allowed computations (theorem 3.4). □

Theorem 5.1 (Soundness of the exclusive cover test) *Let H be an allowed clause and P be an allowed program. Then for every computed answer θ to the safe computation of $P \cup \{\leftarrow H_{\otimes} \wedge \sim H_{\odot}\}$, $H_{\odot}\theta$ is ground and H exclusively covers $H_{\odot}\theta$ with respect to P.*

Proof. Let I be a Herbrand model for *comp*(P). Let θ be a computed answer for $P \cup \{\leftarrow H_{\otimes} \wedge \sim H_{\odot}\}$. Then by soundness of SLDNF-resolution (theorem 3.3) θ is a correct answer. By definition (3.10) of a correct answer $\forall(\{H_{\otimes} \wedge \sim H_{\odot}\}\theta)$ is a consequence of *comp*(P). By lemma 5.1 θ is ground so $\{H_{\otimes} \wedge \sim H_{\odot}\}\theta$ is a consequence of *comp*(P). Rearranging, $H_{\otimes}$ is a consequence of *comp*(P) and $\sim H_{\odot}$ is a consequence of *comp*(P). Therefore $H_{\otimes}\theta$ is true in I and $H_{\odot}\theta$ is false in I. Therefore $H_{\odot}\theta$ is ground and H exclusively covers $H_{\odot}\theta$ in I (definition 5.3). The result follows from the definition (5.4) of exclusive cover with respect to a program. □

The exclusive cover of a hypothesis can then be evaluated by membership queries, by the question procedure also used for diagnosis. Notice that the exclusive cover of an arbitrary clause is not necessarily ground, but it is for an allowed clause and allowed program. Conveniently, allowedness also guarantees that the computation in the evaluation of exclusive cover is safe.

A large number of true atoms in the exclusive cover suggests that a hypothesis is "useful" because its addition to the program would enable it to represent many new facts. A large number of false atoms suggests that a hypothesis is bad because its addition to the program would add a large number of false facts to the theory. But the existence of false atoms in the exclusive cover does not necessarily mean that a

clause is false, because the interpretation used for evaluation of the exclusive cover is the possibly incorrect theory; it is not the target. Similarly, even when all the atoms in the set are true, the hypothesis may be false. The exclusive cover test says something about how plausible and how useful a hypothesis is; it does not guarantee correctness.

Related hypothesis testing techniques, based on interpreting a goal constructed from the hypothesis, are used in MARVIN, CIGOL and CLINT. These systems test only definite hypotheses with respect to definite programs and their soundness is not established.

5.6.1 Experiments about self-recursive predicates

Unfortunately the exclusive cover test may under-rate the usefulness of some recursive clauses, in the same way that generalized subsumption fails to capture the power of recursive clauses in generalization.

Example 5.2 (Exclusive cover) Let P be the program $\{p(a)\leftarrow\}$ and let H be the clause $p(f(x))\leftarrow p(x)$. Then the exclusive cover of H with respect to P is the set $\{p(f(a))\}$ although $comp(P \cup \{H\}) \models \{p(a),\ p(f(a)),\ p(f(f(a))),\ \ldots\}$ □

This deficiency of the theoretical generality model has a practical consequence in MINERVA, unfairly biasing against recursive predicate definitions. But a simple trick is implemented in MINERVA to work around the problem. When a clause being tested is self-recursive, that is, has an atom in the body which unifies with the atom in the head, its exclusive cover is determined to be the union of the exclusive cover (definition 5.4) and the *second order* exclusive cover, defined as follows. There is no ambiguity in the definition because clauses in MINERVA have at most one self-recursive antecedent and atoms of the exclusive cover are ground.

Definition 5.5 (Second order exclusive cover) *Let P be a normal program. Let H be the normal clause $H_{\odot}\leftarrow B_1,\ldots,B_{m-1},B_m,B_{m+1},\ldots,B_n$ and let A be an atom such that H exclusively covers A with respect to P and such that there is a substitution θ with $B_m\theta = A$. Then for any atom A', H* second order exclusively covers *A' with respect to P if $(H_{\odot}\leftarrow B_1,\ldots,B_{m-1},B_{m+1},\ldots,B_n)\theta$ exclusively covers A' with respect to P.*

In the second order test answers to the exclusive cover test are applied back to the the clause and the self-recursive antecedent is deleted. The exclusive cover of that modified clause is then evaluated. So now in example 5.2, the second order exclusive cover is $\{p(f(f(a)))\}$ and the determined union is $\{p(f(a)),\ p(f(f(a)))\}$, slightly improving the apparent power of the clause.

Although the trick will not ameliorate the bias against a clause which is mutually self recursive with another clause in the program, it is often sufficient in practice to enable the generation and adoption of appropriate self-recursive clauses. A more comprehensive solution to the problem would take account of the research on generalizing by implication described earlier in section 3.5.2.3.

5.6.2 Experiments about invented predicates

Experiments testing hypothesised clauses about invented predicates are handled a little differently in MINERVA. Firstly, membership queries cannot be used to verify the cover because the atoms are about invented predicates that do not occur in the target. Secondly, because of the special context in which they occur in the program, the validity or otherwise of many atoms covered by such a clause is irrelevant.

Recall that clauses defining an invented predicate define exceptions to another clause in the program — the invented predicate occurs only in the unique negative literal of the body of only one clause in a program. Therefore, the only atoms that are relevant in assessing a clause defining an exception predicate are those which are prevented from being covered by the excepted clause *because* they are covered by the exception clause.

Example 5.3 (Exception clause cover) Consider the normal program P, describing part of the family of figure 2.1.

$$\begin{array}{l} mother(kim, kate) \leftarrow \\ mother(kim, kris) \leftarrow \\ married(bill, kim) \leftarrow \\ stepfather(bill, kate) \leftarrow \\ stepfather(bill, kris) \leftarrow \\ stepfather(tom, betty) \leftarrow \\ father^0(bill, kate) \leftarrow \\ father(X,Y) \leftarrow married(X,Z), mother(Z,Y), \sim father^0(X,Y) \end{array}$$

Then the hypothesis $father^0(U,V) \leftarrow stepfather(U,V)$ exclusively covers the atoms $father^0(bill, kris)$ and $father^0(tom, betty)$. But $father(tom,betty)$ is not covered by $father(X,Y) \leftarrow married(X,Z), mother(Z,Y), \sim father^0(X,Y)$ so its validity is irrelevant to the evaluation of the hypothesis.

□

To detect only relevant exclusive cover, the exclusive cover for an invented predicate is defined as follows. In contrast to the exclusive cover for an observational predicate, we define it only by the computation which evaluates it.

Definition 5.6 (Exception exclusive cover) *Let E be a normal allowed clause $E_\odot \leftarrow E_1, \ldots E_n$ about an exception predicate. Let P be a normal program of allowed clauses. If there is a unique allowed normal clause $H \in P$, $H_\odot \leftarrow H_1, \ldots, H_m, \sim A$ such that A and $E_\odot$ have the same invented predicate symbol, then let μ be the mgu of $E_\odot$ and A (so $E_\odot\mu = A\mu$). For every computed answer θ to the safe computation of $P \cup \{\leftarrow(E_\otimes\mu \wedge H_\otimes\mu)\}$, E* exclusively covers *$E_\odot\mu\theta$ with respect to P.*

Like the exclusive cover test for observational predicates, the negated head of the clause occurs in the computation goal. This ensures that the cover evaluated is indeed

exclusive — not also covered by clauses in the program. It is easy to see that $E_{\odot}\mu\theta$ is ground, by applying much the same argument as the proof of lemma 5.1. Note that if there is no clause in the program using the exception predicate, then its exclusive cover is the empty set.

Example 5.4 (Exception exclusive cover) Refer to the program of example 5.3. The exception exclusive cover of the hypothesis $father^{0}(U,V) \leftarrow stepfather(U,V)$ is evaluated by $P \cup \{\leftarrow stepfather(U,V),\ married(U,Z),\ mother(Z,V),\ {\sim}father^{0}(U,V)\}$ giving $\{father^{0}(bill,\ kris)\}$. □

Atoms about invented predicates cannot be verified by a membership query because the predicate does not occur in the target. Recalling that such atoms represent exceptions to the corresponding observational predicate, the observational predicate may be asked about instead. When procedure question is invoked for an invented predicate, the atom is replaced by an atom about the corresponding observational predicate with the same arguments. This atom is verified against the fact database or, if necessary, by a membership query. Just as when question is used in diagnosis, the invented predicate atom is assumed true whenever the corresponding observational predicate instance is false and vice-versa. By this reasoning, we can now unambiguously describe ground atoms of exception predicates as true or false.

Example 5.5 (Invented predicate membership query) In the previous example (5.4), if $father(bill,\ kris)$ is false then $father^{0}(bill,\ kris)$ is assumed true and is the only atom of the true cover of the hypothesis. □

5.7 Redundancy

Picture the active, diagnosing learner gradually developing a program. False clauses in the program will eventually be located by contradiction backtracing, and presumably, repaired. True clauses that are adopted at some time will be superseded by later hypotheses, becoming redundant. Unless they can be recognized, such redundant clauses can clutter the program and impede reasoning. But MINERVA can recognize them by evaluation of their exclusive cover. First, let us define redundancy in the usual way, but adapted for normal clauses and programs. Then we describe a test for redundancy based on exclusive cover.

Definition 5.7 (Redundancy) *Let P be a normal program and C be a normal clause. C is* redundant *in P if $comp(P) \models \forall C$.* [Buntine 1988, page 165]

Theorem 5.2 (Testing redundancy) *Let P be a normal program and C a normal clause. If the safe computation of $P \cup \{\leftarrow C_{\otimes} \wedge {\sim} C_{\odot}\}$ finitely fails (that is, has a finitely-failed SLDNF-tree) then C is redundant in P.*

Proof. By soundness of NAF (theorem 3.2) if the computation finitely fails then $\leftarrow C_{\otimes} \wedge \sim C_{\odot}$ is a logical consequence of *comp*(P). Rearranging, $\forall(C_{\odot} \leftarrow C_{\otimes})$ is a logical consequence of *comp*(P). By definition 5.7 C is redundant in P. □

Note that if P and C are allowed, as in MINERVA, then every computation of the redundancy test $P \cup \{\leftarrow C_{\otimes} \wedge \sim C_{\odot}\}$ is safe, although it might not terminate.

Buntine [1988] suggests a different redundancy test for definite clauses with respect to definite programs. It is more complicated, requiring skolemisation of the clause and a computation for each clause in the program with a head which unifies with its head. It is incorporated in the ILP learners CIGOL [Muggleton and Buntine 1988] and CLINT [De Raedt and Bruynooghe 1992].

The redundancy test of theorem 5.2 may be viewed in terms of the exclusive cover test of theorem 5.1 and the exception exclusive cover test of definition 5.6: a clause is redundant if its exclusive cover is the empty set. This view conveniently defines a clause about an exception predicate to be redundant when the exception predicate is not used in another clause, or when it exclusively excludes no atoms from the cover of the clause it excepts.

5.7.1 Sleep

The exclusive cover redundancy test is incorporated in MINERVA, where it is used in the procedure sleep. MINERVA sleeps whenever the opportunity is given by the environment. The environment also determines the duration of sleep by interrupting to terminate it. When interrupted, MINERVA idly awaits another command or learning example.

During the sleep interval, each program clause in turn is evaluated. The clauses are tried in the fixed order they occur in the program, but at the termination of each sleep session the clause being evaluated at the time of interruption is marked. The next sleep commences trying the marked clause and continues onward through the program, cycling back to the first clause of the program after trying the last clause. But if sleeping is terminated while still evaluating the same clause with which the session commenced then the succeeding clause is marked instead, ensuring that each clause is tried eventually.

MINERVA does not only remove redundant clauses in this process. Heuristic criteria, detailed in the next chapter, determine whether the clause together with any relevant exception clauses in the program offers a simpler structure than an alternative comprising a set of ground unit clauses — one for each atom of exclusive cover. To determine this, a clause is temporarily removed from the program as it is examined. Its exclusive cover with respect to the remaining program is evaluated using the membership interpreter, returning one atom of cover at a time. A decision to return the clause to the program is made after determining only a subset of the exclusive cover, sufficient to justify the decision by the heuristic criteria. Otherwise the full exclusive cover is

evaluated and the clause is not returned to the program. Instead, the exception clauses are also removed and a unit clause for each atom is adopted.

There is one other circumstance that may cause a clause to be removed and replaced by its exclusive cover during sleeping. Recall that non-*nvi* clauses are particularly problematic for non-terminating computations because they introduce derivations which cannot be pruned by the SIR_M loop check. During sleeping, the existence of such clauses in the program can cause non-termination of the exclusive cover redundancy test, even when the clause being evaluated is itself *nvi*. Such clauses can also cause difficulty for the interpreters at other stages of learning, even though they may have been benign in the context of the program as it was at the time of adoption.

Therefore, to ease the interpreter difficulties, the sleeping procedure will also remove any *nvi* clause for which the evaluation procedure consumed a complete sleeping session from the start to the point of interruption. The clause is replaced by the exclusive cover evaluated. But this procedure has its cost, too. If sleeping periods are short, say less than the greatest period of time permitted for processing each example, there is a risk of removing benign and useful clauses at this time.

5.8 Summary

This chapter has commenced a description of the implementation of MINERVA. The top level algorithm and revision strategies have been described; incorporating generalization by clause addition and specialization by clause removal or generalization of exceptions. The implementation of strategic features of MINERVA have been introduced: the use of questions for diagnosis and experimentation; the short term memory for facts; and the interruptibility of the learning process. Some key components of MINERVA have also been described in depth: interpretation of programs; diagnosis for error detection; experimentation for hypothesis evaluation; and program simplification by removal of redundant or overly complex clauses. The language structure of programs developed by learning in MINERVA, being a restricted language of normal programs, has been defined and justified by its effect on interpretation and diagnosis procedures.

A complete description of MINERVA requires only two further, closely coupled, components. The succeeding chapter defines a heuristic measure of hypothesis preference and shows how it is used to choose the specializing operator and also to guide the search through the generalization hierarchy defined in chapter 9.

6

Heuristic Search for a Revision

6.1 Introduction

We have said that when a false clause is diagnosed, MINERVA uses heuristic criteria to decide between two options. One option is to remove the clause and replace it by unit clauses defining its true exclusive cover. The other is to add an exception literal to the clause and then to generalize the definition of the exception predicate. Likewise when an uncovered atom of an exception predicate is diagnosed, the same heuristic criteria determine whether to remove the clause to which it is an exception or to generalize the exception predicate instead. The same criteria is also used during sleeping to determine whether an overly complex clause should be removed and replaced in the program by unit clauses defining its exclusive cover.

After diagnosing an uncovered atom, the unit clause comprising the atom itself would be a suitable inductive hypothesis — but choosing this alone would limit theories to deductive consequences of observations. Development of a predictive or infinite theory would be impossible. So if time permits, MINERVA proceeds to search for clauses more general than the unit clause comprising the uncovered atom. In the course of the search MINERVA may discover facts about its world hitherto unknown.

This chapter describes the heuristic-guided decision procedure for choosing a revision to correct a diagnosed fault. It also describes the subsequent search through the generalization hierarchy of candidate inductive hypotheses when a decision to generalize is made. This search is guided by heuristic criteria compatible with that used for the initial decision — designed to minimise complexity of the revised program.

6.2 Heuristic Evaluation of a Revision

The value of a revision is based on a measure of the net decrease in complexity of the program resulting from the application of the revision. This is measured by the complexity of the clauses participating in the revision. The complexity of a clause is the length of its flat representation; see definition 3.6.

Definition 6.1 (Complexity of clauses) *Let C be a normal clause $C_{\odot} \leftarrow A_1, \ldots, A_p, \sim B_1, \ldots, \sim B_n$. Let D be a copy of C except that negative literals in $C_{\otimes}$ are replaced by their complement in $D_{\otimes}$. That is, $D = C_{\odot} \leftarrow A_1, \ldots, A_p, B_1, \ldots, B_n$. Let l be the number of literals in $flat(D)$ (including the atom of the head). Then the* complexity *of C, $complexity(C) = l + n$.*

The complexity *of a set of clauses is the sum of the complexity of each clause in the set. Let A be an atom. The* complexity *of A is $complexity(A \leftarrow)$. The* complexity *of a set of atoms is the sum of the complexity of each atom in the set.*

Because of the restricted occurrence of negative antecedents in MINERVA's program clauses, each negative antecedent contributes exactly two points to the complexity of the clause. Each distinct constant contributes one point and each distinct term contributes one point for its function symbol plus any points due to the terms comprising its arguments. Each atom contributes one point for its predicate symbol plus any points due to its argument terms. Variable occurrences make no contribution to complexity.

Example 6.1 (Complexity) Let C be $p(x,y,a) \leftarrow q(x,b), s(y,z), \sim p^C(x,y,a)$. Then C is transformed to $p(x,y,v) \leftarrow q(x,w), s(y,z), p^C(x,y,v), a_0(v), b_0(w)$. The number of literals in this clause is 6 and the number of negative literals in C is 1, so $complexity(C)$ is 7. □

The complexity measure is merited by its simplicity of evaluation and its uniform treatment of constant, function and predicate symbols of a clause. At the end of this chapter when it becomes clearer how this heuristic is used in MINERVA, we show how it relates to the *minimum description length* principle, often used in machine learning.

Definition 6.2 (Value of a Revision) *Let $\mathcal{R}$ be a revision to program P comprising the set $\mathcal{A}$ of clauses for assertion in P and the set $\mathcal{D}$ of clauses in P marked for retraction. Then the* value *of $\mathcal{R}$ is given by $complexity(\mathcal{D}) - complexity(\mathcal{A})$.*

Let us now see how this heuristic is used in MINERVA: for simplifying the program when sleeping, for choosing a specialization, and for guiding the search for a generalization.

6.3 Value Guided Simplification

Section 5.7.1 described the process by which MINERVA attempts to simplify a program by applying simplifying revisions in procedure sleep. For this purpose "simpler" means less complex according to the complexity measure.

A clause together with any clauses defining exceptions to it may be retracted from the program and replaced by a unit clause for each atom of its exclusive cover. This is done whenever the value of the revision is strictly positive. That is, such a revision is applied if the complexity of the clause and the clauses defining exceptions to it exceeds the complexity of the set of atoms comprising its exclusive cover.

Example 6.2 (Simplifying revision) Let P be the program

father(X,Y)← *married(X,Z), mother(Z,Y),* ∼ *father*^{0}*(X,Y)*
father^0*(andrew, mia)*←
married(andrew, alice)←
mother(alice, mia)←
mother(alice, andrea)←
mother(alice, michael)←

The exclusive cover of the first clause is {*mother(alice, andrea), mother(alice, michael)*} and the complexity of the cover is 6. The complexity of the clause is 5 and the complexity of the second clause which defines an exception to the first clause is 3. Therefore the value of the revision which removes the first and second clause and asserts the two clauses *mother(alice, andrea)* ← and *mother(alice, michael)*← instead is $(5+3)-6=2$. That revision would be applied by MINERVA given the opportunity in sleep. □

6.4 Value Guided Specialization

In response to a new observation the diagnoser of MINERVA may determine a false clause or an uncovered atom in the current program. When an uncovered atom is about an observational predicate, MINERVA proceeds to generalize the unit clause, aiming to discover additional facts.

But sometimes a false clause about an observational predicate is diagnosed. If so, then the clause is not already excepted and MINERVA must choose between two options in order to specialize the predicate definition: remove the clause, or except the clause. Alternatively, an uncovered atom about an exception predicate is diagnosed. Again MINERVA must choose between two options: remove the clause which it excepts, or generalize the exception predicate.

Excepting a clause is a conservative and speedy revision — the diagnosed problem itself may be corrected quickly without affecting the other atoms covered by the clause. MINERVA could go on to generalize the exception predicate and thus discover and

exclude other false atoms covered by the clause. The final combination of the clause and its exception clauses sometimes provides a simpler program than is possible using only definite clauses.

On the other hand, if the blamed clause is removed then every atom exclusively covered by that clause in the program is also removed from the theory. This might be a useful revision — it might remove many false atoms from the theory; and there could be a simpler way to cover the true atoms covered by the clause, using language which was not available at the time of its adoption. Indeed many of the true atoms covered by the clause might also be covered by later, better clauses and the clause might exclusively cover only few true atoms. Especially if the clause already includes an exception literal, the program might be simplified by removing the clause and replacing it by unit clauses for true atoms it exclusively covers. But when the clause covers a large number of true atoms finding an alternative covering clause could be difficult; and it could be more complex. How can MINERVA choose between the options?

6.4.1 Choosing a specializing revision

MINERVA supports both specialization strategies but chooses between them *before* commencing generalization of an exception predicate. It might be better to delay the decision until a good generalization of the exception predicate has been found, so that the choice may be fully informed. But, because the generalization search space may be large, this would mean that often there is insufficient time to properly investigate the removal option. Although some other exceptions to the clause might be discovered during the search for a generalization, that search is directed by the structure of the program and many other false covered atoms could remain unnoticed when an interruption forces a decision to be taken.

Figure 5.2 described the revision procedure of MINERVA. In that procedure the choice of specializing revision was left to procedure **decide-fault** which is described in figure 6.1.

In response to an exception, procedure **decide-fault** first evaluates the complexity of the clause and its exception clauses, assuming that the new exception will be handled as a new unit exception clause. If it does not already include an exception literal, this means accounting for the additional complexity that would be introduced by excepting it. The procedure then verifies atoms of the exclusive cover of the clause in the program minus the clause, found by the exclusive covers test using the membership interpreter. Each atom is ground, so procedure **question** can verify the atom against the facts database or by a membership query.

The complexity of each true atom of cover is counted in favour of retaining the clause in the program and excepting it as necessary; the complexity of each false atom of cover is counted in favour of retracting the clause. Having thus considered the cover of the clause, if it would be simpler to remove the clause and assert unit clauses as necessary for the true cover, then this is the determined revision. On the other hand, if there is an interruption during the procedure, or if it is simpler to except the clause and assert

```
procedure decide-fault
input parameters
   A: false atom
   C: clause
output parameters
   Revision: revision
var
   Cover, Faulty: set of atoms
   P': program
   Complex: integer
   Next: atom
   E: set of clauses
begin
   if excepted(C) then
      E := set of clauses that except C
      Complex := complexity(C) + complexity(E) + complexity(A)
   else
      Complex := complexity(C) + exception-penalty + complexity(A)
   endif
   P' := P − C
   Faulty := {A}
   Cover := {}
   while there is another atom Next such that C exclusively covers Next in P'
         and Next ∉ (Faulty ∪ Cover) and Complex > 0 do
      if question(Next) then
         Complex := Complex − complexity(Next)
         Cover := Cover ∪ {Next}
      else
         Complex := Complex + complexity(Next)
         Faulty := Faulty ∪ {Next}
      endif
   endwhile
   if Complex > 0 then
      Revision := "retract ({C} ∪ E) and assert Cover"
   else
      Revision := "make an exception"
   endif
end
```

Figure 6.1: Choosing a specializing revision

unit exception clauses for each exception found then a decision is made to except the clause.

When the clause has a large cover or the interpreter is slow to return answers, the decision procedure itself can be very time-consuming. So instead of evaluating the exclusive cover of the clause entirely before proceeding to search for a generalization of the exceptions (see figure 5.2), a compromise approach is taken. As each atom of

the cover is verified, its contribution towards the proposed revision is counted. If at some time the balance favours excepting the clause, that is Complex ≤ 0, because a relatively high complexity of true covered atoms has been verified, then the procedure terminates prematurely with a decision to perform the excepting revision, with time remaining to search for a generalization for the exception.

6.4.2 Assimilating a specializing revision

If interrupted during procedure decide-fault which decides whether to except a clause or to remove it, the decision to except is taken immediately. This means applying a revision which excepts the clause unless it is already excepted, inventing a new exception predicate and replacing the original clause in the program by the excepted one. Ground unit clauses are asserted which define the exception predicate for the diagnosed error as well as any others discovered during the decision procedure.

If decide-fault terminates with a decision to remove a clause then it, together with any clauses defining its exception predicate, is removed immediately. Unit clauses are also asserted for each of the true exclusively covered atoms, and MINERVA idly awaits further input.

Alternatively decide-fault terminates with a decision to except a clause. The exception atom is generalized in procedure best-generalization; any other exception atoms discovered in decide-fault becoming the initial *discovery* set for the procedure. The purpose of the discovery set will be made clear as we describe the generalization procedure.

6.5 The Search for a Generalization

In chapter 9 a generalization hierarchy will be defined — structured by an absorption operator called *f-absorption* coupled with restriction and rooted at a flat input clause. To find a covering hypothesis for a diagnosed uncovered atom MINERVA searches this space, rooted at the flat clause form of the atom. The root clause could define either an observational predicate or an exception predicate. In this section the search space is filled in to give a finer-grained structure — a directed graph of *partial hypotheses*. The partial hypotheses represent stages in the development and evaluation of the clauses more general than the clause at the root. Some nodes in the graph are clauses of the generalization hierarchy, being potential inductive hypotheses, but a direct link between clauses in the generalization hierarchy is replaced by an indirect path between these nodes in the partial hypothesis graph. The intermediate nodes are partial hypotheses representing stages of development between the two clauses. Along the path, the clauses are generated, tested by experiment and evaluated for their contribution to the developing theory. The exception operator is sometimes used to expand the search space beyond the generalization hierarchy to include hypotheses which cannot be expressed in the observational alphabet.

The graph of partial hypotheses is represented in MINERVA as an ordered list of partial

hypotheses — each being an open node in the present state of the search through the graph. A partial hypothesis comprises state information associated with the developmental stage. It may include an explicit clause being the inductive hypothesis of the generalization hierarchy under development. It implicitly represents a number of other inductive hypotheses that may become explicit, separate entries in the list when the partial hypothesis is sufficiently developed. It also includes a record of the exclusive cover of the inductive hypothesis, as far as it has been evaluated.

The heuristic *value* of a partial hypothesis determines its position in the list of partial hypotheses competing for the resources for further development. These resources are both computational ones (space and, more importantly, time) and external ones: asking questions. No attempt is made to determine the relative values of these resources: the best hypothesis is given the resources it requires to progress it to a point that it becomes either complete or poorer than the best. The value of a partial hypothesis estimates the value of the revision resulting from assimilation of the inductive hypothesis it represents. It also estimates the value of the partial hypotheses beyond it in the graph. It indicates the potential contribution of a hypothesis to the theory in terms of cover and complexity.

Because the generalization process of MINERVA is subject to unpredictable interruption, investigation of the most promising generalizations in a best-first manner enables MINERVA to select an inductive hypothesis which is the "best" possible in the time available. Later we expand on how this estimate is made for each partial hypothesis.

6.5.1 Partial hypotheses

Figure 6.2 describes the search for a generalization in MINERVA in procedure best-generalization. In the figure, the variable Open represents the partial hypothesis list of open nodes. Each partial hypothesis has a type that indicates its current stage of development. The type is one of: linear, paired, restrict, testing, or complete. The types represent discrete steps of development at which it is convenient to assess the value of a partial hypothesis and to reconsider its further development.

Initially the partial hypothesis list Open contains one partial hypothesis. It is of linear type and implicitly represents every generalization of the unit clause at the root of the generalization hierarchy. The exclusive cover of the clause is determined without experimentation; it comprises only the diagnosed atom.

The output variable Bestcomplete represents a partial hypothesis of type complete. Initially it is the partial hypothesis representing the initial unit clause, the only complete hypothesis at this stage.

During the following steps, MINERVA searches the generalization hierarchy rooted at the initial unit clause. Only some generalizations are generated in each step. For the others, partial hypotheses representing them are generated and the value of these, compared with the value of other partial hypotheses in the list, determines if and when those generalizations are explicitly generated.

```
procedure best-generalization
input parameters
   A: diagnosed missing atom
output parameters
   Bestcomplete: partial hypothesis
var
   Open, Next, Complete: set of partial hypothesis
   Best, H: partial hypothesis
begin
   Open := { "input: A←; cover: {A}; type: linear" }
   Bestcomplete := "hypothesis: A←; cover: {A}; type: complete"
   while not Open = empty do
      Best := best-of(Open)
      Open := Open – {Best}
      Next := develop(Best)
      Complete := {H ∈ Next | H is of type complete}
      Next := Next – Complete
      Bestcomplete := best-of(Complete ∪ {Bestcomplete})
      Open := Open ∪ Next
   endwhile
end
```

Figure 6.2: Processing partial hypotheses

At each step of the search the partial hypothesis of highest value is selected from the list and processed. That processing produces zero or more different partial hypotheses. Those that are not complete are inserted back into the list ready for the next step of the search. The best partial hypothesis of the newly complete ones and the previous best complete one is retained separately as Bestcomplete and any other complete ones are discarded.

Whenever learning is interrupted during best-generalization, of the *adequate* partial hypotheses, the one of highest value is immediately assimilated into the theory. Later we will be more specific about what kind of hypothesis is adequate.

6.5.2 Development of a partial hypothesis

A partial hypothesis commences its progression through the development process as linear. If it reaches the final stage of development it becomes complete. Along the way, it may take on any of the intermediary types and its development may give rise to several additional partial hypotheses of various types.

The development of a partial hypothesis through the types is illustrated in figure 6.3. The processing at each stage of development is described in figure 6.4.

A linear partial hypothesis is developed to a paired partial hypothesis of level 0 by

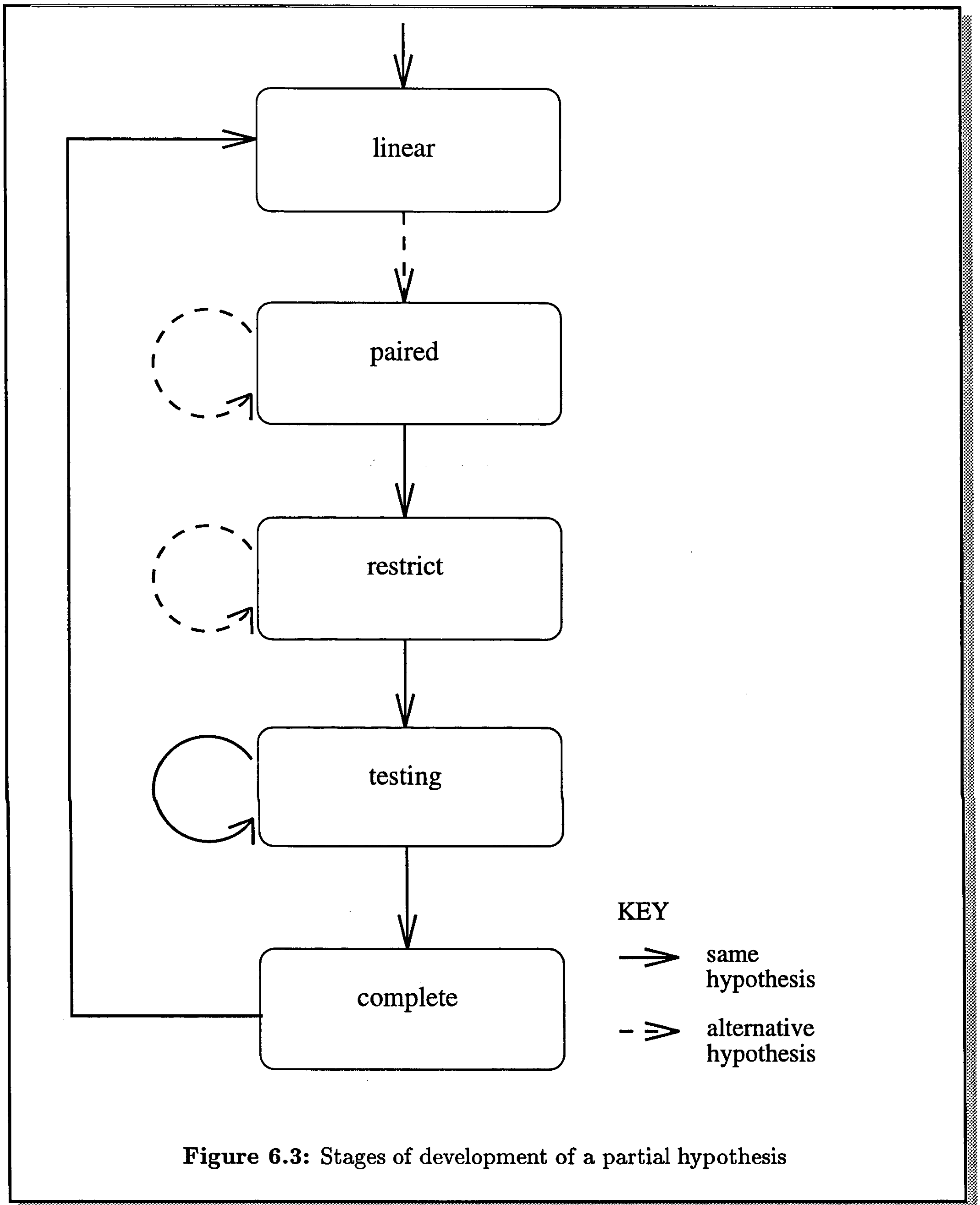


Figure 6.3: Stages of development of a partial hypothesis

flattening it and pairing it with a background clause in the program. The background clause is the "best" in the program, amended by deleting any negative antecedent and flattening. The **paired** partial hypothesis represents the potential f-absorption of the flattened **linear** hypothesis as the input clause with the flat background clause.

The development of a **paired** partial hypothesis by absorption is described in detail in figure 6.5. The hypothesis is developed to some **restrict** partial hypotheses provided

```
procedure develop
input parameters
   H: partial hypothesis
output parameters
   Next: set of partial hypothesis
var
   Hyp, B: clause
   A: atom
begin
   Next:= {}
   case type of H
      linear:
         B:= flatten(best-background(P))
         Next:= {"input: H.input; background: B; cover: H.cover; level: 0; type: paired"}
      paired:
         Next:= paired(H) /* see figure 6.5 */
      restrict:
         Hyp:= unflatten(H.absorb)
         if prunecheck(Hyp) then
            Next:= {"hypothesis: Hyp; cover: H.cover; faulty: {};
                        query: {}; depth: 0; type: testing"}
         endif
         for each A ∈ H.optional do
            Hyp:= H.absorb ∪ {A}
            Next:= Next ∪ {"absorb: Hyp; optional: H.optional − {A};
                              cover: H.cover; type: restrict"}
         endfor
      testing:
         Next:= experiment(H) /* see figure 6.6 */
      complete:
         /* impossible */
   endcase
end
```

Figure 6.4: Processing partial hypotheses

there is a suitable substitution for f-absorption of the input clause and the paired background clause. A restrict partial hypothesis is generated for each suitable substitution. The restrict hypothesis includes both a clause (an inductive hypothesis) generated by f-absorption of the input clause with the background clause using the substitution and a set of optional atoms.

In addition, the next paired hypothesis is generated by pairing the flattened input clause of the linear hypothesis with the next best background clause, with negative literals removed from the body and flattened. Thus each clause in the program is eventually considered as a background clause for absorption with the input clause at level zero. When every clause of the program has been tried with an input clause at

```
procedure paired
input parameters
   H: partial hypothesis
output parameters
   Next: set of partial hypothesis
var
   Hyp, B: clause
   Opt: set of atom
begin
   /* generate next paired hypothesis */
   if H.background = least-best-background(P) then
      if H.level < maxlevel then
         B:= flatten(best-background(P))
         Next:= {"input: H.input; background: B; cover: H.cover; level: H.level + 1;
                        type: paired"}
      endif
   else
      B:= flatten(next-best-background(H.background,P))
      Next:= {"input: H.input; background: B; cover: H.cover; level: H.level;
                     type: paired"}
   endif
   /* now absorb input clause with background clause if possible */
   for each Hyp, Opt ∈ f-absorb(H.level, H.input, H.background) do
      Next:= Next ∪ {"absorb: Hyp; optional: Opt; cover: H.cover; type: restrict"}
   endfor
end
```

Figure 6.5: Applying absorption

some level, the **paired** partial hypothesis is progressed to a **paired** type at the successive level.

Thus each background clause is considered for absorption at ever increasing levels. But the level is bounded above by the maximum number of constants and function symbols occurring in any clause of the program — since no new hypotheses can be generated at a greater level.

Now let us follow the path of a **paired** partial hypothesis which was developed to a **restrict** one in figure 6.4. A **restrict** partial hypothesis usually gives rise to some **restrict** partial hypotheses of lower value and a **testing** hypothesis. The transition to a **testing** hypothesis is made by unflattening the clause generated by f-absorption and subjecting it to a pruning check. If it fails the check, the corresponding **testing** hypothesis is not generated and consequently the clause never becomes the input clause for a later application of f-absorption, thus pruning the generalization hierarchy rooted there from the search space. In any case a new **restrict** hypothesis is generated for each optional atom of the source **restrict** hypothesis by restriction (see definition 3.5).

When the optional atoms of a restrict hypothesis are exhausted, no further restrict hypothesis is generated. Thus all the non-pruned clauses generated by f-absorption and restriction that are represented by a paired hypothesis can be represented as a testing hypothesis.

Once a generalization has been made, it must be tested before becoming suitable for adoption. Testing of a testing partial hypothesis proceeds incrementally: after each step the partial hypothesis is returned to the list, updated according to the results of the step. Each step either determines some additional cover of the hypothesis or asks a question about an atom in the hypothesis' cover. This is described in figure 6.6 and in more detail in section 6.5.4 following.

When fully tested, a testing hypothesis becomes a complete one, comprising a candidate hypothesis marked for adoption. If testing discovered no counter-examples to the inductive hypothesis, it is also recorded as a new linear partial hypothesis, thus preparing for another generation of generalizations generated by f-absorption. If exceptions were discovered during testing then the corresponding linear hypothesis is not generated, thus pruning from the search space the generalization hierarchy rooted at that clause.

Any complete partial hypotheses generated in the search procedure are not returned to the list of open nodes. Instead, they are compared to the current best complete hypothesis — and only the best of these is kept (see figure 6.2) There is no advantage in recording every complete hypothesis as at most one, the best one, is eventually adopted.

If the search for a better hypothesis is interrupted, then the best adequate one is immediately assimilated into the program. Sometimes, the best one is only the initial unit clause.

6.5.3 Hypothesis syntax checking

Sometimes the search space of generalizations by absorption is pruned by analysing the syntactic structure of a clausal hypothesis and rejecting the clause both as an inductive hypothesis and as an input clause for the generation of more general hypotheses by absorption. Figure 6.4 places the checking procedure prunecheck in context.

The check rejects a clause for failing to be allowed or for being a duplicate of a previously investigated clause: these pruning points are identified in chapter 9. The pruning check also enforces the constraints on the occurrence of self-recursive literals and excepted observational predicate literals as defined in section 5.4.1. If necessary, a clause which includes one suitable self-recursive antecedent is re-ordered to place that literal rightmost in the body. Clauses violating these constraints are rejected, along with every clause beyond them in the absorption hierarchy, as they too inherit the difficulties such clauses create for interpreters.

A clause is rejected also if it is too big: if the number of literals exceeds a fixed predetermined bound. In practice, this bound is very rarely exceeded because the

search heuristic favours shorter clauses, but it is necessary to ensure that finite memory resources are respected even though it compromises the completeness of the search.

In addition to the pruning checks, clauses are subject to three non-pruning syntax checks at this time; these checks are not indicated in the code fragment of figure 6.4. Clauses failing these checks skip the testing and complete stage of development and proceed directly to becoming a linear partial hypothesis ready for further generalization. Thus these clauses are rejected as candidate inductive hypotheses. A clause fails a non-pruning syntax check if it is the product of least general absorption (the associated set of optional literals is empty); if it is not connex (see definition 9.8); or if it violates the term embedding bound. The first two checks are justified in chapter 9 and the third in section 5.4.1.

6.5.4 Hypothesis testing

Each inductive hypothesis is tested to determine its validity and contribution to the theory in case it is chosen for assimilation. Figure 6.6 describes how this is done in procedure experiment.

In addition to the inductive hypothesis in question, a testing partial hypothesis comprises a depth parameter and three sets of atoms: query, cover and faulty. The first of these is initially empty but whenever it is empty when the testing hypothesis is selected for development, some more cover is evaluated. Procedure cover-deeper uses the iterative-deepening cover interpreter to determine another subset of the exclusive cover of the hypothesis, as described in section 5.6. The depth acts as a place marker for recommencing the search for cover at a later step, being set to finished when the limiting depth bound is reached.

In programs where there are large numbers of atoms returned by the cover interpreter, especially in domains with function symbols, the limiting depth bound may be dynamically reduced from the preset maximum bound. This happens when the cover of a clause being tested exceeds the cover of the input clause from which it was generated by a preset threshold amount *and* the heuristic value of the clause is the highest yet in the current generalization round (invocation of best-generalization in figure 6.2). Then the current testing depth is made the new limiting bound for subsequent hypothesis testing in the round. This achieves a measure of protection against over-stretched memory resources due to a very successful generalization — yielding a large increase in cover from a single generalization step. By avoiding deeper testing the need to use memory for more cover is avoided. To be considered better than the hypothesis in question, subsequent hypotheses must achieve a better cover in equal or less depth.

Having computed another subset of cover, the new subset is partioned into query: those atoms for which the validity is not recorded in the facts database, cover: those atoms which are known to be true in the facts database and faulty: those atoms similarly known to be false.

When selected at a later time, with a non-empty query set, the validity of a single

```
procedure experiment
input parameters
   H: partial hypothesis of type testing
output parameters
   Next: set of partial hypothesis
var
   Depth: positive integer
   Newcover: set of atom
begin
   if H.query = {} then
      if H.depth = "finished" then
         Next := {"hypothesis: H.hypothesis; cover: H.cover;
                  faulty: H.faulty; type: complete"}
         if H.faulty = {} then
            Next := Next ∪ {"hypothesis: H.hypothesis; cover: H.cover; type: linear"}
         endif
      else
         /* find some more cover */
         Depth, Newcover := cover-deeper(H.hypothesis,H.depth)
         Next := {"hypothesis: H.hypothesis; cover: H.cover; faulty:H.faulty;
                  query: Newcover; depth: Depth; type: testing"}
      endif
   else /* test another atom of cover */
      A := some atom ∈ H.query
      if question(A) then
         Next := {"hypothesis: H.hypothesis; cover: H.cover ∪ {A}; faulty: H.faulty;
                  query: H.query − {A}; depth: H.depth; type: testing"}
      else
         Next := {"hypothesis: H.hypothesis; cover: H.cover; faulty: H.faulty ∪ {A};
                  query: H.query − {A}; depth: H.depth; type: testing"}
      endif
   endif
end
```

Figure 6.6: Hypothesis testing

arbitrary atom of query is established or refuted by a membership query, and the atom is accordingly moved to cover or faulty respectively.

When the cover interpreter reports that it has returned all the cover and it has all been verified, the testing partial hypothesis is transformed to a complete hypothesis comprising the inductive hypothesis, the confirmed true exclusive cover and the confirmed false exclusive cover.

Further, if no cover was confirmed false, the testing hypothesis also spawns a linear partial hypothesis, ready for further generalization. The cover is included in the linear hypothesis and becomes the starting point for evaluating the successive partial hypotheses, since every generalization of the clause has at least the same cover.

6.5.4.1 Discovered facts

During hypothesis testing, whenever a question is answered affirmatively, another true atom, uncovered by any clause in the program, is discovered. This atom is not already in MINERVA's theory because it belongs to the exclusive cover of the inductive hypothesis being tested. It may happen that the hypothesis being tested is never adopted because a better one is found — and yet the better hypothesis may not cover those discovered atoms.

Accordingly, during the search of the partial hypothesis graph, discovered atoms are recorded separately from the partial hypothesis list as a *discovery* set. When eventually a partial hypothesis is assimilated into the program, the discovered facts which are not known to be covered by the chosen hypothesis are also assimilated — simply as additional unit clauses. Each atom of the discovery set is thus included in the revised theory and the discovery set is discarded.

6.6 Evaluating a Partial Hypothesis

During generalization, the partial hypothesis list comprises a number of partial hypotheses, each at a discrete stage of development. At each iteration of the generalization procedure, the most promising hypothesis from the stack is selected for development to its next step. The heuristic metric called *value* determines which hypothesis is the most promising, aiming to estimate its value as a revision to the program (definition 6.2). It incorporates both the cover of the hypothesis, favouring hypotheses that exclusively cover many true atoms, and the complexity of the hypothesis, favouring shorter hypotheses.

Whenever the generalization procedure terminates, the current best hypothesis which is also *adequate* is selected and assimilated into the theory. The same value measure is used to determine which is best.

6.6.1 Value of a partial hypothesis

In general, the value of a hypothesis is an optimistic estimate for the decrease in complexity of the program due to assimilation of the hypothesis, taking into account the facts discovered during the development of partial hypotheses. A hypothesis of higher value would include the diagnosed atom and the other true discovered atoms in the theory while increasing the program complexity to a lesser extent. Although a simpler clause has a higher value than a more complex one of the same cover, the size and complexity of the atoms of the true cover of a clause is a more significant factor in determining its value.

Informal Definition 6.3 (Value) *The* value *of a partial hypothesis is the negative sum of the complexity of the clauses to be asserted in the program. The clauses comprise*

the inductive hypothesis together with unit clauses for each of the discovered facts that are not covered by the hypothesis.

Because the complexity due to facts not covered by a particular hypothesis equals the total complexity of all discovered facts less the complexity of those covered by the hypothesis in question, the *relative value* of a partial hypothesis is sufficient for comparison with other partial hypotheses in the partial hypothesis list. Unlike the value, the relative value of a partial hypothesis is unchanged whenever a new fact is discovered in the course of experimentation with a different hypothesis.

Informal Definition 6.4 (Relative value) *The* relative value *of a partial hypothesis is the sum of the complexity of the exclusive cover of the inductive hypothesis less the sum of the complexity of the clauses of the inductive hypothesis.*

For some partial hypothesis types the inductive hypothesis it represents and may eventually become is not yet determined. For these an estimate of the complexity is made — being an upper bound on the future actual complexity. For most partial hypothesis types the cover of the hypothesis is not yet determined either. For these the estimate made represents a lower bound on the future actual cover. Let us see precisely how this estimate is made for each partial hypothesis type. In the following, the attributes of a partial hypothesis are named as they are in the schematic code fragments.

6.6.1.1 Linear type

The value of a linear hypothesis H comprising a single definite clause to become the input clause for absorption is calculated by a simple application of the relative value principle.

$$complexity(\mathsf{H.cover}) - complexity(\mathsf{H.input})$$

6.6.1.2 Paired type

A partial hypothesis H of paired type is a pair of a flat input clause and a flat background clause from the program. It is not known whether the pair is suitable for absorption but on the optimistic assumption that it would be, the value is given by

$$complexity(\mathsf{H.cover}) - complexity(\mathsf{H.input}) - \mathsf{H.level} + complexity(\mathsf{H.background}) - 2$$

This estimates the complexity of the cover less the complexity of the inductive hypothesis that might be generated by f-absorption of the input clause with background clause. The cover is underestimated as the cover of the input clause — because the more general clause has at least the same cover. The complexity is estimated as the complexity of the inductive hypothesis of f-absorption, assuming that the pair is suitable and that the body of the background clause has the same number of literals as

the suitable instance of it. The complexity of this clause would be the complexity of the input clause less the complexity of the body of the background clause plus the complexity of the head of the background clause. Because it is flat, the latter term is exactly one and the complexity of the body of the background clause is the complexity of the full background clause less one. Also, H.level extra antecedents are assumed to be in the input clause by f-absorption, so these are also taken into account in the complexity calculation.

6.6.1.3 Restrict type

A restrict partial hypothesis comprises a definite clause generated by absorption, the cover of which has not been evaluated. But the cover of a restrict hypothesis is at least the cover of the input clause. The relative value of a restrict hypothesis H is

$$complexity(\mathsf{H.cover}) - complexity(\mathsf{H.absorb})$$

6.6.1.4 Testing type

In a testing partial hypothesis the exclusive cover of a hypothesised clause is evaluated by the iterative deepening cover interpreter coupled with verification of the covered atoms. Covered atoms not yet verified make up query, but those subsequently found to be true migrate to cover and those false to faulty. The relative value of the hypothesis optimistically assumes that each atom of query will be found true whenever faulty is empty. Otherwise, once even a single exception is found, the relative value assumes that the complexity of query is distributed between true atoms and false atoms in the same ratio as the complexity of the known atoms in cover and faulty are distributed between those two sets. This encourages testing of hypotheses found to have high cover, and thus promotes discovery of new atoms, but discourages development of hypotheses found to have flaws in proportion to the significance of the flaws.

The clause of the hypothesis is a definite clause but when it is assimilated it will be excepted if necessary to exclude any atoms in faulty. Therefore the relative value of a testing hypothesis includes a penalty on the complexity of the hypothesis: two for the inclusion of an exception literal plus the total complexity of the exceptions for exception predicate-defining unit clauses.

Thus the relative value of a testing partial hypothesis H when H.faulty is empty is given by

$$complexity(\mathsf{H.cover}) + complexity(\mathsf{H.query}) - complexity(\mathsf{H.hypothesis})$$

When H.faulty is non-empty it is given by

$$\begin{aligned}&complexity(\mathsf{H.cover})\\&+ \left(complexity(\mathsf{H.query}) \times \tfrac{complexity(\mathsf{H.cover}) - complexity(\mathsf{H.faulty})}{complexity(\mathsf{H.cover}) + complexity(\mathsf{H.faulty})}\right)\\&- (complexity(\mathsf{H.hypothesis}) + 2 + complexity(\mathsf{H.faulty}))\end{aligned}$$

6.6.1.5 Complete type

A complete hypothesis is one for which all the exclusive cover to the depth bound of the cover interpreter has been verified. In general its relative value is given by the complexity of cover less the complexity of the hypothesis, allowing as before for the exception of the hypothesis and the adoption of the exception-defining clauses if any cover was found to be false.

Before generalization, the relative value of the initial complete hypothesis at the root of the generalization hierarchy is zero because the complexity of the single atom of cover is identical to the complexity of the unit clause. For every other complete hypothesis the relative value is merely inherited from the testing stage of development. If the complete hypothesis H is assimilated it will be excepted if faulty is non-empty. Although in practice the hypothesis is only excepted at the time of assimilation, if we assume that *Clauses* in the following expression comprises the inductive hypothesis (H.hypothesis), excepted if necessary, together with any associated exception-defining clauses, and we see that the relative value of H amounts to

$$complexity(\mathsf{H.cover}) - complexity(Clauses)$$

If the hypothesis is assimilated then the relative value maps directly to the value of the revision it represents by subtracting the complexity of the full set of true discovered atoms.

6.6.2 Best background clause

Earlier we described the pairing of an input clause with each background clause in procedure paired (figure 6.5) in preparation for absorption. The background clauses are ordered for this purpose, best first. Referring to the value of paired partial hypotheses, we can see that the background clause giving the highest value is that of the greatest complexity. This is because, if the input clause is suitable for absorption with it, the resulting restrict hypothesis will be of highest value. Accordingly, the best-background and next-best-background of procedure paired refer to clauses in the program ordered by decreasing complexity. Clauses of equal complexity are ordered temporally, as they occur in the program.

6.7 Hypothesis Assimilation

Eventually the search for a revision to correct a diagnosis terminates: either because it is complete or because it is interrupted. In any case the best revision computed up to the point of termination is assimilated into the program.

6.7.1 Adequacy

The best revision at a point of interruption during best-generalization is defined by the partial hypothesis of highest value which is *adequate*.

Any complete hypothesis is *adequate*. This includes the initial unit clause at the root of the generalization hierarchy, so there is always at least one adequate hypothesis available. Because testing of a hypothesis is time-consuming, especially if it has a large cover and a consequentially high value, it is useful to also permit a testing clause to be adopted without requiring testing to be complete. To implement this principle, a testing clause is considered adequate if, at the time of interruption, it has no faulty atoms and it has a value higher than a threshold. The threshold value is arbitrary, implemented as a parameter to MINERVA.

6.7.2 Assimilating a partial hypothesis

If interrupted during the search for generalizations in procedure best-generalization, the highest-value adequate partial hypothesis is translated to a revision and immediately applied. If its inductive hypothesis is about an observational predicate then the revision comprises the inductive hypothesis supplemented to include a negative exception literal if faulty is non-empty, and auxiliary unit clauses defining each atom of faulty as instances of the exception predicate. On the other hand, if the inductive hypothesis is about an exception predicate, it cannot be excepted further but is instead supplemented with auxiliary unit clauses defining each atom of faulty as instances of the corresponding observational predicate.

The revision also comprises unit clauses to be asserted for any true atoms discovered during the procedure which are not included in the known cover (cover and query) of the hypothesis.

6.8 Minimum Description Length

This heuristic value concept embodied in MINERVA is based on the *minimum description length* principle, introduced by Rissanen [1978]. According to this principle, the best theory representation minimises the sum of the description length of a hypothesis and the description length of the data coded relative to the hypothesis.

In MINERVA the heuristic value identifies the "description length" of a hypothesis with the complexity measure. A chosen revision minimises the increase in description length necessary to incorporate the data into the theory. Discovered facts *not* covered by an inductive hypothesis contribute to the length by virtue of their own complexity — requiring assertion of separate unit clauses. Facts covered by a hypothesis are encoded by the hypothesis, hence make no contribution to complexity. Redundancy checking during sleeping ensures that subsequent changes to the program have not made clauses

more complex than the facts they exclusively cover: the description length of the data must justify a clause.

More commonly, the description length is measured by the number of bits in a string which encodes a theory representation and data according to some predefined coding scheme. For example, Muggleton, Srinivasan and Bain [1992] define a suitable coding scheme for programs and facts based on identifying a program with a universal Turing machine program and the facts as proofs of the data from the program; Georgeff and Wallace [1984] define a scheme for bit string encoding of theories and data represented as probabilistic finite state automata.

Of course, the coding scheme affects the preference order of revisions. In MINERVA's model, predicate, function and constant symbols are not encoded as bit strings — they represent themselves and each counts once. On the other hand, bit string coding schemes can be time-consuming to compute and rely on the availability of the complete language of observation at the time of development of the coding scheme. In our interactive learning scenario, this could change with each new example. The major observable difference between MINERVA's complexity measure and the bit string coding schemes is that because they are based on the frequency of occurrence of symbols, the latter tend to favour hypotheses that use symbols which occur often in the program and data. This difference is unlikely to be very significant to long-term learning.

6.9 Summary

This chapter concludes the procedural description of the learning algorithm MINERVA. It describes a simply evaluated heuristic based on the minimum description length principle that induces an order of preference on revisions and which encourages the discovery of new facts by experimentation.

The heuristic guides a decision to specialize either by removing a clause or by generalizing the definition of exceptions to a clause. In generalization, the heuristic determines the focus of attention of resources towards alternative generalizations, directing the search through the generalization hierarchy defined by absorption. When investigation of revisions is prematurely terminated or is completed, the heuristic determines which of the revisions evaluated is applied.

The heuristic is also used to simplify the program structure occasionally.

7

Modelling the Environment

7.1 Introduction

In order to empirically evaluate the learning performance of MINERVA, we need something to take the part of the environment according to the model we have assumed. SAMPLER fulfills this role for any co-operative incremental learner, and in particular for MINERVA.

SAMPLER's main task is to select examples from a target theory for incremental presentation to MINERVA, and to answer questions posed by MINERVA by referring to the target. It acts as an interface to the target on behalf of MINERVA, to which the target is available only through examples and questions. SAMPLER draws examples from the language of the target according to a *sample strategy*, several of which are provided by SAMPLER. Appropriate presentation of examples from a target, especially when the language includes function symbols, raises a number of difficulties and SAMPLER incorporates a variety of solutions to these difficulties.

SAMPLER also acts as a timekeeper for a learner by interrupting MINERVA whenever it has spent too long responding to a command. This limits the time available to MINERVA for learning from any single example in a way that surprises MINERVA: it is not informed in advance of the amount of time available.

7.2 The Target

The target is represented by a logic program. This is an ordinary normal program, that may be executed by any PROLOG interpreter. The language of observation made available to the learner is the language of the predicate, constant and function symbols

of the program. A Herbrand model of the program, as computed by the SAMPLER interpreter, is the theory from which examples are drawn and membership queries answered. Positive examples are true atoms in the model and negative examples are false.

7.3 Sample Strategies

Examples are selected from the target for presentation to MINERVA according to the manner of a user-selected *sample strategy* that determines both the examples and the order of presentation. The strategies offered by SAMPLER fall broadly into classes: deterministic, complexity-probability, arity-probability, positive and alternating. The basic behaviour of the strategies may be modified by using *conditional loading* controls in the target program file.

Depending on the sample strategy, examples are generated and classified by one of two program interpreters, the generating interpreter (*g-interpreter*) and the testing interpreter (*t-interpreter*). Each is a modified SLDNF-refutation procedure; a detailed description is given in section 7.4. For every sample strategy questions are answered using the t-interpreter.

Most of the strategies incorporate probability-based random sampling. The sampling of these strategies is modified by the user-selected choice of the probability density function used. Before describing the individual sampling strategies, we describe the probability functions they use.

7.3.1 Probability distributions

The selection of discrete (integral) random variables are made from continuous probability density functions. The corresponding cumulative probability distributions are defined by the following functions. They are each parameterized by *min*, the least possible value for the random variable and *max*, the greatest possible value for it.

uniform	$F(x) = (x - min)/(max - min + 1)$
square	$F(x) = \sqrt{(x - min)/(max - min + 1)}$
inverse	$F(x) = ln(x - min + 1)/ln(max - min + 2)$

Given one of the above cumulative distributions, F, for any integer X such that $min \leq X \leq max$ the probability of selecting X is given by $F(X + 1) - F(X)$.

The uniform strategy makes each value for X equally likely. The square distribution is biased towards lower values for X, and the inverse is biased further towards lower values. The inverse function is based on Rissanen's [1983] universal prior probability function for the positive integers. His function is not bounded above like ours, but an upper bound is important in practice because of bounded computing resources.

7.3.2 Deterministic strategies

7.3.2.1 Ordered strategy

The ordered strategy presents examples in the order they appear in the target program. The atom at the head of each clause, in order of the clauses in the program, is used as a goal for which the g-interpreter is invoked. Each answer substitution, in the order produced by the g-interpreter is applied to the atom, and each result presented as an example. Consequently, subject to conditional loading controls, all examples are positive. The g-interpreter's usual upper bound on the number of answers to a goal is not enforced because atoms are not stored.

The ordered strategy is unsuitable for programs representing large theories of multiple concepts because it will produce every atom covered by a single clause before progressing to another concept.

7.3.2.2 Herbrand strategy

The herbrand strategy generates examples by enumerating the Herbrand base of the target program. The chief advantage of the herbrand strategy over the ordered one is its completeness — it enumerates the complete Herbrand base even when it is not finite due to the occurrence of function symbols in the language of the target. Although it is not a very useful strategy it is described here as a precursor to more practical variations.

Initially, the complete program is examined for the collection of all predicate, function and constant symbols occurring in it: the alphabet of the target. Examples are generated as needed by constructing atoms from the alphabet in order of increasing term depth, and then determining their classification using the t-interpreter. Recalling that the term depth of an atom indicates the maximum number of function symbol applications to a term in the atom (definition 3.16), this enables a complete enumeration of the Herbrand base even when it is infinite, because the alphabet is finite. For any particular term depth, atoms are ordered as follows. Predicate symbols are ordered by increasing arity and, for predicate symbols of the same arity, in alphabetic order. The instances of a particular predicate of a particular term depth are ordered as follows. For each argument a functor is chosen, in order of increasing arity and alphabetic order within that. For functors of non-zero arity, their arguments are instantiated in the same fashion, limited by the term depth. The terms for each argument are such that the rightmost changes the fastest and the leftmost the slowest.

It is apparent that the order of presentation of examples in the herbrand strategy resembles a syntactic-simplest-first approach. Whenever the alphabet does not contain function symbols, so the Herbrand base is finite, examples are ordered in increasing arity of predicate symbol. When function symbols are present, examples are ordered in increasing term depth, and within that by increasing predicate symbol arity.

The herbrand strategy has drawbacks. Although it is complete, examples are never

repeated. The order of presentation of examples is deterministic: this would appear to be highly unnatural as an environmental model.

7.3.3 Complexity-probability strategies

The complexity-probability strategies randomly select atoms from the Herbrand base of the target language and then use the t-interpreter to determine their classification. There are three strategies in the class: one for each of the probability distributions. Accordingly the strategies are named *uniform-complexity*, *square-complexity* and *inverse-complexity*.

These strategies aim to present examples in a "natural" way which is random but ensures that simple examples are more likely to be presented earlier and more often. Muggleton and Quinlan [1993] first suggested the idea to base probabilistic example selection on a complexity measure, randomly sampling the complexity from Rissanen's [1983] universal prior probability function.

Atoms are randomly selected from the Herbrand base according to *atomic-complexity*, which is defined as follows.

Definition 7.1 (Atomic-complexity) *The* atomic-complexity *of atom* $p(t_1,\ldots,t_n)$ *is* $1 + \sum_{i=1}^{n} complexity(t_i)$ *where* $complexity(t)$ *is defined for term* t *as follows. For function symbol* f *of arity* m *and terms* s_i $(i = 1,\ldots,m)$, $complexity(f(s_1,\ldots,s_m))$ *is* $1 + \sum_{i=1}^{m} complexity(s_i)$. *For constant symbol* a, $complexity(a)$ *is 1.*

When an example is needed, one is constructed like this. First, a complexity factor greater than zero but less than a predefined fixed upper bound is selected randomly from the indicated probability function. If one exists, an atom of atomic-complexity equal to the complexity factor is constructed using the target language symbols as follows. Random selections made during the construction of the atom are all taken from the uniform distribution.

An arity less than the complexity factor is chosen randomly: each arity of predicate symbols in the language other than zero is equally likely. One of the predicate symbols of that arity is then chosen randomly. Having decremented the complexity factor to allow for the predicate symbol, terms for instantiation of the arguments of the predicate symbol are constructed. A complexity factor for each argument is randomly chosen, in a random order, to sum to the complexity factor. Terms of each of those complexities are constructed as before: for each term the arity of a functor is chosen randomly, a function symbol of that arity chosen randomly, and its arguments are chosen in a random order. When the complexity factor of a term is to be zero, a random constant is selected to instantiate that term.

Finally, the classification of the example is determined by the t-interpreter acting on the atom as a goal.

When the alphabet contains no function symbols, the complexity-probability strategies coincide: generating atoms about each predicate with uniform probability, each argument instantiated by constants with uniform probability. This offers a reasonable random domain-independent sampling strategy. Examples, especially simple ones, are often repeated by the complexity-probability strategies.

A difficulty of the complexity-probability strategies is that the process of example generation may be quite slow, especially for languages with function symbols but no unitary function symbols. The example generation process may often spend time attempting to construct an atom of a complexity for which there is no atom in the language. The *arity-probability* sampling strategies are more efficient than probability-based sampling strategies.

7.3.4 Arity-probability strategies

The arity-probability strategies also use a complexity-based probability distribution to construct atoms from the Herbrand base of the target. Like the complexity-probability strategies they are parameterised by the choice of probability density function, so the strategies in this class are *uniform-arity*, *square-arity* and *inverse-arity*. The arity-probability strategies randomly sample from the probability distribution to determine the arity of a functor for a term as it is being constructed. This is often more efficient than the complexity-probability strategies, but it is difficult to characterise the probability distribution of the examples constructed in this way.

As each example is required its predicate symbol is chosen randomly from all the predicate symbols of the target, such that each symbol is equally likely. Each argument of the predicate symbol is chosen by first selecting an arity according to the selected probability distribution. A functor of the chosen arity for that term is selected randomly from all the functors of that arity in the alphabet — each being equally likely. If the chosen functor is of non-zero arity (that is a function symbol) then each argument term is constructed in the same way. In each case the parameters of the distribution function are defined by *min* being the least arity of function and constant symbols in the target and *max* being the greatest arity. Whenever there is no functor of arity equal to the random arity selected, another arity is independently selected. In all cases a fixed upper bound on the term depth is imposed to ensure that the sampling terminates in bounded time.

In this way an atom is built up, using the selected probability distribution to determine the arity of each term functor. Construction of a term terminates when a functor of arity zero is chosen, and is more likely to terminate sooner for lower arities. This ensures that atoms of lower term depth are more likely, but exactly how likely is difficult to characterise. The uniform strategy makes each arity equally likely: hence examples drawn from the uniform strategy tend to be deep. The square strategy favours lower arities, so the examples tend to be simpler, and the inverse distribution even more so.

When an atom has been constructed in this way its classification is determined by the t-interpreter.

In practice, the examples produced by the arity-probability strategies are similar to those of the corresponding complexity-probability strategies. The major difference is that arity-probability strategies present an example of each predicate symbol with equal probability, while complexity-probability strategies present examples of predicates of lower arity with a higher probability. A predicate symbol of an arity the same as many other predicate symbols is less likely to have an example of it presented than a predicate symbol with an arity shared by few other predicate symbols.

Although in practice arity-probability and complexity-probability strategies present examples with those of greater simplicity being of higher probability, and hence more frequent, an empirical problem arises. In relatively complex programs incorporating many function and constant symbols, most of the complex atoms of a particular predicate are negative examples. Indeed, complex positive examples may be so rare that a learner may learn a very simple and false theory which has a very high degree of accuracy on the examples. The *positive* strategies address this deficiency.

7.3.5 Positive strategies

The positive sample strategies are also probability based but they present only positive examples. There are three strategies in the class: *uniform-positive*, *square-positive* and *inverse-positive* corresponding to each probability function. For these strategies the chosen probability function is used to randomly determine a factor and an atom of that atomic-complexity is randomly selected from a pre-computed set of true atoms if one exists.

The set of true atoms is constructed by treating an instance of each predicate, instantiated by distinct variables in each argument position, as a goal for invocation by the g-interpreter. Each answer substitution is applied to the goal, giving true atoms. The atoms thus formed are grouped into atomic-complexity classes.

The presentation of examples proceeds by randomly selecting a complexity in the range up to the maximum complexity of the pre-computed examples. The selection is done by the inverse probability distribution, setting *min* to 1 and *max* to the maximum complexity. One of the positive examples of that complexity, if there is one, is selected (each with equal probability) and is presented to MINERVA.

The positive strategies present only positive examples. For typical domains true atoms of a given atomic complexity are sparse amongst all atoms of that complexity, especially for higher complexities, so the approach economises on storage requirements. However, most learners need negative examples to counter a tendency for over-generalization.

7.3.6 Alternating strategies

The *alternating* strategies offer a compromise between the positive-only positive strategies and the mostly negative complexity-probability or arity-probability strategies.

The alternating strategies draw examples from a pair of other strategies. One strategy of the pair is a positive strategy and the other is either a probability-complexity or an arity-complexity strategy. Any of the three probability distributions may be used for sampling but the same distribution is used for both strategies of the pair. This permits six alternating strategies: each of the three positive strategies paired with the corresponding probability-complexity strategy and each of the three positive strategies paired with the corresponding probability-complexity strategies.

Each example of the alternating strategy is drawn with equal probability from one of the paired strategies comprising it. This means that positive examples are presented with probability of at least one-half and examples of lower complexity are presented with a higher probability. However, when true atoms are sparse the ratio of unique negative examples to unique positive examples increases with the sample size, because the rarer positive examples are more likely to be repeated.

7.3.7 User-control of example selection

Finer control over sampling may be achieved using *conditional loading* control facilities of NUPROLOG [Thom and Zobel 1990] in the target program. Using a special goal *evalonly* with the conditional loading facility, it is possible to define a subset of the atoms of the target from which SAMPLER selects examples according to the sample strategy chosen. This subset may be defined by a subset of the predicate, function and constant symbols of the target or by a subset of atoms or clauses of the target program. In any case, the complete target is used for evaluation of the classification of atoms for examples or questions.

7.4 Target Program Interpreters

Two interpreters, the *t-interpreter* and the *g-interpreter*, are used for the evaluation of goals with respect to the target program. They are simpler and less robust than the interpreters of MINERVA because a designer has control over the structure of the program.

7.4.1 Testing ground atoms

The t-interpreter is used for testing evaluation of ground atomic goals which arise as questions posed by a learner or as atoms selected from the Herbrand base. It is almost a standard PROLOG interpreter, being an SLDNF-refutation procedure with a computation rule which always selects the leftmost atom in a goal and selects clauses in the order they appear in the program. This implements *unsafe* negation.

The only non-standard feature of the t-interpreter is the use of a fixed depth bound to limit the length of SLD-derivations constructed in the search for an SLDNF-refutation.

This depth bound, presently set at 50, avoids some non-terminating computations but does not prevent them in the cases of example 3.9.

7.4.2 Generating ground atoms

Ground atoms for positive examples are generated from a program representing a target by the g-interpreter. The goals for g-interpretation may be non-ground; standard SLDNF-refutation procedures typically encounter non-termination difficulties with such goals. To resolve this, the g-interpreter searches SLDNF-trees in an iterative-deepening fashion (see section 3.6.2.1), and bounds the search with a limit on the length of SLD-derivations, presently being 25. The standard computation rule which selects left-most literals is used: resulting in unsafe negation. Subsidiary SLD-derivations constructed during evaluation of negative literals in goals are bounded in length but are explored in the standard depth-first fashion rather than by iterative deepening.

The g-interpreter also imposes an upper limit on the number of unique answers returned in the search for the SLDNF-refutations of an initial goal. The upper bound, currently set at 500, enables storage of the atoms during the execution of SAMPLER. Because of the iterative-deepening nature of the g-interpreter, the instances recorded are those with the shortest top-level SLDNF-refutations — typically the "simplest" ones.

The designer of the target program must ensure that answer substitutions computed by the g-interpreter are ground. If it is not possible to design the target program in this way then sample strategies using the g-interpreter must be avoided.

7.5 Time Limits

SAMPLER enforces user-defined time limits on example processing by a co-operative learner program. A parameter to SAMPLER defines the real time in seconds available to the learner for each example. This can be either a fixed number for every example, or the upper bound of a range from which the number for each example is uniformly randomly sampled. Whenever SAMPLER presents an example to the learner a timer is started. If the learner asks questions during processing of the example the timer is suspended until SAMPLER makes the answer available. When the time limit has expired SAMPLER interrupts the learner by a *signal*, presents another example to the learner, and starts the timer again. If the learner requests another example before the time has expired then the timer is reset for the next example.

7.6 Other SAMPLER features

7.6.1 Sleep control

SAMPLER can schedule regular sleep commands for the learner under its control. This is done by providing parameters to SAMPLER that specify the frequency and duration. The frequency is defined by the number of intervening examples to be presented between sleep commands; the duration is specified as the number of seconds available for the learner to respond to the command. If it has not responded earlier, SAMPLER interrupts the learner after the specified time period with a signal, and another example is presented.

7.6.2 Statistics

SAMPLER collects a number of statistics during the life of the learner under its control and reports them on termination. These include the total number of positive and negative examples presented, the number of those that were unique, the number of questions that were answered, the number of times sleep was scheduled, and the number of times that example processing by the learner was interrupted by time out signals.

7.7 Summary

SAMPLER is an environmental modelling tool. It selects examples from a program representing a target theory according to a user-controlled sample strategy. Sample strategies present examples deterministically or randomly; some of the random ones present simple examples sooner and more often than complex ones.

SAMPLER also enforces time limits for example processing or sleeping by a learner. The learner is interrupted by a signal when a time allowance is expired.

8

Performance evaluation

8.1 Introduction

In this chapter the learning performance of MINERVA is illustrated by analysing attempts to learn several different target theories. Some of the theories are taken from the machine learning literature to enable comparison of MINERVA with other ILP learners. In many cases reported experiments with other learners are as nearly as possible replicated for evaluation of MINERVA. MINERVA's performance is evaluated according to the reported criteria applied to those other learners.

The reader should be aware that MINERVA is designed for conditions and purposes quite different to other learners of the machine learning literature. The final series of experiments in the chapter evaluate MINERVA in terms of its ability to achieve the goals for which it was specifically designed.

For the experiments described here MINERVA and SAMPLER were compiled under NUPROLOG version 1.5.24 [Thom and Zobel 1990] and executed on a Sun Sparcstation IPX configured with 32 Mb memory. The implementation of MINERVA is an experimental prototype and has not been optimised for time or space efficiency.

8.1.1 Experimental design

For most experiments the SAMPLER tool was used to generate examples. A single random seed of value 1 was used unless otherwise indicated. In repeat exercises sequential seeds were used $(1, 2, 3, \ldots)$.

Examples were selected from the target according to the sample strategies described in chapter 7. The sample strategy was chosen according to characteristics of the target

domain. In large domains (particularly where there are function symbols) an alternating strategy was used to ensure a reasonable ratio of positive examples occurring in the input. A complexity probability strategy was preferred to an arity-probability strategy in smaller problems, but the improved efficiency of the example generation for larger problems again led to the choice of an arity-probability strategy. In domains without function symbols the probability distribution is irrelevant. In domains with function symbols, an inverse probability was usually used to ensure a high probability of early presentation of simple examples.

Experimental results are reported in terms of numbers of "unique" examples presented to MINERVA in order to learn a particular theory. This does not count repeated examples nor any example which MINERVA has already asked about prior to its presentation.

In most cases regular sleeping was permitted for lengthy periods of time. Personal judgement was used to set example-processing time limits and sleep duration parameters. Larger and more complex domains, particularly highly recursive ones, usually require longer times for effective learning, but the times used for the experiments were not necessarily the minimal necessary to achieve the reported results — in most cases learning performance was not sensitive to small variations in the parameters and no attempt was made to minimise them. Of course, the actual time period used is not very meaningful as the extent to which it correlates with the time available for learning by MINERVA depends on machine configuration and other compute activity. The parameters used for the experiment are reported but a close examination of them is unwarranted.

When these general principles require adjustment for particular experiments justification is given.

8.2 Learning by Absorption

In the first experiment we compare the performance of MINERVA with its closest relative, MARVIN on an English grammar domain [Sammut 1981b]. MARVIN also generalizes by absorption, although in a different way to MINERVA. It has no diagnosis or revision capability and generalization performance is very strongly dependent on optimal ordering of carefully chosen input examples. MINERVA achieves similar results with a random presentation of examples.

The target theory describes a very small English grammar. A "dictionary" is given as initial background knowledge to the learners: defining two each of nouns, proper nouns, determiners, intransitive verbs, and transitive verbs as follows.

$$noun(giraffe) \leftarrow$$
$$noun(apple) \leftarrow$$
$$pnoun(john) \leftarrow$$

pnoun(mary)←
det(the)←
det(a)←
iverb(dreams)←
iverb(sleeps)←
tverb(dreams)←
tverb(eats)←

In addition to the dictionary, the target defines four predicates: *np* for noun phrase, *vp* for verb phrase, *sent* for sentence and *list* for word sequences. Also in the target there are the binary function symbol ".", for sequence construction and the constant "[]" for sequence termination: these symbols are represented in the customary notation for PROLOG in the program fragments below.

MARVIN required specially crafted examples, with unique constant symbol "names" given to each noun phrase, verb phrase and list, in order to enable MARVIN to locate "relevant" background knowledge. MINERVA does not require these symbols. The examples for our experiment were drawn directly from the target theory, represented by the program reported as successfully learnt by MARVIN [Sammut 1981b], using a uniform alternating arity-probability strategy.

The problem domain is fairly simple because it involves only a small number of symbols, but the use of a function symbol and the need for mutually recursive and non-*nvi* clauses make it beyond the capabilities of some learners.

8.2.1 Results

MINERVA was permitted 60 seconds for each example and one uninterrupted sleep afterwards. Learning was terminated after its program appeared to be stable. This happened when a total of 46 examples had been presented, although only ten of these were distinct positive examples. Only the following eight positive examples prompted revision of the program by MINERVA and in each case MINERVA was interrupted when learning them.

sent([a, giraffe, dreams, john], [])
sent([the, giraffe, eats, a, giraffe], [])
list([eats, a, giraffe])
np([mary, eats, a, giraffe], [eats, a, giraffe])
np([the, apple, dreams, the, giraffe], [dreams, the, giraffe])
list([dreams])
sent([mary, dreams], [])
vp([dreams, mary], [])

MINERVA asked a total of 117 questions, of which only 15 were answered negatively, to learn a correct program of the initial program plus the following clauses. Apart from clause order and minor notational differences, the program is the same as that learnt by MARVIN:

np([X| Y], Y)← list(Y), pnoun(X)
vp([X], [])← iverb(X)
np([X, Y| Z], Z)← det(X), list(Z), noun(Y)
list([])←
list(X)← vp(X, Y)
sent(X, Y)← np(X, Z), vp(Z, Y)
vp([X| Y], [])← np(Y, []), tverb(X)

In the course of learning the following clauses were also adopted by MINERVA but later removed from the program because they were redundant or too complex.

sent([X, Y, Z, U], [])← tverb(Z), noun(Y), det(X), pnoun(U)
sent([X, Y, Z, U, Y], [])← tverb(Z), noun(Y), det(X), det(U)
list([X, Y, Z])← tverb(X), noun(Z), det(Y)

The number of hand-picked examples given to MARVIN is not reported. In contrast with MINERVA's 117 questions, MARVIN asked only 23 questions to learn the same program. MARVIN depends on carefully chosen examples and the teacher's instruction to terminate search for hypotheses when the teacher recognizes the current hypothesis to be the desired one. MINERVA instead depends on comprehensive environmental experimentation to choose hypotheses.

8.3 Incremental Learning

In this experiment the ability of MINERVA as an incremental learner is demonstrated by comparing it with the incremental ILP learner, CLINT. In many respects, CLINT is close in nature to MINERVA, although the language of programs of MINERVA is greater, MINERVA is interruptible, and the techniques for generating covering clauses are very different.

To demonstrate MINERVA's performance we replicate a reported experiment with CLINT [De Raedt and Bruynooghe 1992] in the well-known domain of the the card game Eleusis [Dietterich and Michalski 1986]. Although the domain was also used for FOIL performance evaluation [Quinlan 1990], the CLINT experiment is so unlike the FOIL one that simultaneous comparison is not possible.

Initial background knowledge is comprised of definitions for 20 predicates in terms of the constant symbols for suits: *hearts, diamonds, clubs*, and *spades*, and ranks: *ace*, *two*, *three four*, *five*, *six*, *seven*, *eight, nine, jack, queen*, and *king*. The form of the definitions is not reported for the CLINT experiment: we used ground unit clauses for definitions for the unary predicates *odd-rank*, *non-odd-rank*, *odd-suit*, *non-odd-suit*, *face*, *non-face*, *red*, and *non-red*; and for the binary predicates *precedes-rank* and *precedes-suit*. We used functor-free definite clauses for the unary *rank* and *suit* and the binary *same-colour*, *non-same-colour*, *non-precedes-rank*, *non-precedes-suit*, *lower-rank*, *non-lower-rank*, *lower-suit*, and *non-lower-suit*.

The 4-ary target concepts *legal1* and *legal2* are defined in the target program by the successfully learnt definitions of the CLINT experiment, given below. We assume for this purpose that the fourth clause is misprinted in the report [De Raedt and Bruynooghe 1992]; otherwise the problem is too simple and too unlike the FOIL experiment. *legal1* is satisfied when a face card is followed by a non-face card or vice versa. *legal2* is satisfied when a card is followed by one of the succeeding rank but a non-lower suit, or by one of the preceding rank and a lower suit.

We hand-picked input examples to achieve the desired results: like for CLINT we need only one positive example for each clause of the two target concepts, as follows.

legal1(ace, diamond, two, spade)
legal1(three, spade, queen, heart)
legal2(four, heart, five, diamond)
legal2(five, spade, four, club)

SAMPLER answered questions with reference to the target and interrupted MINERVA after 180 seconds for each example.

8.3.1 Results

MINERVA learnt the following correct clauses. All but the second is the same as those learnt by CLINT.

legal1(X,Y,Z,U)← suit(Y), suit(U), non-face(Z), face(X)
legal1(X,Y,Z,U)← suit(U), legal1(Z,Y,X,Y)
legal2(X,Y,Z,U)← non-lower-suit(Y,U), precedes-rank(X,Z)
legal2(X,Y,Z,U)← lower-suit(Y,U), precedes-rank(Z,X)

Instead of the second, CLINT learnt the equivalent but longer clause:

legal1(X,Y,Z,U)← suit(Y), suit(U), non-face(Z), face(X).

MINERVA asked 924 questions, in contrast to CLINT's economical 37. A few of MINERVA's are due to missing answer diagnosis where all existing clauses are suspect irrespective of whether they were initially given, but the major portion are due to its attempt to exhaustively search a larger hypothesis space and to exhaustively test inductive hypotheses under consideration — continuing even after a suitable hypothesis has been found.

8.4 Theory Refinement

Theory refinement algorithms, such as AUDREY [Wogulis 1991], EITHER [Mooney and Ourston 1991], ABE [O'Rorke et al. 1989] and FORTE [Richards and Mooney 1991, Richards and Mooney 1995] are designed to correct an initially incorrect program to account for errors indicated by a batch of examples. In this experiment we examine the performance of MINERVA as a theory refinement algorithm by comparing it with

FORTE. Performance is measured by the accuracy of the learnt theory with respect to a test set that is a subset of the target theory.

Again we use a family relations domain, although a larger one than previously. The particular problem presented here was selected by Richards and Mooney [1995], who give performance results for FORTE along with results for the non-revising single-predicate ILP learners GOLEM [Muggleton and Feng 1990] and FOIL [Quinlan 1990] on the same problem. We aim to replicate the experiments on MINERVA to enable comparison with all three.

The target is represented by a program describing relations between 86 people across five generations. There are twelve target binary predicates to be learnt: *wife*, *husband*, *mother*, *father*, *daughter*, *son*, *sister*, *brother*, *aunt*, *uncle*, *niece*, *nephew* for which there are altogether 744 true atoms. The target also defines two "intermediate" binary predicates *au* and *sibling* and three "basic" binary predicates *gender*, *married*, and *parent*. The basic and intermediate predicates are not represented in examples but complete and correct definitions for the basic predicates are given as an initial correct program. Clauses of the initial program are ground unit, except for one: *married(X, Y)←married(Y,X)*.

Here is an "ideal" correct definition for the intermediate and target predicates:

wife(X, Y)← gender(X, female), married(X, Y)
husband(X, Y)← gender(X, male), married(X, Y)
mother(X, Y)← gender(X, female), parent(X, Y)
father(X, Y)← gender(X, male), parent(X, Y)
daughter(X, Y)← gender(X, female), parent(Y, X)
son(X, Y)← gender(X, male), parent(Y, X)
sister(X, Y)← gender(X, female), sibling(X, Y)
brother(X, Y)← gender(X, male), sibling(X, Y)
aunt(X, Y)← gender(X, female), au(X, Y)
uncle(X, Y)← gender(X, male), au(X, Y)
niece(X, Y)← gender(X, female), au(Y, X)
nephew(X, Y)← gender(X, male), au(Y, X)
au(X, Y)← married(X, Z), sibling(Z, U), parent(U, Y)
au(X, Y)← sibling(X, Z), parent(Z, Y)
sibling(X, Y)← parent(Z, X), parent(Z, Y), noteq(X, Y)

In the FORTE experiment *noteq*, true for any two different ground arguments and used in the definition of *sibling*, is an inbuilt predicate. For MINERVA we gave the definition *eq(X,X)←gender(X, Y)* as additional background knowledge in the initial program, treating *eq* as a basic predicate and replacing *noteq* in the ideal program by $\sim$ *eq*.

Although this theory is moderately large, it should be relatively easy to learn because simple definitions exist, having no function symbols, only one clause per definition and at most three antecedents per clause.

FORTE is aided by auxiliary information describing the dependency structure of desired predicate definitions; there is no comparable hint given to MINERVA. For MINERVA the

particular difficulty with this domain is the large number of clauses and lack of structure in the initial program: being largely comprised of unit clauses.

8.4.1 Experimental method

We replicate two experiments of Richards and Mooney: an induction experiment and a refinement experiment. In each, MINERVA was given five exercises — each exercise using a different random example sequence drawn from the target program using the alternating uniform complexity-probability sample strategy. Results are reported at points where 10, 25, 50 and 100 positive examples (discounting repeated examples) respectively were observed. Our distribution of negative examples is slightly less than 50% overall, but as there are many more false atoms in the target, the ratio of unique negative to unique positive examples grows with the sample size. For the other learners examples were generated randomly but ensuring that on average two-thirds of examples were negative.

In each exercise of MINERVA, 200 seconds was allowed per example and sleeping was permitted for 200 seconds after every 20 examples.

The results of figures 8.1 and 8.2 represent the average sample sizes and achieved accuracy for the five repeated exercises of MINERVA, compared with those reported by Richards and Mooney [1995]. MINERVA's accuracy measure differs from the one used for the other learners in the figure. For them Richards and Mooney define a *sample set* comprising all true atoms of the target predicates (744) and a subset of the false atoms of the target predicates: 1488 randomly generated false atoms. Examples for each learning exercise are drawn from this sample set, and the remaining atoms comprise the test set for that exercise. Accuracy is defined as the proportion of the atoms in the test set that are correct in the learnt theory. For MINERVA's learning we use examples of the target predicates drawn randomly from the target by SAMPLER. We define accuracy for MINERVA as the proportion of atoms in the sample set of Richards and Mooney that are correct in the learnt theory. The difference relatively overrates our reported accuracy by at most 1.2% at the greater sample sizes, and usually much less.

8.4.2 Induction experiment

In the induction experiment the initial program for FORTE, FOIL and GOLEM comprised complete and correct definitions for the basic predicates. It is not clear whether definitions for the intermediate predicates were included; for MINERVA they have not been. The performance of FORTE was compared with the performance on the same sample set by GOLEM version 1.0α and FOIL version 5.1. As the latter two are single predicate learners, they were given an easier problem of twelve independent learning tasks, one for each target predicate.

Figure 8.1 shows comparative results: all but those for MINERVA are interpolated from the empirical measurements of Richards and Mooney [1995]. In the figure, "FORTE-"

Positive Examples	Accuracy % MINERVA	FORTE-	FORTE	GOLEM	FOIL
0	66.7	67	67	67	67
10	74.0	62	76	67	67
25	80.8	72	92	67	86
50	85.6	82	97	67	86
100	92.9				

Total Examples	Accuracy % MINERVA	FORTE-	FORTE	GOLEM	FOIL
0	66.7	67	67	67	67
17	74.0	60	68	67	67
50	80.8	68	85	67	73
125	85.6	79	97	67	83
334	92.9				

Figure 8.1: Experimental results for induction

refers to a version of FORTE with its "relational pathfinding" component disabled: a component which Richards and Mooney consider particularly well suited to this domain.

Because of the difference in distribution of examples, we have shown two comparison points: the first compares performance at equal numbers of positive examples and the second at equal numbers of combined positive and negative examples. In this experiment, because of the extent of the initial background knowledge, the negative examples presented play a very minor role in MINERVA's performance.

We can see that MINERVA's performance compares well with relational batch learners. MINERVA is aided by the ability to ask questions, unavailable to the other learners, but the various extra-logical hints given to the other learners are unavailable to MINERVA.

8.4.3 Refinement experiment

The second experiment investigates the ability of MINERVA to deal with a theory-refinement problem in which a revising learner is given an incorrect theory and a set of examples with which to correct it.

For the FORTE experiment the ideal program given above was randomly corrupted and used as an initial program for learning, supplementing the correct definitions for the basic predicates. A corruption is an application of one of the following corrupting operators: delete rule; add rule (1-3 antecedents); delete antecedent; add antecedent; change antecedent (delete plus add); and change variable. The form of corruptions corresponds closely with the predefined classes of errors which FORTE is designed to

Examples	Accuracy %	
	MINERVA	FORTE
0	86.8	92
15	89.4	93
44	90.0	97
115	90.9	99
236	93.9	100

Figure 8.2: Experimental results for refinement

correct, so we should expect this to aid the performance of FORTE. The reader is referred to Richards and Mooney's [1995] account for more detail.

In our experiment also five randomly corrupted versions of the ideal target program were generated. These programs were generated, as nearly as possible, in the same way as those for the FORTE experiment. We used two programs containing three corruptions and three programs of four corruptions each to achieve the same average corruption as for the FORTE experiment: 3.6 corruptions per program. In two cases the corrupt programs contained a clause with a variable in the head not occurring in the body of the clause: such clauses are not acceptable in MINERVA's program. To each such clause the literal *gender(X, U)*, was added to the body, where X is the variable in question and U is a new variable not occurring elsewhere in the clause.

The corrupt programs supplemented the correct definition for the basic predicates as an initial program before learning commenced.

Our initial average accuracy was 86.8%, due predominately to errors in true atoms; some of the programs are completely correct for the false atoms. The initial average accuracy for the five programs of FORTE was much higher at 91.7%. This difference highlights the variability of the problem, and might be significant for the reported experimental results.

The results are presented in figure 8.2 with the comparative results for FORTE interpolated from Richards and Mooney's [1995] results as before. The table entries represent accuracy achieved for exercises with 0, 10, 25, 50 and 100 unique positive examples respectively in the experiment with MINERVA. This time the accuracy reported in the figure is compared with the accuracy achieved by FORTE in terms of total (positive and negative) sample sizes, since the negative examples have an important role to play in detection of initial false clauses. The version of FORTE for the results here is the better one on this domain: that also labelled FORTE in figure 8.1.

The results for MINERVA are based on just one exercise for each corrupted program at each sample size, using the same (random) example presentation for each, and averaged over the five exercises. Individual exercise results are highly variable. For example, the fifth program has accuracy of 97.2% after only 10 positive examples.

Given the high variability of the results, due to the nature of the corruptions and the extent to which examples enable their diagnosis, conclusions drawn from this experiment must be weak. Recall that the corruptions are designed to be almost exact reversals of the correcting operators of FORTE. Subject to these considerations, we conclude that MINERVA might be a weaker theory refinement algorithm than others.

8.5 Learning Multiple Predicate Theories

We have already demonstrated in section 8.2 that MINERVA is capable of learning programs defining multiple inter-dependent concepts. Usually such programs are the domain of incremental learners, but MPL [De Raedt et al. 1993] is a batch learner designed for them. In this experiment we show that for a simple problem, MINERVA's learning performance compares well with that of MPL.

MPL's first experiment is replicated for MINERVA. Initially the learners were given a background program of ground unit clauses as follows.

male(prudent)←
male(leon)←
male(luc)←
male(willem)←
male(rene)←
male(pieter)←
male(etienne)←
male(bart)←
male(stijn)←

female(laura)←
female(alice)←
female(lieve)←
female(lucy)←
female(esther)←
female(yvonne)←
female(soetkin)←
female(rose)←
female(katleen)←
female(an)←

parent(bart, stijn)←
parent(bart, pieter)←
parent(luc, soetkin)←
parent(willem, lieve)←
parent(willem, katleen)←
parent(rene, willem)←
parent(rene, lucy)←
parent(prudent, esther)←
parent(katleen, stijn)←
parent(katleen, pieter)←
parent(lieve, soetkin)←
parent(esther, lieve)←
parent(esther, katleen)←
parent(yvonne, willem)←
parent(yvonne, lucy)←
parent(leon, rose)←
parent(alice, rose)←
parent(etienne, luc)←
parent(rose, luc)←
parent(etienne, an)←
parent(rose, an)←
parent(laura, esther)←

Examples of the binary target predicates *ancestor*, *mother* and *father* were drawn by SAMPLER from a target program with a uniform complexity-based sample strategy, a 100 second time limit for each example and sleeping permitted only after all examples were presented. In De Raedt et al.'s [1993] experiment the complete set of atoms about the target predicates in the Herbrand base were used. In MINERVA's experiment learning was halted after all 78 true atoms were presented, either as positive examples or as answers to questions, irrespective of the number of negative examples presented at that time.

8.5.1 Results

MINERVA added the following clauses to the initial program, resulting in a correct program. Only six examples prompted revision; all other atoms of the theory were discovered by experimentation or generalization when revising for those six examples.

ancestor(X, Y)← parent(Z, lieve), parent(X, Z), parent(bart, Y)
mother(X, Y)← female(X), parent(X, Y)
ancestor(X, Y)← parent(Z, Y), parent(X, Z)
ancestor(X, Y)← parent(X, Y)
father(X, Y)← male(X), parent(X, Y)
ancestor(X, soetkin)← female(Y), parent(X, Y)

The program learnt by MINERVA is identical to that reported for MPL on clauses for *mother* and *father*. The clause defining *ancestor* in terms of *parent* is also identical. But MPL learnt only one other clause for ancestor: *ancestor(X,Y) ←parent(X,Z), ancestor(Z,Y)* in place of MINERVA's three. Because the domain is so small, MINERVA's definition of *ancestor* is heavily dependent on the presentation order of examples.

The experiment with MINERVA was repeated four more times with different random presentation orders. In every exercise correct programs were learnt, including the same clauses for *mother* and *father*, but the clauses for *ancestor* varied. In one of the five exercises the program learnt by MINERVA was identical to that of MPL.

8.6 Learning with MINERVA

Having investigated MINERVA's performance on several problems to enable performance comparison with a range of other inductive learners, we now turn to investigating the capabilities of MINERVA by intensive analysis of performance in a single domain. In particular we investigate variations to several factors: the amount of background knowledge available; the sampling strategy for examples; the extent of interleaving examples of interdependent concepts; the time allowed for learning and sleeping; and the presence of noise in the environment.

8.6.1 The graph domain

This series of experiments deals with a domain based on graph relations depicted in figure 8.3. The domain concerns thirty-five constant terms related by the predicate *arc* into five disconnected graphs. *arc(a,b)* should be interpreted to mean that there is a directed arc from node *a* to node *b*. In addition to *arc*, we are interested in four other binary predicates and a unary predicate that should be understood as follows.

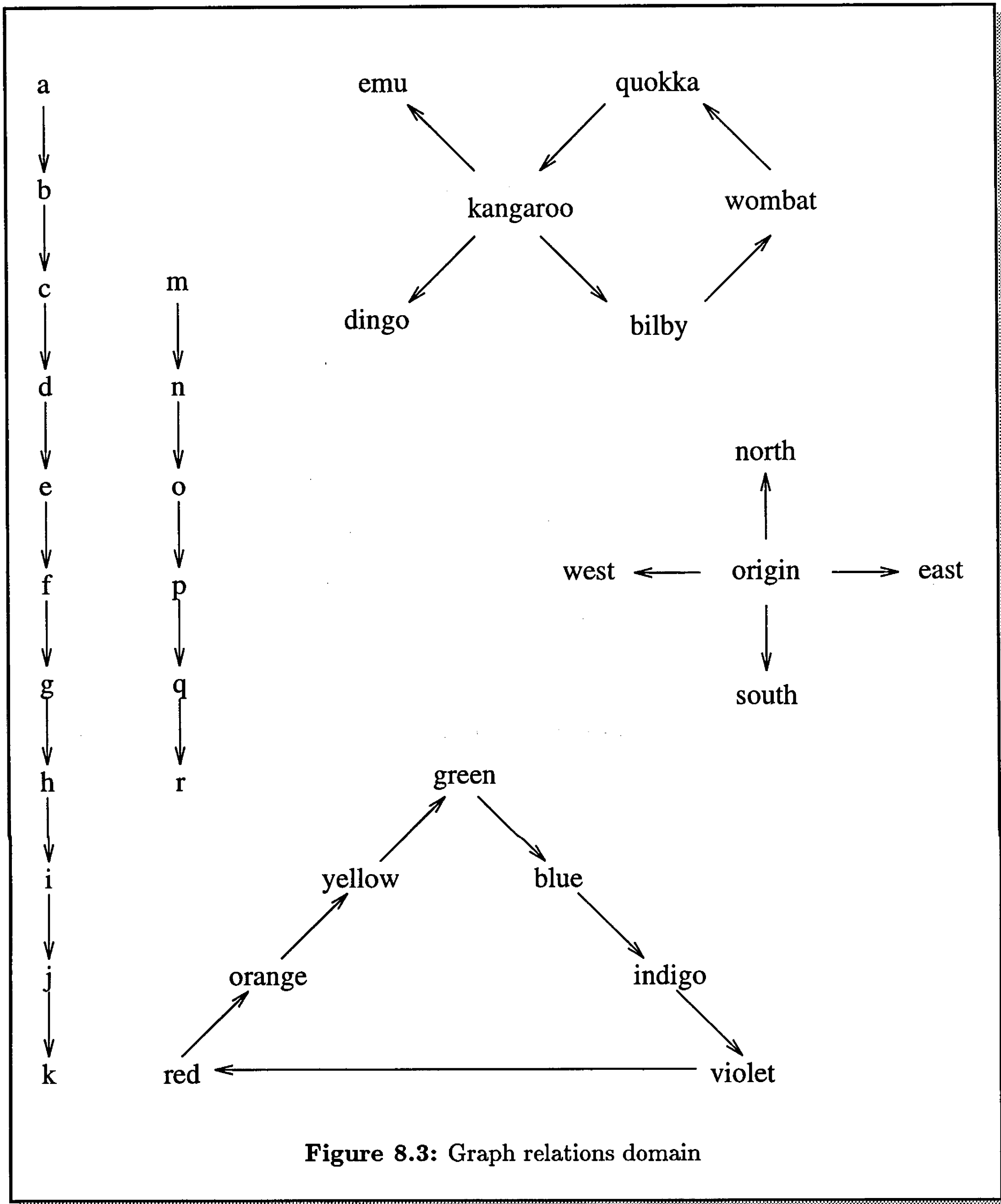


Figure 8.3: Graph relations domain

path(a,c): There is a path, or sequence of arcs, from *a* to *c*.

revpath(c,a): There is a path from *a* to *c*.

cycle(red): There is a path connecting *red* back to *red*.

conn(a,c): There is a path from *a* to *c* or from *c* to *a*.

bicycle(quokka,red): *quokka* and *red* belong to disjoint cycles.

```
arc(a, b)←
arc(b, c)←
arc(c, d)←
arc(d, e)←
arc(e, f)←
arc(f, g)←
arc(g, h)←
arc(h, i)←
arc(i, j)←
arc(j, k)←
arc(m, n)←
arc(n, o)←
arc(o, p)←
arc(p, q)←
arc(q, r)←
arc(kangaroo, bilby)←
arc(bilby, wombat)←
arc(wombat, quokka)←
arc(quokka, kangaroo)←
arc(kangaroo, emu)←
arc(kangaroo, dingo)←
arc(red, orange)←
arc(orange, yellow)←
arc(yellow, green)←
arc(green, blue)←
arc(blue, indigo)←
arc(indigo, violet)←
arc(violet, red)←
arc(origin, north)←
arc(origin, east)←
arc(origin, south)←
arc(origin, west)←
path(X, Y)← arc(X, Y)
path(X, Y)← arc(X, Z), path(Z, Y)
revpath(X, Y)← path(Y, X)
cycle(X)← path(X, X)
conn(X, Y)← path(X, Y)
conn(X, Y)← path(Y, X)
bicycle(X, Y)← cycle(X), cycle(Y), ∼ conn(X, Y)
```

Figure 8.4: Graph relations program

Figure 8.4 shows a target program correctly defining the complete domain. The target theory represented by the program comprises numbers of true and false atoms respectively as follows. *arc*: 32 and 1193; *path*: 147 and 1078; *revpath*: 147 and 1078; *cycle*: 11 and 24; *conn*: 229 and 996; and *bicycle*: 56 and 1169.

8.6.2 Learning interleaved interdependent predicates

Beginning with an initial program comprising the ground unit clause definition of *arc* given in figure 8.4, MINERVA is given examples of the predicates *path*, *revpath* and *cycle*. For this experiment, good definitions for *cycle* and one of *path* or *revpath*, as in the target program of figure 8.4, depend on a good definition of the other of *path* or *revpath*.

The examples are randomly selected by SAMPLER from the relevant subset of the target theory, represented by the program of figure 8.4, by the *uniform-arity* sample strategy. The probability distribution (in this case uniform) is irrelevant because of the absence of function symbols, but the arity-probability class is used to ensure that each predicate is distributed uniformly in the example set. Because positive examples are not sparse in the distribution, there is no need to use an alternating strategy and the arity-probability strategy is sufficient: generating positive and negative examples according to their distribution in the target. By the uniform-arity strategy each predicate is equally likely and every atom of that predicate, whether true or false, is equally likely.

MINERVA is permitted 100 seconds for each example, and sleeping is permitted at intervals of 20 examples for 200 seconds.

Snapshots of the learnt theory are taken at five sample points: when 4, 10, 25, 50 and 100 positive examples respectively have been presented. For this purpose negative examples are not counted but the positive examples at the sample points included both repeated examples and examples given earlier as answers to questions.

The experiment is repeated for five exercises, each using a different random example presentation from the same strategy.

8.6.2.1 Results

The results for each of the five exercises are summarised in figure 8.5. The figure shows the accuracy of each exercise at each sample point and the average accuracy for the five exercises at each sample point. Accuracy is reported only for the concepts *cycle*, *path* and *revpath*, because an accurate definition for *arc* was provided initially.

In addition, the second and third columns of the figure show average total unique examples presented at each point (not including repeated examples or those given as answers), and the average total examples (positive and negative examples, counting repeats) presented at each point. The first line of the figure is a reference point being the accuracy of an empty program that is completely correct for the false atoms of the

Pos Ex	Av Uniq Pos Ex	Av Tot Ex	Accuracy % 1	2	3	4	5	Av
0	0	0	87.7	87.7	87.7	87.7	87.7	87.7
4	3.6	34.6	89.3	89.4	89.1	90.0	88.4	89.2
10	7.6	74.2	91.0	94.9	90.8	89.0	90.1	91.2
25	11.6	180.6	95.0	100.0	-	92.6	99.9	96.9
50	13.8	359.2	100.0	100.0	95.1	94.4	100.0	97.9
100	15.6	707.6	100.0	100.0	98.2	97.7	100.0	99.2

Figure 8.5: Learning three graph relations

target.

Let us examine the results of the first exercise in more depth. Following is the program learnt at the reporting point of 50 examples, excluding the clauses for *arc* which remain unchanged from the initial input program.

cycle(red)←
cycle(wombat)←
cycle(X)← *arc(X, Y), cycle(Y)*
path(X, Y)← *arc(X, Y)*
path(X, Y)← *arc(Z, Y), path(X, Z)*
revpath(X, Y)← *path(Y, X)*

Although slightly different from the relevant part of the target program of figure 8.4, the acquired program is correct and concise. The definition here for *cycle* could be found earlier than that in the figure because it does not depend on a good definition of *path*. Because *cycle* is only a unary predicate there are fewer true or false atoms about *cycle* in the target, so the arity-probability sampling strategy ensures that a higher proportion of instances of *cycle* are presented relatively early in the learning session. A lower proportion of *path* instances (particularly positive ones) are incorporated in the developing theory at the same point.

Earlier clauses acquired during learning this program included several ground unit clauses for each of the predicates and more complex clauses like *path(X, Y)*←*arc(Z, U), arc(X, Z), arc(U, Y)* and the pair *revpath(k, X)*←*path(a, X), not* $revpath^0$*(k, X)* and $revpath^0$*(k, k)*←. Although true, these were all recognised to be redundant when the more general definitions were adopted later, and were removed during sleeping.

Similar but not identical results were obtained for the other four exercises. Those exercises that did not achieve 100% accuracy at the 100 examples sample point were completely correct for the true atoms but included some overly-general clauses that covered some false atoms. Revisions to correct these clauses would be made when a covered false atom is diagnosed due to the occurrence if an appropriate example in the input at a later time.

Time (secs)	Accuracy %	True Questions	False Questions
10	91.9	80	22
25	94.1	137	34
50	100.0	222	12
100	100.0	239	12
200	100.0	354	23
random	91.8	94	91

Figure 8.6: Varying the learning time

The accuracy point missing from the figure, for the third exercise, denotes a point where interpretation of the learnt theory by a third-party interpreter was very time-consuming or non-terminating and so the accuracy measure was difficult to obtain. The later results for that exercise indicate that MINERVA was able to recover from the difficulty at a later point in learning through the sleeping mechanism. In fact, the clause $revpath(X,X) \leftarrow path(X,Y), revpath(Y,Y)$ was removed from the program. Although the clause is true, it is easy to see why it leads to non-terminating interpretation when a correct definition for *path* has been acquired in the domain where cycles are present.

The results overall demonstrate MINERVA's ability to learn a concise representation of a theory of interdependent concepts presented as random examples, achieving good accuracy even with only a small number of examples.

8.6.3 Varying the learning time

In the previous experiment MINERVA was permitted up to 100 seconds (real time) to process each example presented. In this experiment we examine the effect of varying that time allowance: over a range of 10 to 200 seconds per example. The same example presentation is used for each exercise, with results reported at the point of 25 positive examples presented in each case. The 25 examples includes repeats and examples presented earlier as answers to questions. In other respects, the experimental conditions of the previous experiment are duplicated.

We also repeat the experiment once using a random time limit. In this case the time limit for each example is randomly selected from a uniform distribution between 0 and 200 seconds.

8.6.3.1 Results

The results are summarised in figure 8.6. The second column indicates the accuracy achieved for the three relations. The third column indicates the number of questions

Sleep Freq	Sleep Duration	No. Sleeps	Accuracy %	No. Clauses
5	200	21	100.0	11
20	200	5	100.0	11
50	200	2	100.0	11
100	200	1	100.0	11
20	50	5	100.0	11
20	100	5	100.0	11
20	200	5	100.0	11
20	300	5	100.0	11
20	500	5	100.0	11
–	–	0	100.0	17

Figure 8.7: Varying the sleep pattern

asked by MINERVA during the exercise that were answered true by SAMPLER and the fourth column similarly indicates questions answered false.

As we might expect, the results indicate that learning performance, measured as accuracy, improves with time available up to the time sufficient to learn the theory correctly. Question efficiency decreases as MINERVA spends spare time searching for increasingly less likely hypotheses. The result for the single random time exercise suggests that performance for random time limit learning is much poorer than the average time limit allowed, but many more exercises of this nature would be required for strong conclusions.

8.6.4 Varying the sleep pattern

Now we turn to examining the effect of the frequency and duration of sleep on the learning problem. In this experiment we reproduce the conditions of the first experiment on this domain to learn the graph relations *path*, *revpath* and *cycle*, but vary the sleep patterns permitted for MINERVA. Using only one example presentation of 25 positive examples (including repeats) we permit sleeping to occur at intervals ranging from 5 to 100 total examples and for times ranging up to maximums of 50 to 500 real time seconds. We compare these results with an exercise under the same conditions but permitting no sleep at all.

8.6.4.1 Results

The results are reported in figure 8.7. The table is grouped in three sections: the first shows the results for variations in the sleep frequency; the second for variations in sleep duration and the final section shows when no sleep is permitted. In addition

to accuracy, the table shows the number of clauses defining the three relations in the learnt program: this is significant because sleeping aims to remove redundancy in the program. The actual number of sleeps permitted in each exercise is also shown because it depends on the total number of examples presented rather than the number of positive examples alone.

The results indicate that for this particular experiment varying the sleep pattern makes almost no difference to the learnt program. Indeed, only one sleep is as good as many, whatever the duration. Although this pattern is frequently exhibited it is not necessarily maintained for other domains or even example presentations: experience demonstrates that sleeps become more important in highly recursive and many-concept domains. Its beneficial role in learning was demonstrated earlier in one exercise of the the first experiment in the graph domain.

8.6.5 Tolerating noise

Although MINERVA does not include any particular mechanisms designed for handling noise in examples and answers to questions, there is some ability to tolerate noise as a consequence of other design features. In this experiment we briefly observe MINERVA's performance in a noisy environment on the three-predicate graph problem as before. Examples and answers generated by SAMPLER are subject to mutation with a probability fixed for each exercise between 0 and 20%. If an example is selected for mutation its classification is reversed (positive-to-negative and negative-to-positive). If an answer to a question is selected for mutation it is complemented (true-to-false and false-to-true).

Accuracy is measured after 25 genuinely positive examples have been presented, irrespective of the classification given to them after mutation. The sample strategy and background knowledge of the earlier experiments are duplicated, with a time limit of 100 seconds imposed and sleeping permitted for 200 seconds after each 20 examples.

8.6.5.1 Results

The results are reported in figure 8.8. The second column of the figure represents accuracy of the learnt theory over the complete target, and the third column accuracy of the learnt theory on the true atoms of the target alone. We can see that overall accuracy decreases with increasing noise, but that in most cases the decrease in accuracy of a learnt theory is much less than the error rate in the environment. The decrease in accuracy is mostly due to under-generalization in learning as can be seen by the relatively poor accuracy on the true atoms.

Noise %	All Accuracy %	True Accuracy %
0	100.0	100.0
2	99.3	99.0
5	92.9	44.4
10	94.4	58.9
15	89.4	27.6
20	88.4	18.1

Figure 8.8: Tolerating noise

Strategy	Accuracy %					
	1	2	3	4	5	Av
Uniform-positive	95.6	97.4	100.0	89.5	96.8	95.9
Uniform-arity						96.9

Figure 8.9: Learning from positive examples

8.6.6 Learning from positive examples

Now we consider changing the sample strategy for the three-predicate graph problem. Because positive examples play the major role in prompting learning by MINERVA, in this experiment we use a positive-only sample strategy. For the most part, MINERVA finds the counter examples needed to avoid over-generalization during hypothesis testing.

We use five example presentations drawn from the positive only *uniform-positive* strategy, each selecting 25 positive examples. MINERVA is permitted 100 seconds for each example and sleeping for 200 seconds after each 20 examples. Because learning is terminated after 25 examples, this permits only one sleep in each exercise.

8.6.6.1 Results

Figure 8.9 gives the accuracy results for the five exercises together with their average. In the second row the figure shows the comparative average accuracy for the *uniform-arity* sample strategy under the same conditions (from figure 8.5). We can see that there is little difference in the accuracy results between a sample where positive examples are represented according to their proportional occurrence in the target (as in the unform-arity strategy) and another where only positive examples are presented (the uniform-positive strategy), provided that the number of positive examples in the sample is maintained.

Predicate	Accuracy %						
	empty	1	2	3	4	5	Av
arc	97.4	97.7	98.0	98.1	97.9	97.8	97.9
path	88.0	95.3	94.4	94.7	95.8	96.7	95.4
revpath	88.0	95.1	94.4	94.5	95.8	96.8	95.3
cycle	68.6	100.0	100.0	100.0	97.1	100.0	99.4
Overall	90.9	96.1	95.7	95.8	96.5	97.1	96.2

Figure 8.10: Confusing the background knowledge

8.6.7 Confusing the background knowledge

So far in the graph domain MINERVA has commenced learning with a complete and correct definition for *arc* initially provided. In this experiment we show that this is not necessary: although the problem becomes much harder it is still possible for MINERVA to learn definitions for the concepts.

In a sequence of five exercises we use the uniform-arity sample strategy to draw random examples from the target predicates *arc*, *path*, *revpath* and *cycle* in five different sequences. We permit 150 seconds per example and sleeping for 200 seconds after each 20 examples. We examine the learnt theory after 100 positive examples have been presented.

8.6.7.1 Results

The results in figure 8.10 show the accuracy for each predicate of each learnt theory in the five exercises as well as the the overall accuracy for each theory and the average accuracy for all five exercises. In the second column the accuracy for an empty definition of each predicate is given for comparison: these figures represent for each predicate the proportion of false atoms in all atoms of the predicate in the target.

It is notable that the accuracy for *arc* is particularly low. The inability to find a good definition for *arc* hinders the development of good definitions for the other predicates. The reason for the inability to find a good definition for *arc* is a consequence of two factors: the relative sparsity of positive examples of *arc* in the target (so that only four to eight positive examples occurred in the input for each exercise) and the lack of interesting generalizations of an atom of *arc* in terms of other predicates. This latter factor inhibits the ability of MINERVA to find its own examples as it tests generalizing hypotheses: experiments yield too many negative answers, directing the search towards highly specialized hypotheses.

Nevertheless, although the learnt programs are fairly complex in each case, the experiment demonstrates MINERVA's ability to learn a reasonably accurate theory even when

Predicate	Accuracy %						
	empty	1	2	3	4	5	Av
conn	81.3	97.6	100.0	100.0	97.6	92.1	97.5
bicycle	95.4	96.8	99.3	97.1	100.0	99.6	98.6
Overall	88.0	98.6	99.8	99.3	99.4	97.9	99.0

Figure 8.11: Learning five graph relations

fundamental background knowledge is presented by examples interspersed with those of dependent concepts, rather than *a priori* identified and supplied.

8.6.8 Extending the problem

In a final group of experiments in the graph relations domain we investigate learning two more concepts, *conn* and *bicycle*, on top of those already learned.

In each exercise of this group SAMPLER selects examples by the uniform-arity sample strategy. Accuracy results are reported at points where a fixed number of positive examples have been presented; repeated examples and examples previously known to MINERVA by answers to questions are included in the count, but negative examples are not.

For five exercises learning commences with a program of correct definitions for *arc*, *path*, *revpath*, and *cycle* as initial background knowledge. The initial program itself has previously been learned by MINERVA from an initial program defining *arc* and a presentation of fifty positive examples of the other predicates. It was indeed the first program learned in the graph domain experiments of section 8.6.2. SAMPLER selects examples of the predicates *conn* and *bicycle*, in a different random presentation for each exercise, stopping at 50 positive examples. MINERVA was permitted 200 seconds for each example and sleep for 300 seconds after each 20 examples.

8.6.8.1 Results

The accuracy results for each of the five exercises are presented in figure 8.11. The figure shows the accuracy for the *conn* and *bicycle* predicates separately, as well as the overall accuracy for the each theory of all five predicates, excluding *arc*. Average accuracy for the five exercises and the accuracy of the empty program is also shown.

These results can be compared with an exercise in which the five predicates are presented interleaved, commencing only with an initial ground unit clause definition for *arc*. In this case, at the point of 100 interleaved positive examples of *path*, *revpath*, *cycle*, *conn* and *bicycle*, MINERVA has observed the same total number of positive examples as in the exercises of figure 8.11, allowing for the 50 used in development of

Predicate	Accuracy %						
	empty	1	2	3	4	5	Av
conn	81.3	100.0	94.6	97.0	100.0	98.7	98.1

Figure 8.12: Learning a single extra graph relation

the initial background knowledge. But in this case we find the accuracy of *conn* to be 99.5%, *bicycle* to be 95.0% and the overall accuracy to be 98.6%. Repeating the exercise with a different presentation order, we find the accuracy of *conn* to be 97.1%, *bicycle* to be 100.0% and the overall accuracy to be 97.1%.

There is clearly high variability in these exercises, and many more exercises would be needed for strong conclusions, but we can observe that MINERVA is capable of learning a highly accurate program of multiple interdependent concepts from small numbers of examples, even when the examples of five different concepts are interspersed.

Although there is little difference in the accuracy, there is a notable difference in the form of the learned programs depending on the presentation order of predicates. The latter two exercises produced programs of 67 and 191 clauses respectively, while the former five exercises produced final programs of only 46 to 69 clauses, with an average of 57 clauses each. These include the 32 initial *arc* clauses.

8.6.8.2 A smaller extension

Finally, we consider learning only a single predicate, *conn*, commencing with the same initial program defining *arc*, *path*, *revpath*, and *cycle*. Because it is an easier problem (there is less background knowledge to search), we can reduce the time per example to 150 seconds and sleeping time to 200 seconds, leaving the sleeping frequency at every 20 examples.

After five different presentations of examples of *conn*, its accuracy is given in figure 8.12. Accuracy is measured after 25 examples — this being half of the number of examples for the exercises with *conn* and *bicycle* in figure 8.11. Because true atoms about *conn* are more common than true atoms about *bicycle* in the target, this actually represents a point of fewer positive examples of *conn* than in the earlier exercises (figure 8.11).

By comparison with the accuracy of *conn* in figure 8.11, it appears that learning *conn* alone, given appropriate background knowledge, is an easier problem for MINERVA than learning *conn* and *bicycle* together. Nevertheless, given the extra time for each example, MINERVA performs well on the more difficult task.

8.7 Summary

In this chapter MINERVA's learning performance has been demonstrated on several domains: English grammar, Eleusis, family relationships and graph relations.

In the earlier experiments, experimental conditions designed for other learners were nearly as possible replicated for MINERVA, enabling a demonstration of MINERVA's performance in a range of learning settings. These experiments show that MINERVA's performance compares well to that of batch and incremental learners; of revising and non-revising learners; of multiple and single-predicate learners and of learners with a range of background knowledge representations: from ground unit atoms through to normal clauses.

In the latter suite of experiments the MINERVA's capabilities were tested in a range of problems based on the graph relations domain. We found that although MINERVA is capable of learning interdependent concepts from randomly sequenced examples, learning performance is not independent of example order. Learning is improved when basic concepts are well understood before examples of dependent concepts are introduced. This is not surprising: it reflects a very common principle in human education. We found also that example efficiency increases but that question efficiency decreases as more time is made available for learning from each example.

We also experimented with a range of other parameters in the environment. We found that some sleep is beneficial for learning performance but that increasing sleep occurrence and duration indefinitely is not beneficial. The optimum minimum amount of sleep for a learning problem is likely to be domain-dependent.

Although MINERVA is not specifically designed for noisy environments, we found that a small error rate in the environment barely affects learning performance. We also demonstrated that positive examples have a much greater role than negative examples in learning by MINERVA.

In all the experiments of the chapter, SAMPLER proved to be a flexible tool for simulating a wide variety of environmental conditions for incremental learning evaluation.

9

Generalization by Absorption

9.1 Introduction

We have argued that MINERVA's learning strategy must be responsive to a single example, be founded on active experimentation, make use of background knowledge, and support incremental improvement. In this chapter we study absorption, a generalizing operator which offers these features.

In chapter 3 the model of generality known as generalized subsumption was introduced. It deals with the generality of definite clauses with respect to background knowledge represented as a definite program. The inverse-resolution operator, absorption, was introduced as a tool for generalizing definite clauses according to generalized subsumption.

In this chapter we explore the properties of absorption in depth. We present some new results for the completeness of absorption. We introduce mechanisms for reducing the non-determinacy inherent in absorption thus simplifying the structure of the search space. We extend absorption to enable generalization of normal clauses and we propose a model of generality for normal clauses according to which it generalizes.

This chapter places MINERVA's generalization procedure on a formal foundation, developing the particular absorption operator called *f-absorption* that is used in MINERVA. The practical implications of the analysis were described in section 5.3.2, where MINERVA's generalization procedure was outlined. The stepwise search through the generalization space of chapter 6 builds upon the theoretical structure mapped out here. Readers preferring to avoid a detailed view of MINERVA's theoretical foundations may skip to the next chapter.

9.2 Absorption

Previously we gave a loose description of absorption and showed how it could be interpreted as reversing a step of the deductive inference rule known as binary resolution. Here we give a more precise definition describing the form of absorption used in MINERVA.

First we define *absorb*, a particular instance of the more general absorption operator of Muggleton and Buntine [1988]. Although absorption is usually applied to a pair of definite clauses, we will find it convenient to use the notation to refer also to absorption of a definite clause or goal body with a definite clause. The absorption of a clause with a clause produces a set of clauses; the absorption of a goal body with a clause produces a set of goal bodies.

Definition 9.1 (Absorption of a goal, *absorb***)** *Let* $\leftarrow I$ *be a definite goal and* B *be a definite clause such that* I *and* B *share no common variables. Let* θ *be a substitution for the variables of* B *such that* $B_{\otimes}\theta \subseteq I$. *Then* $absorb(I, B, \theta)$, *the* absorption *of* I *with* B *using* θ, *is the set of clause bodies* G *such that*

$$G = (I - B_{\otimes}\theta) \cup \{B_{\odot}\theta\} \cup S$$

for some S *such that* $S \subseteq B_{\otimes}\theta$.

Also, $absorb(I, B)$, *the* absorption *of* I *with* B, *is the set of clause bodies* G *such that there exists a substitution,* θ *such that* $G \in absorb(I, B, \theta)$.

$\leftarrow I$ *is called the* input *goal and* B *is the* background *clause. The atoms in the set* S $(\{\} \subseteq S \subseteq (B_{\otimes}\theta))$ *are called* optional *and the substitution* θ *is called the* suitable *substitution.*

Definition 9.2 (Absorption of a clause, *absorb***)** *Let* I *and* B *be definite clauses with no common variables. Let* θ *be a substitution for the variables of* B *such that* $B_{\otimes}\theta \subseteq I$. *Then* $absorb(I, B, \theta)$, *the* absorption *of* I *with* B *using* θ, *is the set of clauses* H *such that* $H_{\odot} = I_{\odot}$ *and* $H_{\otimes} \in absorb(I_{\otimes}, B, \theta)$.

Also, $absorb(I, B)$, *the* absorption *of* I *with* B, *is the set of clauses* H *such that* $H_{\odot} = I_{\odot}$ *and* $H_{\otimes} \in absorb(I_{\otimes}, B)$.

In this role, I *is called the* input *clause and* B *is the* background *clause.*

Here we introduce a notational convention: sometimes we write $[S]$ to refer to an arbitrary subset of set S. This enables us to refer to an arbitrary clause $H \in absorb(I, B, \theta)$ as $H = I_{\odot} \leftarrow (I_{\otimes} - B_{\otimes}\theta) \cup \{B_{\odot}\theta\} \cup [B_{\otimes}\theta]$.

Example 9.1 (Absorption) Let I be the input clause

$$grandfather(X,Z) \leftarrow father(X,Y),\ \ father(Y,Z)$$

and B be the background clause

$$parent(U,V) \leftarrow father(U,V).$$

Then $absorb(I,B)$ is the set of clauses

$$grandfather(X,Z) \leftarrow parent(Y,Z),\ father(X,Y)$$
$$grandfather(X,Z) \leftarrow parent(Y,Z),\ father(X,Y),\ father(Y,Z)$$
$$grandfather(X,Z) \leftarrow parent(X,Y),\ father(Y,Z)$$
$$grandfather(X,Z) \leftarrow parent(X,Y),\ father(Y,Z),\ father(X,Y)$$

□

One particular member of the absorption of a given input clause with a given background clause using a given suitable substitution is distinguished: the least general [Muggleton 1991]. It is less general than the others because it is θ-subsumed by them. It is only equivalently general to the input clause with respect to any program which contains the background clause, so least general absorption does not strictly generalize — see my proof of a very similar result [Taylor 1993]. We can also distinguish the *most general* which θ-subsumes the others.

Definition 9.3 (Least general absorption, Most general absorption) *Let I and B be definite clauses such that I and B share no common variables and θ be a substitution for variables in B such that $B_{\otimes}\theta \subseteq I_{\otimes}$. Then the* least general absorption *is the clause $I_{\odot} \leftarrow I_{\otimes} \cup \{B_{\odot}\theta\}$. The* most general absorption *is the clause $I_{\odot} \leftarrow (I_{\otimes} - B_{\otimes}\theta) \cup \{B_{\odot}\theta\}$.*

Another distinguished application of absorption we call *unit absorption* — when the background clause is a unit clause. It is a particular case of least general absorption because there are no optional atoms. It can be applied to any input clause since the suitable subset condition holds trivially. It produces only one clause for each suitable substitution, and each of these is equivalently general to the input clause.

Any clause constructed by absorption may be generalized further by using it as the input clause to a successive application of absorption with another background clause. We call the linearly iterated absorption function $absorb^k$ and the corresponding closure function $absorb^*$. Variables in a background clause must be renamed as they are used so that no variables also occur in an earlier background clause or earlier iterated absorption products.

Definition 9.4 (Iterated absorption of a goal, $absorb^k$) *Let $\leftarrow G$ be a definite goal (called the* initial input goal*) and let P be a definite program. $absorb^0(G,P)$ is the set $\{G\}$. $absorb^k(G,P)$ for $k > 0$ is the set of clause bodies H such that there is a clause body $G' \in absorb^{k-1}(G,P)$ and a clause B that is a variant of a clause $B' \in P$ with new variables such that $H \in absorb(G',B)$.*

Definition 9.5 (Absorption closure of a goal, $absorb^*$) *Let* $\leftarrow G$ *be a definite goal and* P *be a definite program. Then* $absorb^*(G,P) = \{H | H \in absorb^k(G,P)$ *for some* $k \geq 0\}$

For brevity, assume that the functions *iterated absorption of a clause* ($absorb^k$) and *absorption closure of a clause* ($absorb^*$) are also defined for an initial input clause and a program. In each case the clauses in the set have heads that are just the same as the head of the initial input clause. Their bodies are each members of the set produced by the corresponding function on the body of the initial input clause.

9.2.1 Features of absorption

Use of background knowledge

Generalization by absorption can be seen as "justified" by background knowledge. Absorption generalizes a concept description to cover other instances which are "like" those already covered, where the likeness is determined by background knowledge. In the example (9.1), the relation *father* in the input clause is replaced by the relation *parent* in the generalization because the background clause describes *parent* as being a more general "feature" than *father*.

By contrast, truncation (generalization by θ-subsumption) assumes, in isolation of background knowledge, that some "feature" of a concept description is irrelevant and should be discarded. In the nomenclature of Michalski [1983], truncation implements the *turning constants to variables* and *dropping condition* rules whereas absorption implements the *climbing generalization tree* rule.

Soundness

Absorption is sound for generalizing according to generalized subsumption: for clauses I and B, every clause in $absorb(I,B)$ is more general than I by generalized subsumption with respect to any program containing B. Further, any clause in $absorb^*(I,P)$ is more general than I by generalized subsumption. Proofs of very similar results are given by Jung [1993] and by myself [Taylor 1993].

Incrementality

Absorption is well suited to an active, incremental, interruptible learner because it enables a stepwise investigation of the generalization hierarchy rooted at an initial input clause. Absorption may be applied to an input clause to generate an inductive hypothesis which may then be tested. If the test is successful, then the hypothesis may become the input clause for further generalization by absorption. If testing reveals that a particular generalization is false, then the generalization hierarchy rooted there may be pruned away because every clause more general is also false.

By iterated applications in this way the one operator can incrementally generate the generalization hierarchy.

Non-determinacy

The extent of the non-determinacy in its application is a source of difficulty with absorption. Consider, for example, computing the absorption closure from an initial input clause. At each step of expansion of the search hierarchy, for each input clause in the set, there may be a number of suitable background clauses in the program. For each of these, there may be multiple suitable substitutions. For any input clause, background clause, and suitable substitution there is a product generated for each subset of the instantiated atoms in the body of the background clause. Finally, in the more general form of definition 3.4, the number of inverse substitutions yielding distinct clauses may be large.

In order to contain the indeterminacy, learners that generalize by absorption usually employ a special instance of the general operator we have described. For example, CIGOL [Muggleton and Buntine 1988] restricts background clauses to unit clauses; FORTE [Richards and Mooney 1995] always chooses the optional literals to be the empty set; and MARVIN [Sammut and Banerji 1986] limits its use of unit absorption and inverse substitution. These restrictions are detrimental to the completeness of the operator.

Another approach, underlying the generalization strategy of ITOU [Rouveirol 1991b] and closely related to GOLEM [Muggleton and Feng 1990] uses iterated least general absorption, called *saturation*, followed by a final truncation step. Saturation initially computes a starting clause by exhaustive iterated least general absorption of the input clause — effectively computing the theory corresponding to the background program combined with the body of the input clause. The starting clause is only equivalently general to the input clause. It is then truncated, thus generalizing by θ-subsumption. The indeterminacy in a more general use of absorption is replaced by the indeterminacy of truncation. But θ-subsumption is a generality relation which does not take account of background-knowledge, so having compiled out the background knowledge the generalization step is knowledge-free. Usually it is desirable to prune the search for a suitable truncation generalization at this point by referring back to the background knowledge for detection of logically redundant literals.

Incompleteness

The procedure coupling least general absorption and truncation is claimed to be complete for generalized subsumption [Rouveirol 1991a, Rouveirol 1991b]. Jung [1993] gives a proof for a restricted class of programs. But unfortunately it gives up the most promising features of absorption as a generalizing operator. Background knowledge is used only to define the theory — the generality relationships between concepts captured in the program structure is not exploited. No inductive hypothesis at all can be determined until the starting clause is constructed; this construction may take a long

time if it terminates at all, and the starting clause itself may be huge, incorporating redundant and irrelevant literals. The search of the generalization hierarchy does not take advantage of the background knowledge to enable pruning and to give direction in the search.

Instead, MINERVA searches the generalization hierarchy by iterative absorption, avoiding truncation entirely. We shall see how this may be done efficiently by generating only the most interesting members of the absorption closure at each level, without giving up the completeness of the search.

This motivates a new completeness proof for absorption. In this approach we strike a compromise between using absorption deterministically and using it to incrementally generate successive candidate hypotheses with increasing generality. We find that we can remove some of the indeterminacy of absorption by exploiting its relation to SLD-resolution, while retaining its basic nature as an operator for generalization in increments guided by background knowledge.

We find that we do still need a final inverse substitution step for completeness, but we present an argument in favour of avoiding it anyway.

9.3 Completeness of Absorption

9.3.1 Preliminary definitions

In order to exploit the close relationship between absorption and SLD-resolution, we now treat definite clauses and definite goals as having multisets of atoms in their bodies rather than sets. Logic programming usually treats goal and clause bodies as sequences of atoms but the order of the atoms is irrelevant to the results used here. Later we return to the view in which clause bodies are treated as sets of atoms or literals. To enable translation between representations we introduce $set(C)$ to be a function mapping a multiset to the corresponding set or a multiset-based clause to a set-based clause. The notation $C_\otimes$ is reused to refer to the multiset comprised of the atoms in the body of the goal or clause, C. The symbol $\subseteq_M$ refers to the sub-multiset relation.

Now let us define an absorption operator on multiset-based input goals and background clauses to produce a multi-set based goal. It corresponds to most general absorption, having no optional atoms.

Definition 9.6 (Multiset absorption of a goal, $absorb_M$) *Let $\leftarrow G$ be the definite goal $\leftarrow G_1, \ldots, G_j, \ldots, G_k, \ldots, G_n$ and B be the definite clause $B_\odot \leftarrow B_1, \ldots, B_m$ such that G and B share no common variables. Then $absorb_M(G, B)$ is the set of goal bodies G' such that there is a substitution, θ for the variables in B such that $B_\otimes\theta \subseteq_M G$ (say $B_1\theta, \ldots, B_m\theta$ is identical to $G_j, \ldots, G_k$) and G' is $G_1, \ldots, G_{j-1}, G_{k+1}, \ldots, G_n, B_\odot\theta$.*

Definition 9.7 (Multiset absorption of a clause, $absorb_M$) *Let I and B be definite clauses which have no common variables. Then $absorb_M(I, B)$ is the set of clauses*

H such that $H_\odot$ is $I_\odot$ and $H_\otimes \in absorb_M(I_\otimes, B)$.

In this role, I is called the input *clause and B is the* background *clause.*

Similarly, the reader may assume the appropriate definitions for $absorb^k_M$ and $absorb^*_M$ on goals and clauses, by referring to the earlier definitions of $absorb^k$ and $absorb^*$, adding the subscript ($_M$) to occurrences of *absorb*, $absorb^k$ and $absorb^*$ there.

9.3.2 Inversion of SLD-resolution

The following theorem tells us that a single application of $absorb_M$ reverses a single derivation step in an SLD-derivation. It also says that the *symbols* (that is, constant, predicate and function symbols) in the input clause are preserved in the generalization when they do not occur in the background clause. This second result is obvious but later we shall see that its less obvious consequences enable us to reason about properties of absorption.

Theorem 9.1 (Absorption as inversion of SLD-resolution) *Let B be a definite clause and let $\leftarrow G'$ be a definite goal and let δ be a substitution such that B and G' share no common variables. Let $\leftarrow G$ be derived from $\leftarrow G'$ and B using mgu μ. Let B' be a variant of B that shares no variables with G or $G\delta$. Then $G'\mu\delta \in absorb_M(G\delta, B')$. Let p be a predicate symbol in G. Then p occurs in B or in G'. Let c be a constant or function symbol in G. Then c occurs in B or in G'.*

Proof. Let B be the clause $A \leftarrow B_1, \ldots, B_q$. Let $\leftarrow G'$ be the goal $\leftarrow A_1, \ldots, A_m, \ldots, A_k$ and A_m be the selected atom in G'. By definition 3.7, $G = (A_1, \ldots, A_{m-1}, B_1, \ldots, B_q, A_{m+1}, \ldots, A_k)\mu$ and $A_m\mu = A\mu$. So for any substitution δ, $G\delta = (A_1, \ldots, A_{m-1}, B_1, \ldots, B_q, A_{m+1}, \ldots, A_k)\mu\delta$. Let ρ be the renaming substitution so $B'\rho = B$ and $B\rho = B$.

Now consider $absorb_M(G\delta, B')$. G' and B share no variables and $B'_\otimes\rho\mu\delta = B_1, \ldots, B_q\mu\delta$ so we can see that $B'_\otimes\rho\mu\delta \subseteq_M G\delta$. By definition 9.6, $absorb_M(G\delta, B')$ includes $(A_1\mu\delta, \ldots, A_{m-1}\mu\delta, A_{m+1}\mu\delta, \ldots, A_k\mu\delta, A\rho\mu\delta) = (A_1, \ldots, A_k)\mu\delta$ which is $G'\mu\delta$.

Now each literal in G is an instance of a literal in B or G' so a predicate symbol p occurring in G also occurs in B or G'. Each constant or function symbol in G is in $(A_1, \ldots, A_{m-1}, B_1, \ldots, B_q, A_{m+1}, \ldots, A_k)$ or in the output of μ. If the former holds for c, then c occurs in B or G' because each of these literals occurs in B or G'. If the latter, because μ is an *mgu* for A_m and A, c occurs in A_m or in A. If c occurs in A_m then A occurs in G'. If c occurs in A then c occurs in B. □

Now we are ready to show how repeated applications of absorption step through an SLD-derivation in reverse, as illustrated in figure 9.1.

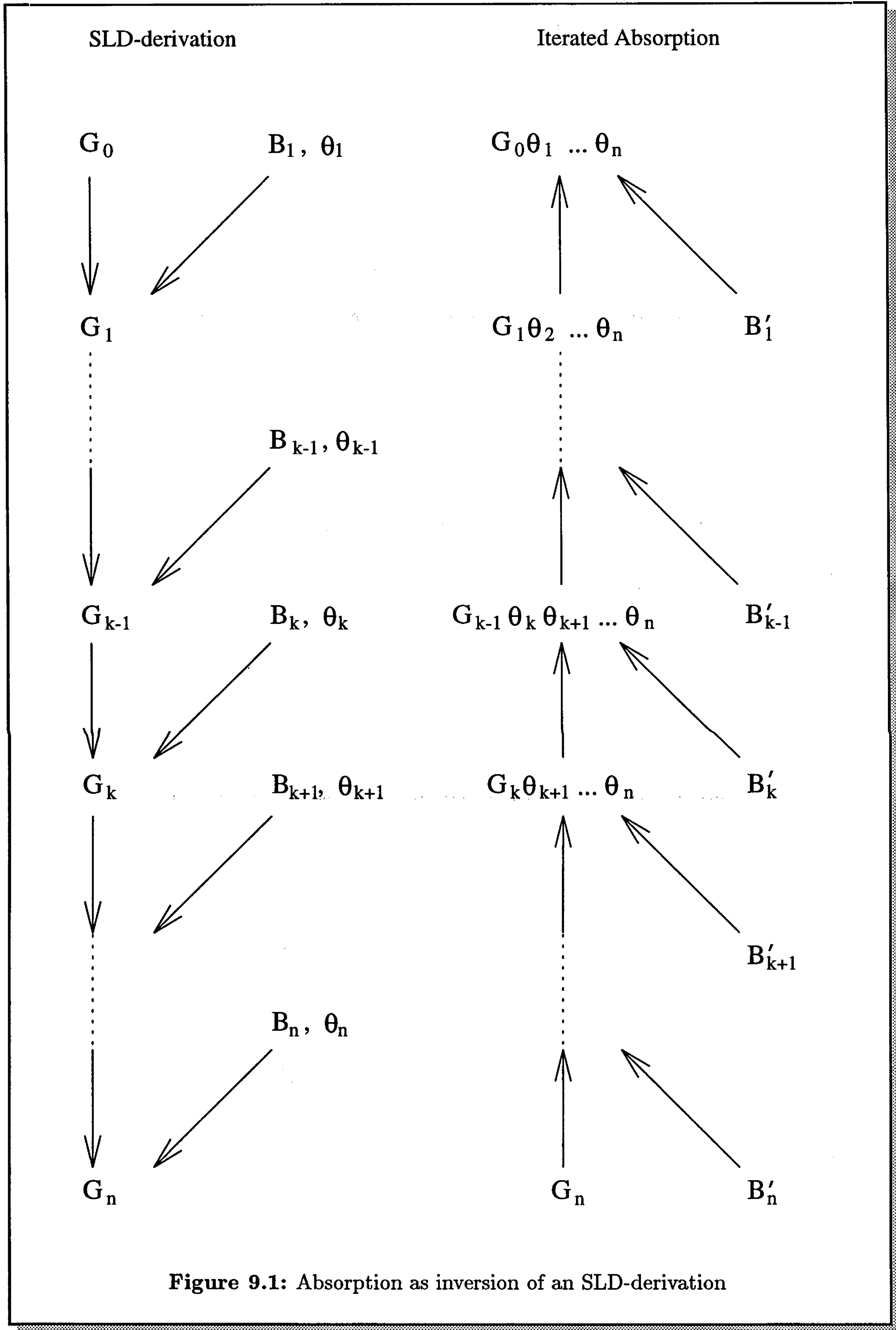


Figure 9.1: Absorption as inversion of an SLD-derivation

Lemma 9.1 (Reversing a derivation) *Let the sequences of definite goals* $\leftarrow G_0, \leftarrow G_1, \ldots, \leftarrow G_n$, *substitutions* $\theta_1, \theta_2, \ldots, \theta_n$, *and definite clauses* $B_1, \ldots, B_n$ *be an SLD-derivation. Let* δ *be a substitution. Let* θ_{n+1} *be the identity substitution. Then for each* G_k *in the derivation* $(0 \leq k \leq n)$, $G_k\theta_{k+1}\ldots\theta_n\theta_{n+1}\delta \in absorb_M^{n-k}(G_n\delta, \{B_1, \ldots, B_n\})$. *Furthermore, any symbol in* G_n *also occurs in some* B_i $(k < i \leq n)$ *or in* G_k.

Proof. The proof is by induction starting from the end of the derivation (when $k = n$) and working backwards. For the base case, set $k = n$ and the result follows directly from the definition of $absorb_M^0$.

Assume the lemma holds for k where $k > 0$ and $k \leq n$ and show it holds for $k - 1$. G_k is derived from G_{k-1} and B_k using *mgu* θ_k. Let B'_k be a variant of B_k with new variables, not occurring in any $G_i\theta_{i+1}\ldots\theta_{n+1}\delta$, any B_i, or any G_i for $i \geq k$. Then by theorem 9.1 $G_{k-1}\theta_k\theta_{k+1}\ldots\theta_{n+1}\delta \in absorb_M(G_k\theta_{k+1}\ldots\theta_{n+1}\delta, B'_k)$.

By the induction hypothesis $G_k\theta_{k+1}\ldots\theta_{n+1}\delta \in absorb_M^{n-k}(G_n\delta, \{B_1, \ldots, B_n\})$. Also, $B_k \in \{B_1, \ldots, B_n\}$. So, by applying $absorb_M$ to each element of $absorb_M^{n-k}(G_n\delta, \{B_1, \ldots, B_n\})$ to get $absorb_M^{n-(k-1)}(G_n\delta, \{B_1, \ldots, B_n\})$ we have that $G_{k-1}\theta_k\theta_{k+1}\ldots\theta_{n+1}\delta \in absorb_M^{n-(k-1)}(G_n\delta, \{B_1, \ldots, B_n\})$.

Let s be a symbol in G_n. By the induction hypothesis either s occurs in some B_i $(i > k)$ or in G_k. If the former then s occurs in some B_i for $i > k - 1$. If the latter then s occurs in G_k but by theorem 9.1 s also occurs in B_{k-1} or in G_{k-1}. That is s occurs in G_{k-1} or in some B_i for $i > k - 1$. □

This enables us to establish that for any goal in a derivation, an instance of it may be recovered by iteratively and linearly applying absorption to a later goal in the derivation. When G_k is a goal in an SLD-derivation for $P \cup \{G_0\}$ we say that there is a derivation of G_0 *via* G_k.

Lemma 9.2 *Let there be an SLD-derivation for* $P \cup \{\leftarrow G\}$ *via* $\{\leftarrow G'\}$. *Then there exists a substitution* θ *such that for any substitution* δ, $G\theta\delta \in absorb_M^*(G'\delta, P)$. *Furthermore any symbol occurring in* G' *also occurs in either* G *or* P.

Proof. The result follows immediately from lemma 9.1 by definition of $absorb_M^*$. □

We are nearly ready to identify a starting point, an initial input goal, from which a derivation may be followed backwards using absorption. Because the body of the initial input goal for generalization might be unsatisfiable in the program, we need to separate the part of the refutation which matches atoms of the initial goal, and the clauses that are needed to do it, from the remainder of the refutation and program. Using Lloyd's preliminary switching lemma, the succeeding lemma cuts an SLD-refutation into two parts — a derivation which arrives at a goal without using certain unit clauses of the program and a refutation of that goal using the unit clauses alone.

For notational convenience henceforth, when G is a set of atoms, the set of unit clauses $\{(\leftarrow A) | A \in G\}$ is written simply as G.

Lemma 9.3 (Switching lemma) *Let P be a definite program and G be a definite goal. Suppose that $P \cup \{G_0\}$ has an SLD-refutation $G_0, G_1, \ldots, G_{q-1}, G_q, G_{q+1}, \ldots, G_n$ with input clauses $C_1, \ldots, C_n$ and mgus $\theta_1, \ldots \theta_n$. Suppose that G_{q-1} is $\leftarrow A_1, \ldots, A_{i-1}, A_i, \ldots, A_{j-1}, A_j, \ldots, A_k$ and G_q is $\leftarrow (A_1, \ldots, A_{i-1}, C_{q\otimes}, \ldots, A_{j-1}, A_j, \ldots, A_k)\theta_q$ and G_{q+1} is $\leftarrow (A_1, \ldots, A_{i-1}, C_{q\otimes}, \ldots, A_{j-1}, C_{q+1\otimes}, \ldots, A_k)\theta_q\theta_{q+1}$. Then there exists an SLD-refutation of $P \cup \{G_0\}$ in which A_j is selected in G_{q-1} instead of A_i and A_i is selected in G_q instead of A_j. Furthermore, if σ is the computed answer for $P \cup \{G_0\}$ from the given refutation and σ' is the computed answer from the new refutation then $G_0\sigma$ is a variant of $G_0\sigma'$.* [Lloyd 1987b, lemma 9.1]

Lemma 9.4 (Splitting a refutation) *Let U be a set of ground unit clauses and P be a definite program. Let $\leftarrow G$ be a goal. If there is a refutation of $P \cup U \cup \{\leftarrow G\}$ then there is a derivation for $P \cup \{\leftarrow G\}$ terminating in a goal $\leftarrow G'$ and a refutation of $U \cup \{\leftarrow G'\}$. Furthermore there is a substitution γ such that $set(G'\gamma) \subseteq U$.*

Proof. Let the refutation of $P \cup U \cup \{\leftarrow G\}$ be the sequences of goals $\leftarrow G_0, \leftarrow G_1, \ldots, \leftarrow G_n$, substitutions $\theta_1, \theta_2, \ldots, \theta_n$, and clauses $B_1, \ldots, B_n$.

Firstly consider the cases that either no clause from U occurs in $B_1, \ldots, B_n$ or every one of $B_1, \ldots, B_n$ is a clause from U. Then the lemma is trivially satisfied.

Now assume that there are m $(0 < m < n)$ occurrences of a variant of a clause from U in $B_1, \ldots, B_n$. Let j be the least of the indices $1, \ldots, n$ such that $B_j \in U$. Let A_j be the literal in $\leftarrow G_{j-1}$ which is selected and unifies with B_j. Now repeatedly apply the switching lemma (9.3) to delay the selection of A_j in the refutation until there is a goal with no other literal. B_j is unit so the derivation step when A_j is selected is now the last step of the new refutation.

Repeat this transformation $m - 1$ times for each step of the refutation that a clause from U is used. That is, successively delay the selection of atoms which are resolved with a clause in U, moving each to the end of the current refutation. Let this new refutation be the sequences of goals $\leftarrow G'_0, \leftarrow G'_1, \ldots, \leftarrow G'_n$, substitutions $\theta'_1, \ldots, \theta'_n$, and clauses $B'_1, \ldots, B'_n$. G'_0 is G_0.

Every clause in $B'_1, \ldots, B'_{n-m}$ is a variant of a clause from P. Every clause in $B'_{n-m+1}, \ldots, B'_n$ is a ground clause from U. The goal $\leftarrow G'_{n-m}$ contains only atoms which unify with ground atoms in U. The sub-sequences of the derivation commencing with $\leftarrow G'_{n-m}$, θ'_{n-m+1} and B'_{n-m+1} comprise a refutation. The computed answer for this refutation (definition 3.11) is the composition $\theta'_{n-m+1} \ldots \theta'_n$ which grounds variables in G'_{n-m} because each constituent substitution is an *mgu* with a ground atom. Therefore for each atom A such that A occurs in $G'_{n-m}\theta'_{n-m+1} \ldots \theta'_n$, it must be that $(A \leftarrow) \in U$. □

Having established the relationship between iterated absorption of goals and SLD-derivation, the completeness result for SLD-resolution may be used to establish a completeness result for the absorption closure of a goal. The SLD-resolution completeness result attributed to Hill is reproduced first.

Theorem 9.2 (Completeness of SLD-resolution) *Let P be a definite program and G a definite goal. Suppose that $P \cup G$ is unsatisfiable. Then there exists an SLD-refutation of $P \cup \{G\}$.* [Lloyd 1987b, theorem 8.4]

Theorem 9.3 (Completeness of multiset absorption) *Let P be a definite program, U a set of ground unit clauses $\leftarrow G$ a definite goal and δ a substitution. If $P \cup U \models \exists G$ then there are substitutions α and ϵ and a definite goal $\leftarrow G'$ such that $set(G'\epsilon) \subseteq U$ and $G\alpha\delta \in absorb^*_M(G'\delta, P)$. Furthermore any symbol occurring in G' also occurs in G or P.*

Proof. By the completeness of SLD-resolution (theorem 9.2) if $P \cup U \models \exists G$ (that is, if $P \cup U \cup \{\leftarrow G\}$ is unsatisfiable), then there exists an SLD-refutation of $P \cup U \cup \{\leftarrow G\}$. By lemma 9.4 there is a derivation for $P \cup \leftarrow G$ terminating in a goal $\leftarrow G'$ and a refutation for $U \cup \{\leftarrow G'\}$ and a substitution ϵ such that $set(G'\epsilon) \subseteq U$. By lemma 9.2 there is a substitution α such that $G\alpha\delta \in absorb^*_M(G'\delta, P)$ and any symbol in G' also occurs in G or P. □

Using these results we can establish a result for the completeness of $absorb_M$ for generalizing with respect to generalized subsumption. The following technical lemma is necessary for the formulation of the completeness result.

Lemma 9.5 *Let I and I' be sets of literals. Let θ be a substitution mapping variables to distinct constants which do not occur in I' nor in I. If there is a substitution ϵ such that $I'\epsilon \subseteq I\theta$ then there is a substitution δ such that $I'\delta \subseteq I$.*

Proof. Let J be the subset of I such that $I'\epsilon = J\theta$. Let $\theta' \subseteq \theta$ such that for each $v_i/t_i \in \theta'$, v_i occurs in J. Then $J\theta = J\theta'$. Let θ' be $\{v_1/t_1, \ldots, v_n/t_n\}$. Then the t_is are distinct constants and no t_i occurs in I or I' or J. Let ϵ' be the subset of ϵ such that for each $u_i/s_i \in \epsilon'$, u_i occurs in I'. Then $I'\epsilon = I'\epsilon'$. Some of the output terms of ϵ' may have some of $\{t_1, \ldots, t_n\}$ occurring in them. Define substitution δ as follows. For each $u_i/s_i \in \epsilon'$, let $u_i/s'_i \in \delta$ where s'_i is constructed from s_i by replacing each occurrence of t_j in s_i by v_j in s'_i for each $v_j/t_j \in \theta'$. Now no variables in $\{v_1, \ldots, v_n\}$ occur in I' or $I'\epsilon'$ so $I'\delta\theta' = I'\epsilon'$. Further, no term in $\{t_1, \ldots, t_n\}$ occurs in I' or J (by assumption) or in $I'\delta$ by the definition of δ.

Now we have that $I'\delta\theta' = J\theta'$. Consider v_1/t_1 in θ'. Let θ'' be $\{v_2/t_2, \ldots, v_n/t_n\}$. v_1 occurs in no t_i so $\theta' = \theta''\{v_1/t_1\}$. v_1 occurs in J by definition of θ'. Assume v_1 does not occur in $I'\delta$. Then t_1 does not occur in $I'\delta\theta'$ so t_1 does not occur in $J\theta'$ so v_1 does

not occur in J. So v_1 occurs in both $J\theta''$ and $I'\delta\theta''$ and because t_1 is a constant which occurs in neither and $I'\delta\theta''\{v_1/t_1\} = J\theta''\{v_1/t_1\}$ it must be that $I'\delta\theta'' = J\theta''$.

Continue successively removing each $v_i/t_i (2 \leq i \leq n)$ in this manner and eventually we have that $I'\delta = J$. Further $J \subseteq I$, so $I'\delta \subseteq I$. □

Now we show the completeness of absorption for definite clauses. It tells us that iterated absorption operating on subsets of the body of the input clause is complete for generalization, provided that we allow for an inverse substitution to be applied as a final step. This theorem is a major result of the chapter.

Theorem 9.4 (Completeness of multiset absorption for definite clauses) *Let H and I be definite clauses with distinct variables and P a definite program such that $H \succeq_P I$. Then there are substitutions σ and γ and a goal $\leftarrow I'$ such that $H_{\odot}\sigma$ is identical to $I_{\odot}$ and $set(I') \subseteq set(I_{\otimes})$ and $H_{\otimes}\sigma\gamma \in absorb^*_M(I', P)$. Furthermore, each predicate, constant or function symbol in I' also occurs in P or in $H_{\otimes}\sigma$.*

Proof. By the testing theorem (3.1) if $H \succeq_P I$ then there is a substitution σ such that $H_{\odot}\sigma$ is identical to $I_{\odot}$ and a substitution θ which distinctly grounds the variables in I using new constants not occurring in H, I or P such that $P \cup I_{\otimes}\theta \models \exists(H_{\otimes}\sigma\theta)$. Weakening this, $P \cup I_{\otimes}\theta \models \exists(H_{\otimes}\sigma)$.

Now $set(I_{\otimes}\theta)$ is a set of ground unit clauses so by the completeness of absorption for goals (theorem 9.3) there are substitutions α and ϵ and a goal $\leftarrow G'$ such that $set(G'\epsilon) \subseteq set(I_{\otimes}\theta)$ and for any substitution β, $H_{\otimes}\sigma\alpha\beta \in absorb^*_M(G'\beta, P)$ and the symbols in G' occur in $H_{\otimes}\sigma$ or P. In particular no constant in the output of θ occurs in G' because if one did it would also occur in P or $H_{\otimes}\sigma$ which would contradict the construction of θ. Therefore lemma 9.5 applies and there is a substitution δ such that $set(G'\delta) \subseteq set(I_{\otimes})$. Letting β be δ, $H_{\otimes}\sigma\alpha\delta \in absorb^*_M(G'\delta, P)$. Let I' be $G'\delta$ (so $set(I') \subseteq set(I_{\otimes})$) and let γ be $\alpha\delta$. Then $H_{\otimes}\sigma\gamma \in absorb^*_M(I', P)$. Because the symbols in G' occur in $H_{\otimes}\sigma$ or P, so do the symbols of I'. □

9.3.3 Completeness of absorption for definite clauses

Our treatment of clause bodies as multisets of literals is a source of redundancy in our generalization hierarchy. When a literal is duplicated in a clause body, the clause is equivalently general to the clause without the duplicated literal. This is easily seen with reference to the definition of generalized subsumption. More commonly in ILP clause bodies are regarded as sets of literals. We have used the multiset approach in order to draw directly on the results of logic programming, but now we modify our approach and use clauses, goals and the *absorb* function we defined initially. Note that absorption acting on an input clause may now generate a set of more general clauses for each suitable substitution.

To show that the completeness of absorption also holds for clauses we show that for every clause generated by $absorb_M$, the corresponding set-based clause is generated by *absorb*, and so the completeness of the former carries to the latter.

Theorem 9.5 (Relating multiset absorption to absorption) *Let B be a clause $\leftarrow G$ and $\leftarrow I$ goals such that $G \in absorb_M(I, B)$. Then $set(G) \in absorb(set(I), set(B))$. Furthermore the symbols occurring in I also occur in B or G.*

Proof. Let $\leftarrow G$ be a goal such that $G \in absorb_M(I, B)$ and let θ be the substitution used to generate G. Let B be a clause such that $B\theta$ is the clause $A\theta \leftarrow \mathcal{B}_1\theta, \ldots, \mathcal{B}_n\theta$ where $n \geq 0$ and A is an atom and each $\mathcal{B}_i\theta$ $(i = 1, \ldots, n)$ is a multiset of identical atoms $B_i\theta$ and the atoms of $\mathcal{B}_i\theta$ and $\mathcal{B}_j\theta$ are distinct when $i \neq j$.

Because $\mathcal{B}_1, \ldots, \mathcal{B}_n\theta \subseteq_M I$, I can be represented as $\mathcal{B}_1\theta, \ldots \mathcal{B}_n\theta, \mathcal{I}_1, \ldots \mathcal{I}_m$ where $m \geq 0$ and each $\mathcal{I}_i (i = 1, \ldots, m)$ is a multiset of identical atoms I_i and the atoms of $\mathcal{I}_i$ and $\mathcal{I}_j$ are distinct when $i \neq j$ but the atoms of any $\mathcal{I}_i$ are not necessarily distinct from the atoms of some $\mathcal{B}_j\theta$. Assume that each of $\mathcal{B}_1\theta, \ldots, \mathcal{B}_k\theta$ contain identical atoms respectively to each of $\mathcal{I}_1, \ldots, \mathcal{I}_k$ $(0 \leq k \leq m$ and $k \leq n)$ and elsewhere each $\mathcal{B}_i\theta$ is distinct from every $\mathcal{I}_j$. Then $\{B_1\theta, \ldots, B_k\theta\}$ is identical to $\{I_1, \ldots, I_k\}$ and, by the definition of $absorb_M(I, B)$, G is $\mathcal{I}_1, \ldots, \mathcal{I}_m, A\theta$ and $set(G)$ is $\{I_1, \ldots, I_m\} \cup \{A\theta\}$.

Now $set(I)$ is $\{B_1\theta, \ldots, B_n\theta, I_{k+1}, \ldots, I_m\}$ and $set(B)_{\otimes}\theta$ is $\{B_1\theta, \ldots, B_n\theta\}$ so $set(B)_{\otimes}\theta \subseteq set(I)$. Therefore each member of $absorb(set(I), set(B), \theta)$ is $(\{B_1\theta, \ldots, B_n\theta, I_{k+1}, \ldots, I_m\} - \{B_1\theta, \ldots, B_n\theta\}) \cup \{A\theta\} \cup S$ for some $S \subseteq \{B_1\theta, \ldots, B_n\theta\}$. Let S, the optional literals, be $\{B_1\theta, \ldots B_k\theta\}$. Then we see that there is a $G' \in absorb(I, B)$ such that G' is $\{B_1\theta, \ldots, B_k\theta, I_{k+1}, \ldots, I_m\} \cup \{A\theta\}$. Recalling that $\{B_1\theta, \ldots, B_k\theta\}$ is identical to $\{I_1, \ldots, I_k\}$, G' is $\{I_1, \ldots, I_k, I_{k+1}, \ldots, I_m\} \cup \{A\theta\}$ which is $set(G)$.

The argument in the proof of theorem 9.1 may be used here also to establish that the symbols occurring in I also occur in B or G. □

We have shown that absorption on goals computes at least the same generalizations as multiset absorption on goals, and that the multiset absorption closure for clauses is complete for generalized subsumption up to an inverse substitution (theorem 9.4). The completeness theorem for the absorption closure for definite clauses follows by a simple inductive argument, omitted for brevity.

Theorem 9.6 (Completeness of absorption for definite clauses) *Let H and I be definite clauses with distinct variables and P a definite program such that $H \succeq_P I$. Then there are substitutions σ and γ and a goal $\leftarrow I'$ such that $H_{\odot}\sigma$ is identical to $I_{\odot}$ and $I' \subseteq I_{\otimes}$ and $H_{\otimes}\sigma\gamma \in absorb^*(I', P)$. Furthermore, each predicate, constant or function symbol in I' also occurs in P or in $H_{\otimes}\sigma$.*

This implementation of absorption as an operation on sets of atoms offers several advantages. We no longer represent duplicate antecedents in clauses of the absorption

hierarchy. Using a set-based suitability test we can reduce the number of applications of absorption required to generate a particular generalization when the suitable instance of the background clause contains duplicate atoms — we no longer need to apply absorption a sufficient number of times to an input clause to duplicate the atom there. We no longer need to consider every multiset comprised of atoms of body of the initial clause as roots of alternative absorption hierarchies. Now we need consider only each subset of the body of the initial clause.

The most significant disadvantage of the set-based approach is that absorption of an input clause with a background clause given a particular substitution yields a number of alternative generalizations, one for each combination of optional atoms. Each of these should be considered as candidate hypotheses, but they can be organised into a sub-hierarchy of generalizations using most-general absorption (definition 9.3) and restriction (definition 3.5).

9.4 Eliminating Redundancy

The generalization search space defined by the absorption closure of an initial input clause includes many equivalently general clauses. By avoiding generating and exploring multiple clauses of the same equivalence class we can reduce the size of the search space. Consequently in this section we aim to reduce it to one which is complete in the sense that for every clause more general than the input clause an equivalently general clause is represented in the search space. But we take care to note that although some clauses may be redundant as inductive hypotheses, this does not imply that the search space defined by the absorption closure may be pruned at those nodes. It is often the case that such clauses must remain as input clauses for further applications of absorption in order to generate other non-redundant hypotheses.

9.4.1 Connex clauses

Atoms in a clause body which are not related to the other atoms by a path of shared variables are a source of generalization redundancy. Such atoms may be discarded from the clause resulting in an equivalently general but smaller clause.

Definition 9.8 (Connex, Connected) *Let C be a definite clause. Then an atom $A \in C_{\otimes}$ is directly connected if a variable occurring in A also occurs in $C_{\odot}$. A is* connected *if A is directly connected or if a variable occurring in A also occurs in a distinct atom $B \in C_{\otimes}$ and B is connected. connex(C) is the set of atoms $A \in C_{\otimes}$ such that A is connected. If C is identical to connex(C) then C is* connex.

Example 9.2 (Connex) This clause is not connex: $p(x, y) \leftarrow q(y), t(z, a)$. This clause is connex: $p(x, y) \leftarrow q(y), r(y, z), s(z), t(x, a)$. □

Every atom in a connex clause is connected to the head of the clause by a path of atoms, each sharing a variable with its successor in the path. Non-connex clauses are another source of generalization redundancy because for every non-connex clause in the absorption closure there is an equivalently general connex clause also in the absorption closure. The size of the generalization search space may therefore be reduced by removing non-connex clauses without compromising completeness.

Rouveirol [1991b] claims without proof that a non-connex clause has the same least Herbrand model as the corresponding connex clause formed by deleting the disconnected literals. Instead, in theorem 9.7 we show that a non-connex clause is equivalently general to the corresponding connex clause. The generality is defined by generalized subsumption with respect to a program — but not just any program.

We express the relationship in the circumstances that the non-connex clause is constructed by iterated absorption of an initial input clause with the program. We require that there is an instance of the initial clause body which is a logical consequence of the program. This is quite reasonable — it amounts to assuming that the initial input clause does cover at least one atom and is not just trivial.

Lemma 9.6 (Absorption existence) *Let* $\leftarrow C$ *and* $\leftarrow I$ *be definite goals and* P *be a definite program such that* $C \in absorb^*(I, P)$. *Let* T *be any Herbrand interpretation in which* P *is true and* $\exists I$ *is true. Then* $\exists C$ *is true in* T.

Proof. The proof is by induction on iterations of *absorb*. We show that for each $k \geq 0$, assuming $I_k \in absorb^k(I_0, P)$ and I_k is true in T implies that for any I_{k+1} such that $I_{k+1} \in absorb^{k+1}(I_0, P)$, $\exists I_{k+1}$ is true in T. The result then follows from the definition of $absorb^*(I_0, P)$.

For the base case, let I_0 be I, then $\exists I_0$ is true in T by assumption. For the iterative case, consider some $I_{k+1} \in absorb^{k+1}(I_0, P)$. Then there is an $I_k \in absorb^k(I_0, P)$ and a substitution θ and a clause $B \in P$ such that I_{k+1} is $(I_k - B_{\otimes}\theta) \cup \{B_{\odot}\theta\} \cup S$ for some $S \subseteq B_{\otimes}\theta$ and $B_{\otimes}\theta \subseteq I_k$. By the inductive assumption $\exists I_k$ is true in T. and so there is a substitution γ such that $I_k\gamma$ is ground and true.

Consider first when S is maximal. That is let $I' = I_k \cup \{B_{\odot}\theta\}$. Now $B_{\otimes}\theta\gamma$ is ground and true in T because it is a subset of $I_k\gamma$. Now $\forall(B_{\odot} \leftarrow B_{\otimes})$ is true in T because B is a definite clause in P and P is true in T. So $\forall(B_{\odot}\theta\gamma)$ is true in T and $\forall(\{B_{\odot}\theta\gamma\} \cup I_k\gamma)$ is true in T. Therefore $\exists(I_k \cup \{B_{\odot}\theta\})$ is true in T, that is, $\exists I'$ is true in T.

Relaxing the choice of S, for every possibility for I_{k+1} it holds that $I_{k+1} \subseteq I'$ so $\exists I_{k+1}$ is also true in T. □

Theorem 9.7 (Equivalence of non-connex clauses) *Let* P *be a definite program and* I *a definite clause such that* $P \models \exists I_{\otimes}$. *Let* C *be a definite clause such that* $C \in absorb^*(I, P)$. *Then* C *is equivalent by generalized subsumption with respect to* P *to* $C_{\odot} \leftarrow connex(C_{\otimes})$.

Proof. Let C' be the clause $C_{\odot} \leftarrow connex(C_{\otimes})$. Let A be an atom covered by C in an interpretation T. Then, referring to the definition of cover (3.2), C' also covers A because C and C' share the same head and $C'_{\otimes} \subseteq C_{\otimes}$. Therefore $C' \succeq_P C$.

Now let A be an atom covered by C' in interpretation T where P is true in T. Then there is a θ such that $C'_{\odot}\theta$ is identical to A and so $C_{\odot}\theta$ is identical to A too. Futhermore, $\exists(C'_{\otimes}\theta)$ is true in T. Now $C_{\otimes}$ may be partitioned into $C'_{\otimes}$ and R where these two sets share no common variables by definition of C'. By lemma 9.6 $\exists C_{\otimes}$ is true in T. Therefore $\exists C'_{\otimes}$ is true in T and $\exists R$ is true in T. Because $\exists(C'_{\otimes}\theta)$ is true in T and the input of θ does not include variables in R, $\exists(C_{\otimes}\theta)$ is true in T. Therefore C covers A. Therefore $C \succeq_P C'$.

We have shown $C' \succeq_P C$ and $C \succeq_P C'$ so they are equivalent with respect to P. □

Because of the completeness result, the connex clause corresponding to a non-connex clause in *absorb** is also in *absorb**, so non-connex clauses need not be considered as candidate hypotheses. Unfortunately, we cannot prune branches of the search space rooted in a non-connex clause because a generalization of a non-connex clause may be connex.

9.4.2 Duplicate clauses

Typically there are a number of alternative paths in the absorption closure (that is, alternative choices of background clauses at each step) yielding identical clauses as generalizations. Taking care in the exploration of the search space to avoid re-working duplicates can improve the efficiency of the search.

We can do this by maintaining a record of clauses as they are generated. Whenever a new clause is generated by absorption, if the clause already appears in the record then there is another path through the hierarchy to the same clause and so further iterated absorption of the present clause is unnecessary.

9.4.3 Least general absorption

Each clause which is the result of least general absorption of an input clause is equivalently general to the input clause. Therefore clauses generated by least general absorption, including those generated by unit absorption, need not be considered as inductive hypotheses. Unfortunately though we cannot exclude the generalization hierarchy rooted there from the search space for generalizations because the clause may be needed to act as the input clause for a later absorption step.

9.5 Reducing the Search Space Further

Shortly we will show how the search space may be simplified futher by changing the representation language of programs and clauses. Meanwhile there are two remaining issues of the completeness theorem that may be addressed.

9.5.1 Free variables in a background clause

When a background clause has variables in the head that do not also occur in the body, the suitable substitution is not well constrained by the subset condition of absorption. A suitable substitution may replace such a free variable in the head of the background clause by any other variable occurring elsewhere in the input clause or by any constant symbol or deeper term. Indeed, when the language of the program and input clause includes function symbols, the number of alternative suitable substitutions, each yielding different generalizations, is unbounded. Although each of these generalizations is equivalent in generality, all may be necessary to act as input clauses for later absorption steps, yielding non-equivalent generalizations.

A solution to this problem is to limit the language of programs. When each clause of the background program is *allowed* then there are no such free variables in the head of a clause, so suitable substitutions for absorption are constrained by the subset property on atoms in the body of the background clause.

Furthermore, the absorption of a non-allowed input clause with an allowed background clause cannot yield an allowed clause because variables in any atom introduced to the input clause must also occur elsewhere in the body of the clause. Therefore when only allowed generalizations are sought, the generalization search space beyond a non-allowed clause may be pruned away.

9.5.2 Choosing the root

Now let us describe absorption in a less abstract way to understand the effect of some remaining choices.

Let us characterise the predicate symbol of each atom in the body of a clause to be generalized as a *feature* of the concept members covered by the clause. We can regard each step of non-unit absorption as generalizing some features — replacing a set of features by a single more general one. The replacement feature is more general in the sense that it implies the features it replaces — like a movement towards the root of a feature hierarchy defined by background knowledge, called the *climbing generalization tree rule* by Michalski [1983]. Notably a feature of the input clause is only removed by absorption when it is replaced by another more general feature; features are dropped from a generalization only when sanctioned by background knowledge. On the other hand, unit absorption does not generalize but adds a new and possibly irrelevant feature to the clause, so that the feature is available for further generalization.

Now assume that a non-unit clause for which generalizations are sought is an example for learning. The completeness theorem (9.6) describes a set of absorption closures — one rooted at each subset of the body of the example. The theorem also says that in every generalization of each closure, every feature of the root is either retained or replaced by a more general feature. That is, the only way to treat a feature of the example as entirely irrelevant is to drop it from the body before the first step of absorption. But if we prefer that every feature of the example remains in a generalization unless its removal is sanctioned by background knowledge, then this is exactly what we want. In this case, choosing to search only the absorption closure which is rooted at the full body of the example and ignoring those rooted at its subsets reduces and simplifies the search space while elegantly implementing a kind of relevance assumption: assuming that every feature of the example is relevant.

9.6 Changing the Representation

The completeness result of theorem 9.6 is weaker than we would like: although it allows us to search for generalizations of an input clause in an hierarchy structured by absorption, for completeness we must consider hierarchies rooted at each subset of the body of the input clause. For each clause represented in the hierarchy, we must also consider as a candidate hypothesis each clause generated by applying an inverse substitution. Each absorption step must consider each background clause of the program. Every unit background clause, having an empty body, may be used for absorption with any input clause.

We have characterised the predicates in the body of an input clause as the features of an example. But what of the constant and function symbols of the input clause — the symbols representing the objects concerned in the example? In the completeness theorem (9.6) we can see that constant and function symbols initially present in an input clause usually remain in the generalizations in the hierarchy until the final inverse substitution. Absorption removes a symbol only when a background clause has that symbol in its body but not its head. Such background clauses may be present in a program, but they are not typical. Removing the symbols by applying a final inverse substitution is as problematic as deleting literals from the body of an input clause for the root of the absorption closure — it is a generalization which is made without reference to background knowledge.

This is particularly evident when we consider an initial input clause consisting of a ground unit atom — the usual form of examples in ILP. The input body is empty so the first steps of absorption must be unit absorption, each introducing new features, and later steps generalize only the introduced features. The objects which are present in the initial example occur in the head of the clause so they are ignored entirely until the final inverse substitution!

To solve this problem we can employ the flattening representation change (definition 3.6) which translates the constant and function symbols in the input clause and the background program to predicate symbols. Then all these symbols may be treated

uniformly in generalization and an initial ground unit input clause becomes a functor-free, non-unit input clause.

9.6.1 Flattening for absorption

By initially flattening an input clause and a background clause each constant and function symbol in the clauses is represented as a predicate symbol. Replacing a ground term by a variable in the deep representation is equivalent to removing some flat atoms in the flat representation.

Example 9.3 (Flattening and absorption) Consider the program

$$q(a)\leftarrow$$
$$q(b)\leftarrow$$
$$q(c)\leftarrow$$

Let I be the clause $p(a)\leftarrow$. Absorbing I with the first background clause gives $p(a)\leftarrow q(a)$. But $flat(I)$ is $p(x)\leftarrow a_0(x)$. Absorbing $flat(I)$ with the flattened first background clause, $q(y)\leftarrow a_0(y)$ gives $p(x)\leftarrow q(x), [a_0(x)]$. Choosing the most general of these, the feature a_0 of the input clause has been replaced by the "more general" feature q in the generalized clause. □

Another example of the effect is given at the beginning of section 2.6. Notice that a constant symbol in a clause does not appear in a generalization when it is replaced by a predicate which holds for that symbol. In this way the mechanism by which absorption replaces predicates by more general ones elegantly extends to constant and function symbols. Any term in the input clause is replaced by a variable in a generalization exactly when sanctioned by background knowledge. Unflattening removes extraneous flat atoms that do not affect the clause and binds again some variables to other terms stood for by remaining flat atoms.

9.6.2 Limiting unit absorption

Because every unit clause in a program may participate in absorption with any input clause at every step of iterated absorption, limiting the use of unit absorption is crucial to improving the efficiency of generalization by absorption.

The flattening representation change can also help here. In a program which is the flat representation of an allowed program, every unit clause defines a flat symbol — a constant or function symbol in the deep representation. But the unit clauses themselves do contain constant and function symbols — so a clause created by unit absorption is not flat. Furthermore, the clause is significant not as an inductive hypothesis itself but only as an input clause for a later absorption step. The constant or function symbol introduced by a step of unit absorption cannot be removed by a later step of absorption; it can only be removed by a final inverse substitution. Like the constant and function

symbols in the initial input clause, we would like these constant and function symbols to be represented only as flat predicate symbols, so they may be removed by later steps of absorption when sanctioned by background knowledge.

But even in the flat representation, unit absorption must be permitted for completeness. Unit absorption introduces a flat atom into the body of an input clause which may enable absorption with a background clause that also has an instance of that flat atom in its body. As in the deep representation, unit absorption introduces another "feature" into the input clause for later generalization.

Our solution is to disallow unit absorption but instead to collapse multiple unit absorptions followed by a non-unit absorption into a single new operator called *k-unit absorption*. As before, it is defined for absorption of a goal, but the obvious extension to absorption of a clause by replicating the head of the input clause may be assumed.

Definition 9.9 (k-unit absorption of a goal, *k-absorb*) *Let* $\leftarrow I$ *be a functor-free definite goal and* B *be a non-unit clause in flat program* P *such that* I *and* B *share no common variables. Let there be a variable-pure substitution* θ *for variables in* B *and a set of flat atoms* U *such that (1)* $U \subseteq B_{\otimes}$ *and (2)* $(B_{\otimes} - U)\theta \subseteq I$ *and (3)* $U\theta \cap I = \{\}$ *and (4) no variable in the input of* θ *occurs as the first argument of an atom of* U*. Then the* k-unit absorption *of* I *with* B *using* θ*, k-absorb*(I, B, θ)*, is the set of goal bodies* $((I \cup U\theta) - (B_{\otimes}\theta)) \cup \{B_{\odot}\theta\} \cup [B_{\otimes}\theta]$.

Also, k-absorb(I, B) *is the set of goal bodies* G *such that there exists a substitution* θ *such that* $G \in$ *k-absorb*(I, B, θ).

$\leftarrow I$ *is called the* input *goal and* B *is the* background *clause. The atoms in the set* $B_{\otimes}\theta$ *are called* optional *and the atoms of* U *are called the* skipped flat atoms. *The number of skipped flat atoms is called the* level.

The condition (3) on the skipped flat atoms ensures that the set is as small as possible to enable absorption — adding only the minimal set of flat atoms which are missing from the body of the background clause. The condition (4) on the substitution ensures that k-unit absorption is indeed a generalizing rather than specializing operator: it constrains the skipped flat atoms to stand for terms occurring in the head of the deep version of the background clause. Of course, k-unit absorption at level zero coincides with the definition of absorption of non-unit flat clauses.

Example 9.4 (k-unit absorption) Consider the program

C_1: *married(richard, robyn)*$\leftarrow$
C_2: *mother(robyn, peter)*$\leftarrow$

and let I be *father(richard, peter)*$\leftarrow$. Then the flattened program (without the symbol-defining clauses) is given by

flat(C_1): $married(U, V) \leftarrow richard_0(U), robyn_0(V)$
flat(C_2): $mother(V, W) \leftarrow robyn_0(V), peter_0(W)$

and $flat(I)$ by $father(X,Y) \leftarrow richard_0(X), peter_0(Y)$.

The k-unit absorption of $flat(I)$ with $flat(C_1)$ at level 1 is the clauses of the form

$$father(X,Y) \leftarrow married(X,U), peter_0(Y), [richard_0(X), robyn_0(U)]$$

One of these, call it I', is $father(X,Y) \leftarrow married(X,U), peter_0(Y), robyn_0(U)$ Generalizing further, $k\text{-}absorb(I', flat(C_2))$ at level 0 includes

$$father(X,Y) \leftarrow married(X,U), mother(U,Y).$$

□

At level k, k-unit absorption introduces k existential variables to the body of the generalized clause. Unlike unit absorption, k-unit absorption does not introduce constant or function symbols to the clause, that is, it generates functor-free clauses. Notice that in the deep representation, the most obviously "relevant" kind of unit absorption — when the constant and function symbols of the background clause also occur in the input clause — is transformed to k-unit absorption at level 0 in the flat representation. This can be seen in the second step of the example.

What is the effect on completeness of the generalization hierarchy when unit absorption is not permitted but absorption is replaced by k-unit absorption? Consider a sequence of iterated absorption of an initial input clause, comprising some unit absorption steps to produce an intermediate clause, and then a step of non-unit absorption of the intermediate clause. Let us call the clauses produced by the non-unit absorption step the *absorption products*. Now the intermediate clause would not be represented in the k-unit absorption closure, but as it is only equivalently general to the input clause this will not affect the generality spanned by the remaining clauses. When the later application of non-unit absorption generates a strictly more general absorption product, k-unit absorption instead generates a functor-free clause directly from the initial input clause. Every absorption product clause is an *instance* of a functor-free clause created by k-unit absorption.

But the instantiating substitution merely binds variables occurring as the first argument of a skipped flat atom to the term they stand for. Because the skipped flat atoms are all optional, they are dropped from some of the clauses generated by k-unit absorption. This means that the clauses represented by k-unit absorption include all those of the absorption product plus some more general ones that have replaced constant or function symbol features in the input clause by "more general" predicate symbol features.

Example 9.5 (Relating k-unit absorption to absorption) Reconsidering example 9.4, a unit absorption step gives the intermediate clause

$$\begin{aligned} C' &= absorb(flat(I), (robyn_0(robyn) \leftarrow)) \\ &= father(X,Y) \leftarrow richard_0(X), peter_0(Y), robyn_0(robyn) \end{aligned}$$

A succeeding absorption step $absorb(C', flat(C_1))$ gives

$$father(X,Y) \leftarrow married(X,robyn),\ peter_0(Y),\ [richard_0(X),\ robyn_0(robyn)]$$

These clauses are instances of the clauses of $k\text{-}absorb(flat(I), flat(C_1))$:

$$father(X,Y) \leftarrow married(X,U),\ peter_0(Y),\ [richard_0(X),\ robyn_0(U)]$$

The instantiating substitution is $\{U/robyn\}$. Unflattening the clauses of $absorb(C', flat(C_1))$ we have

$$father(X,\ peter) \leftarrow married(X,\ robyn)$$
$$father(richard,\ peter) \leftarrow married(richard,\ robyn)$$

But unflattening the clauses of $k\text{-}absorb(flat(I), flat(C_1))$ we have

$$father(X,\ peter) \leftarrow married(X,U)$$
$$father(richard,\ peter) \leftarrow married(richard,\ U)$$
$$father(richard,\ peter) \leftarrow married(richard,\ robyn)$$
$$father(X,\ peter) \leftarrow married(X,\ robyn)$$

□

9.7 A Strategy for Generalization by k-unit Absorption

Now we can do a complete generalization search by replacing absorption in the completeness theorem (9.6) by k-unit absorption of flattened clauses. Beginning with an example that is a unit clause, flatten the clause and put each clause made of its head and any subset of its body into an initial working set. Apply k-unit absorption to each clause in the set paired with each suitable flattened background clause (excluding unit symbol-defining clauses), and include the resulting clauses in the set. Continue doing so indefinitely or until the set no longer grows. This set is the k-unit absorption closure of the flattened example, *k-absorb**.

Consider any clause in the set as an inductive hypothesis. Also consider any clause for which there is a substitution which can transform it to one in the set. All of these are candidate inductive hypotheses more general than the example. Some of them will contain flat atoms and should be unflattened before adoption.

But there remain some ways this procedure may be simplified.

9.7.1 Free variables in a background clause

Earlier, when working in the deep representation we suggested that variables in the head of a background clause that do not also occur in the body of the clause give rise to a large range of possible suitable substitutions for absorption. We suggested that by restricting the language of background clauses to allowed clauses this problem does not occur. Unfortunately, when an allowed program which includes function symbols (of non-zero arity) is flattened, it is no longer allowed. Although the flat form of a clause

is allowed if the clause itself is allowed, the symbol-defining clauses for the function symbols are not allowed.

However, using k-unit absorption, such a unit background clause is never directly used for absorption. The choices for a suitable substitution in unit absorption are replaced by choices for the substitution in k-unit absorption. Fortunately, for k-unit absorption we need only consider variable-pure substitutions at this stage, thus considerably constraining the substitution. Furthermore, like for absorption in the deep representation, if an input clause is not allowed then the result of k-unit absorption with an allowed background clause is also not allowed. Because MINERVA seeks only allowed clauses, the search space may be pruned at any non-allowed clause.

9.7.2 Choosing the root

Just as when we were working in the deep representation, the completeness theorem (9.6) requires consideration of generalization hierarchies rooted at each subset of the body of the example. For the deep representation we suggested that a meaningful relevance assumption is made by considering only the absorption closure rooted at the maximal subset.

The same argument applies to the flat representation. When the clause for generalization is a unit clause example, the features in the body of the flattened example are the constant and function symbols of the example. By searching for inductive hypotheses only in the k-unit absorption closure of the full body, completeness is sacrificed but the original features are either retained or replaced by more general features in every hypothesis considered. This is what MINERVA does.

Example 9.6 (Choosing the maximal root) Let P be a definite program including the clause $q(a)\leftarrow$. Let P' be the flattened form of P. Let I be $p(a,b)\leftarrow$. Then $flat(I) = p(x,y)\leftarrow a_0(x), b_0(y)$. Let I' be $p(x,y)\leftarrow a_0(x)$. Let H be $p(x,y)\leftarrow q(x)$. Now $H \in \textit{k-absorb}^*(I',P')$ but $H \notin \textit{k-absorb}^*(flat(I),P')$, although $H \succeq_P I$. □

9.7.3 Avoiding inverse substitution

Working in the flat representation does not obviate the inverse substitution in the completeness theorem (9.6). For completeness, some clauses that do not occur in the k-unit absorption closure must be considered as inductive hypotheses, *viz.* those for which an instance occurs in the closure. Each clause in the closure is flat and functor-free but an inverse substitution could replace multiply occurring instances of a single variable by distinct variables. When the variable occurs as the first argument of a flat atom in the clause this corresponds to replacing a term by a variable in the deep representation. This is undesirable by the relevance assumption — either the new clause already occurred in the k-unit absorption closure or the generalization is not justified by background knowledge.

But the inverse substitution may instead replace a variable occurring multiply in atoms about observational predicates by distinct variables. Again, in this case the available background knowledge does not sanction the generalization.

For this reason, MINERVA avoids the final inverse substitution when working in the flat representation. The consequence of this decision is that when a feature appears multiply in an input clause every considered hypothesis replaces every occurrence of the feature by the same more general feature.

Example 9.7 (Avoiding inverse substitution) Consider the program P

$$q(a) \leftarrow$$
$$r(a) \leftarrow$$

If P' is the corresponding flattened program and I is the clause $p(a,a) \leftarrow$, then $flat(I)$ is $p(x,x) \leftarrow a_0(x)$ and $k\text{-}absorb^*(flat(I), P')$ is

$$p(x,x) \leftarrow a_0(x)$$
$$p(x,x) \leftarrow q(x),\ [a_0(x)]$$
$$p(x,x) \leftarrow r(x),\ [a_0(x)]$$
$$p(x,x) \leftarrow q(x),\ r(x),\ [a_0(x)]$$

Notably it does not include the clause $p(x,y) \leftarrow q(x),\ r(y)$ although it is more general than I with respect to P. □

9.8 Generalizing Normal Clauses

So far in this chapter we have been concerned with using absorption to generalize definite clauses with respect to background knowledge represented as a definite program. What happens when clauses are permitted negative antecedents, as in MINERVA? To answer this question we first develop a model of generality for normal clauses with respect to normal programs, and then we show how absorption can generalize in this model. To simplify this discussion, we return again to the deep representation of clauses and programs.

9.8.1 Normal subsumption

The generalized subsumption model for definite clauses and definite programs is extended to model generality of normal clauses with respect to normal programs by the author [Taylor 1993]. The model of generality is known as *normal subsumption* and coincides with generalized subsumption when only definite clauses and definite programs are concerned. Like generalized subsumption, normal subsumption relates clauses that are about the same predicate.

Definition 9.10 (Normal Subsumption, $\sqsupseteq_P$) *Let C and D be normal clauses and P a normal program. Then $C \sqsupseteq_P D$ (C is more general than D with respect to P by* normal subsumption*) if for any Herbrand interpretation I (for the language of at least P, C, and D) such that comp(P) is true in I, and for any atom A such that D covers A in I, C also covers A in I.*

Normal subsumption can be tested by a computation of a logic programming system. This is established by the following lemma and theorem which are included here to aid the reader's understanding of normal subsumption. The result is analogous to Buntine's [1988] "operational" view of generalized subsumption (theorem 3.1).

Lemma 9.7 (Evaluating Normal Subsumption) *Let C and D be normal clauses and P a normal program. Then $C \sqsupseteq_P D$ if and only if there exists a substitution θ such that $C_{\odot}\theta = D_{\odot}$ and comp(P) $\models \forall(D_{\otimes} \rightarrow C_{\otimes}\theta)$.*

Proof. (*If* part) Assume $C_{\odot}\theta = D_{\odot}$ and *comp*(P) $\models \forall(D_{\otimes} \rightarrow C_{\otimes}\theta)$. Assume that A is an atom covered by D in I, an appropriate interpretation in which *comp*(P) is true. Then there is a substitution, say σ such that $D_{\odot}\sigma = A$ and $\exists(D_{\otimes}\sigma)$ is true. Therefore there is a substitution $\theta\sigma$ such that $C_{\odot}\theta\sigma = A$. Further, $\exists(D_{\otimes}\sigma)$ is true in I, so there exists a ground substitution γ for the variables in $D_{\otimes}\sigma$ such that $D_{\otimes}\sigma\gamma$ is true in I. Because *comp*(P) is true in I and *comp*(P) $\models \forall(D_{\otimes} \rightarrow C_{\otimes}\theta)$, $\exists C_{\otimes}\theta\sigma\gamma$ is true in I. Therefore $\exists C_{\otimes}\theta\sigma$ is true in I. Therefore C covers A in I (definition 3.2). Therefore $C \sqsupseteq_P D$.

(*Only if* part) Assume $C \sqsupseteq_P D$. Then there exists a θ such that $C_{\odot}\theta = D_{\odot}$, by definition 9.10. Let I be any appropriate interpretation in which *comp*(P) is true. Now, for any ground substitution σ such that $D_{\otimes}\sigma$ is true in I, D covers the ground atom $D_{\odot}\sigma$ (definition 3.2). Because $C \sqsupseteq_P D$, C also covers $D_{\odot}\sigma$, so $\exists C_{\otimes}\theta\sigma$ is true in I. That is, in I, $D_{\otimes}\sigma \rightarrow \exists C_{\otimes}\theta\sigma$. Recalling that σ is any ground substitution it follows that $\forall(D_{\otimes} \rightarrow C_{\otimes}\theta)$ holds in I. Recalling that I is any interpretation in which *comp*(P) is true, *comp*(P) $\models \forall(D_{\otimes} \rightarrow C_{\otimes}\theta)$. □

Theorem 9.8 (Testing for normal subsumption) *Let C, $C_{\odot} \leftarrow C_1, \ldots C_n$ and D be normal clauses and P a normal program. Let θ be a substitution for variables in $C_{\odot}$ such that $C_{\odot}\theta = D_{\odot}$. If for all $C_i \in C_{\otimes}$ the safe computation of $P \cup \{\leftarrow(D_{\otimes} \cup \{\sim C_i\theta\})\}$ finitely fails then $C \sqsupseteq_P D$.*

Proof. By soundness of NAF (theorem 3.2), for each C_i, *comp*(P) $\models \forall(\leftarrow D_{\otimes} \wedge \sim C_i\theta)$. That is, *comp*($P$) $\models \forall(\leftarrow D_{\otimes} \wedge \sim C_1\theta) \wedge \ldots \wedge \forall(\leftarrow D_{\otimes} \wedge \sim C_n\theta)$. Rearranging, *comp*($P$) $\models \forall(\sim D_{\otimes} \vee (C_1\theta, \ldots, C_n\theta))$. Rearranging, *comp*($P$) $\models \forall(D_{\otimes} \rightarrow C_{\otimes}\theta)$. Therefore, because $C_{\odot}\theta = D_{\odot}$, the lemma (9.7) applies and $C \sqsupseteq_P D$. □

Unfortunately this test is not always useful because of the requirement that the computations be safe. Even if the program and the clauses involved are *allowed*, the

computation may flounder. It is, however, easy to show that the computation is always safe when the program and clauses are allowed and the more general clause (C in theorem 9.8) is *nvi* (definition 3.15).

9.8.2 Normal absorption

The absorption operator of Muggleton and Buntine has been extended to generalize normal clauses with respect to normal programs by the author [Taylor 1993]. This new operator, called *normal absorption*, generalizes according to normal subsumption. It coincides with absorption when only definite clauses are involved. The definition is motivated by a kind of closed world assumption. If some negative literal occurring in a background clause is not represented by an instance in the body of an input clause then normal absorption dictates that it can be assumed to be there, as long as there is no evidence to the contrary.

The notation $C_\oplus$ is used to mean $\{A | A \in C_\otimes \text{ and } A \text{ is an atom}\}$ and $C_\ominus$ is used to mean $\{{\sim}A | {\sim}A \in C_\otimes \text{ and } A \text{ is an atom}\}$.

Definition 9.11 (Normal Suitability) *Let I and B be normal clauses with distinct variables and let P be a normal program. Let θ be a substitution for variables in B. Then I and B satisfy the* normal suitability criterion relative to P using θ, *$suitable_P(I, B, \theta)$, when $B_\oplus\theta \subseteq I_\otimes$ and $comp(P) \models \forall(I_\otimes \rightarrow B_\ominus\theta)$.*

Definition 9.12 (Normal Absorption) *Let I and B be normal clauses with distinct variables and let P be a normal program. Let θ be a substitution such that $suitable_P(I, B, \theta)$. The* normal absorption *of I with B using θ, $n\text{-}absorb_P(I, B, \theta)$, is the set of clauses*

$$(I_\odot \leftarrow (I_\otimes - B_\otimes\theta) \cup \{B_\odot\theta\} \cup [B_\otimes\theta])\Theta$$

where Θ is any inverse substitution.

The literals in $B_\otimes\theta$ are said to be optional. *In this role I is called the* input clause *and B the* background clause.

The clause including every literal of $B_\otimes\theta$ in the body and with Θ as the identity inverse substitution is called least general normal absorption *with respect to P.*

I proved elsewhere [Taylor 1993] that normal absorption is sound: it does indeed generalize normal clauses with respect to normal programs according to normal subsumption.

9.8.2.1 Computing normal absorption

The application of normal absorption follows in the same manner as for definite absorption apart from the determination of the suitability criterion. Sometimes there is more than one substitution with the desired properties and there is usually a number

of choices for inclusion of optional literals and the inverse substitution. As for definite absorption, these choices may be made according to the learning strategy in which the absorption operator is embedded.

The following theorem describes a computation which can be performed by a logic programming system with a safe computation rule that is sufficient to determine suitability.

Theorem 9.9 (Determining normal suitability) *Given a normal program P and normal clauses I and B with distinct variables, if there is a substitution θ such that $B_{\oplus}\theta \subseteq I_{\otimes}$, and if for each literal L_i such that $\sim L_i \in B_{\ominus}$, the safe computation of $P \cup \{\leftarrow I_{\otimes} \cup \{L_i\theta\}\}$ finitely-fails, then $suitable_P(I, B, \theta)$.*

Proof. By the soundness of NAF (theorem 3.2), if each computation fails then for all L_i defined as above $\leftarrow I_{\otimes} \cup \{L_i\theta\}$ is a logical consequence of $comp(P)$. That is, $comp(P) \models (\forall\sim(I_{\otimes} \cup \{L_1\theta\}))\wedge\ldots\wedge(\forall\sim(I_{\otimes} \cup \{L_n\theta\}))$. Rearranging, $comp(P) \models \forall(I_{\otimes}\rightarrow(\sim L_1\theta\wedge\ldots\wedge\sim L_n\theta))$. That is, $comp(P) \models \forall(I_{\otimes}\rightarrow B_{\ominus}\theta)$. □

The effect of normal absorption is illustrated in the following example. Although the predicate denoted *father*^{0} in the example is an invented predicate, it is treated identically to any other predicate in generalization by normal absorption, and could be replaced by an observational predicate such as *stepfather* instead.

Example 9.8 (Normal absorption) Consider the normal program P, describing part of the family of figure 2.1.

mother(kim, kate)←
married(bill, kim)←
stepfather(bill, kate)←
mother(sally, james)←
mother(sally, mary)←
married(john, sally)←
brother(matthew, john)←
brother(andrew, john)←
father^0*(X, Y)*← *stepfather(X, Y)*
father(X, Y)← *married(X,Z), mother(Z, Y),* ∼ *father*^{0}*(X, Y)*

Let B be the latter clause of P. Assume a positive example

uncle(andrew, james)

led to the inductive hypothesis I:

uncle(U, V) ← *brother(U,F), married(F, W), mother(W, V)*

Then $suitable_P(I, B, \theta)$ with $\theta = \{X/F, Z/W, Y/V\}$ because a safe computation of $P \cup \{\leftarrow$*brother(U,F), married(F, W), mother(W, V), father*^{0}*(F, V)*$\}$ finitely fails. So $n\text{-}absorb_P(I, B, \theta)$ is the set of clauses

$$uncle(U,V) \leftarrow brother(U,F),\ father(F,V),\ [married(F,W),\ mother(W,V),\ \sim father^0(F,V)]$$

□

9.8.3 Difficulties with normal absorption

Unfortunately this theoretical model for normal absorption of normal clauses has some significant drawbacks.

9.8.3.1 Theoretical shortcomings

Normal absorption does not share with definite absorption a natural interpretation as the inversion of one step of binary resolution. The input clause (the resolvent in the corresponding resolution step) may fail to include some negative literals that occur in the product of absorption and hence in the resolvent constructed from it and the background clause. However, we may view normal absorption as comprising two steps: the negative literals missing from the input clause are first added to its body, then absorption of that input clause with the background clause proceeds as for definite absorption. In this case, the second step being absorption of normal clauses does correspond to inversion of resolution. The first step is justified by the closed world assumption: add the negative literals unless there is a evidence that it should not be done, as embodied in the suitability criterion. The closed world assumption is not a classical logic inference rule, so the non-existence of a counterpart resolution step is not surprising.

It is also difficult to cast normal absorption as the inversion of SLD-resolution, or even SLDNF-resolution, as we did for definite absorption. For absorption this enabled both the reduction of some of the indeterminacy customarily required of the operator and a completeness result. It is not obvious that an analogous route may be taken for normal subsumption.

9.8.3.2 Safe and efficient computation

One of the advantages of definite absorption as a generalization mechanism is its ability to generalize from background knowledge by the evaluation of a simple subsumption test. The simplicity is not inherited by normal absorption which requires a resource-expensive computation, possibly involving reference to all other clauses in the program in order to evaluate the suitability criterion. This test is necessary because of the global nature of the closed world assumption.

Computation of the suitability criterion might not even terminate. It might be undecidable. The non-termination problem was discussed in depth in a more general context in chapter 5, but it remains a disadvantage of normal absorption.

Furthermore the suitability criterion is dependent on a safe computation by the logic programming system. This is not always possible for arbitrary programs, although it can be guaranteed for allowed input clauses with allowed programs. In this case the goal of each computation is the body of an allowed input clause plus some positive literal, so it is also allowed.

9.8.3.3 Strictness

As for definite absorption, normal absorption is a generalizing operator. Every atom covered by an input clause remains covered by every product of normal absorption: there can be no exceptions. But sometimes this requirement is too strong. Errors in the background program may mean that some false atom is covered by a clause with respect to the program although it is not covered by the clause in the target. By permitting such exceptions a correct inductive hypothesis could be formulated even when background knowledge is incorrect. Later corrections to the program could fix the source of the problem, and there would no longer be an exception to the clause in question.

Moreover, sometimes expression of a concept in terms of a general rule together with exceptions is natural and compact, as in the clause defining *father* of example 9.8.

9.9 The Strategy for Generalization in MINERVA

Because of the difficulties of the theoretical model of normal absorption and the special nature of negative literals in the clauses of MINERVA — as exception predicates — MINERVA does not use the normal absorption operator. Instead, MINERVA approximates the operator using the k-unit absorption operator (definition 9.9).

9.9.1 Implementing normal absorption

When generalizing, a negative literal may be included in an inductive hypothesis for the purpose of excluding certain exceptions from being covered by the clause. At this point an inductive hypothesis should be viewed as a set of clauses: the generalization produced by absorption then modified by exception, supplemented with clauses defining the exceptions to that clause. Although the excepted clause could be generalized further, MINERVA does not do so, pruning the generalization search space at the node. This is for pragmatic reasons — preliminary experiments suggested that little is gained by generalizing further. Exceptions arise when the clause covers a false atom with respect to the background program. When an exception atom is also covered by the clause in the target, it is often the case that investigation of an alternative generalization which is not excepted leads to a better inductive hypothesis than further generalization of the excepted one. When the atom is covered with respect to the program but not in the target, the errors in the program continue to interfere with each more general

hypothesis and tend to obscure recognition of an otherwise good hypothesis.

Nevertheless, excepted clauses may be adopted and become part of the program under construction by MINERVA. Such clauses become candidates as background clauses in subsequent generalization steps. Normal absorption could be used to generalize an input clause with such a background clause, but instead MINERVA simply deletes the negative literals from the background clause and then performs a definite k-unit absorption step. Recalling that the negative literals are intended only to represent exceptions to a clause, this action amounts to ignoring exceptions otherwise covered by a background clause. To illustrate the effect of this approximation to normal absorption by ignoring exceptions, the earlier example is revisited.

Example 9.9 (Implementing normal absorption) Consider program P, input clause I and background clause B of example 9.8. Then in MINERVA, the absorption of I with B is implemented as the k-unit absorption of I with B' where $B' = father(X,Y) \leftarrow married(X,Z), mother(Z,Y)$. Then $\theta = \{X/F, Z/W, Y/V\}$ is the only suitable substitution and *k-absorb*(I, B', θ) is given by

$$uncle(U,V) \leftarrow brother(U,F),\ father(F,V)[married(F,W),\ mother(W,V)]$$

In this case the approximation of normal absorption misses some of the clauses of the normal absorption — it does not have the clauses which include the negative literal. □

9.9.2 Implementing k-unit absorption

Given that we have excluded the participation of negative antecedents, the formal analysis of absorption in this chapter suggests the generalization strategy for MINERVA described in section 5.3.2 and figure 6.2. Generalizations of a flattened missing atom are generated by iterated applications of most general k-unit absorption coupled with restriction (definition 3.5). For brevity, outside this chapter we have called most general k-unit absorption of a flat clause *f-absorption*. Here is a definition for it, taken from the earlier definition of k-unit absorption (definition 9.9) modified to take account of negative antecedents and appropriate for coupling with restriction. It generates a set of pairs each comprising a flat clause and a set of optional atoms: the clause being the most general k-unit absorption of the flattened input clause and background clause at a given level.

Definition 9.13 (f-absorption, *f-absorb*) *Let I be a functor-free definite clause (called the* input clause*) and B' be a clause of normal program P such that I and B' share no common variables. Let B be constructed from B' by deleting negative antecedents and flattening. Then the* f-absorption *of I' with B at* level *K is the set of pairs of an inductive hypothesis $I_{\odot} \leftarrow ((I_{\otimes} \cup U\theta) - (B_{\otimes}\theta)) \cup \{B_{\odot}\theta\}$ and a set of* optional *atoms $\{B_{\otimes}\theta\}$ such that there exists a variable-pure substitution θ for variables in B and a set of K flat atoms U such that (1) $U \subseteq B_{\otimes}$ and (2) $(B_{\otimes} - U)\theta \subseteq I_{\otimes}$ and (3) $U\theta \cap I_{\otimes} = \{\}$ and (4) no variable in the input of θ occurs as the first argument of an atom of U.*

Before f-absorption can be applied, a definite input clause to be generalized is flattened. A background clause from the program is selected, negative antecedents are deleted and it is flattened. Then f-absorption and restriction can be iteratively applied to generate flat generalizations. These are unflattened prior to experimental evaluation and adoption.

Example 9.10 (f-absorption) Reconsider program P, input clause I and background clause B of example 9.8. Then the f-absorption of I with B generates the clause $uncle(U,V) \leftarrow brother(U,F), father(F,V)$ and the optional atoms $\{married(F,W), mother(W,V)\}$. □

A number of important questions are not answered by f-absorption. While building up the hypothesis space, which clause in the space should be chosen as the next for development? Which suitable background clause should be chosen? When is restriction of a hypothesis a better next step than f-absorption of the hypothesis? How can experimentation be used within the scheme to reject some hypotheses before they are evaluated — to prune the search in the hypothesis space? How can exception predicate invention be incorporated to manage errors in the background program during generalization? Of the hypotheses in the set, which one should be adopted eventually and assimilated into the program? These questions were answered heuristically in chapter 6.

9.10 Summary

In this chapter we have described the generalization operator absorption and showed how it acts as a complete generator for the space of generalizations of a single clause, as defined by generalized subsumption. Although we initially presented absorption as an operator on clauses with multiply occurring antecedents, we have demonstrated advantages to be gained by working with absorption on clauses with unique antecedents. Coupled with the restriction operation to structure the clauses generated by a single application of absorption, a search space for inductive hypotheses more general than an input clause is mapped out.

Some of the clauses in the space may be ignored as inductive hypotheses without compromising completeness, but must remain in the generalization hierarchy to enable generation of other hypotheses. In particular this applies to non-connex clauses, clauses generated by least general absorption, and clauses generated by unit absorption. We have described how k-unit absorption may be used to delay the representation of nodes of the latter kinds until needed for a later generalization step.

We have described two ways in which the search space for generalizations may be significantly simplified at the expense of completeness: by considering only one initial subset and avoiding the final inverse substitution step. But we have argued that, with the aid of the flattening representation change, we only give up some generalizations which are not justified by background knowledge.

We then turned to considering generalization of normal clauses. After developing a model for generality of such clauses, a normal absorption operator was defined and analysed. But, considering the limited use of negative antecedents in MINERVA and some difficulties of the operator, we have described a practical approximation using the definite clause absorption operator.

Finally we named this practical absorption operator based on the most general k-unit absorption operator for flat definite clauses: *f-absorption*. This, coupled with restriction, is the incremental generalizing operator of MINERVA.

10

Conclusions

10.1 Introduction

In this final chapter the threads of the argument developed in the thesis are drawn together by a summary of the work and identification of the opportunities it creates for further research.

10.2 Summary of the Thesis

This work reports the foundational theory, practical implementation and performance results for an active, incremental, first order learner called MINERVA. The learner is continuously interacting with an environment which is the source of learning goals, initial and experimental observations, time resource bounds and ultimate performance evaluation.

MINERVA is set apart as a learning algorithm by its design for incremental improvement whereby it continuously searches for improved inductive hypotheses until its attention is demanded. In complex domains, when background knowledge is weak or faulty, or when examples are observed in a confusing sequence, MINERVA requires more time to develop a good inductive theory. But *something* will be learnt even if only the examples themselves. Given a little more time, diagnosis enables deeper problems in the theory to be repaired. With more time again, it invokes an active learning strategy to eagerly attempt to learn more about the concept at fault.

Fundamental to the incremental improvement of MINERVA is the use of the absorption operator for generalization, and predicate invention for specialization. These operators have been placed on formal foundations in the work. A special form of the generally-

known absorption operator for definite clauses has been defined, offering advantages for incremental generalization while preserving completeness. A new absorption operator for generalization of normal clauses has also been developed. It generalizes clauses in the context of background knowledge represented as a normal program, justified by a new generality model.

A heuristic evaluation function has been introduced to support the stepwise development of revision hypotheses by generalization, specialization and experimentation. The heuristic anticipates the value of an inductive hypothesis in terms of its syntactic simplicity, taking into account the new facts it brings to a theory.

MINERVA incorporates the tools of logic programming technology for interpretation of the programs it learns and for declarative diagnosis of errors. These tools enable effective learning of multiple interdependent concept theories. In MINERVA the conventional tools have been adapted to the needs of a learner with an unusually expressive description language.

In the thesis, the environment is simulated by SAMPLER, a new tool providing random incremental sampling of examples from a user-defined domain and answering the learner's questions about the domain. Samples are drawn according to a user-defined sample strategy: some strategies are designed to present simple examples more frequently than complex ones.

The learning performance of MINERVA has been demonstrated by experiments with some "standard" domains from the machine learning literature. Further experiments demonstrated the unique features of MINERVA's relationship with its environment.

Opportunities for further research building on the achievements of this thesis may be placed along two significant dimensions. The first comprises "tuning" improvements that might improve performance in the framework we have set up. The second relaxes the assumptions about the problem that MINERVA is designed to solve.

10.3 Improvements to MINERVA

First we consider some ways that enhancements to MINERVA might improve its learning performance on the kinds of problems we have demonstrated in the experiments of chapter 8.

Recognizing limited memory resources

MINERVA models the finiteness of resources only in terms of time limits, but the search for a theory often traverses a large space and memory bounds can be significant too. Although the implementation of MINERVA for which experimental results are reported here could be improved to be more space efficient, in any case finite memory resources will bound the search eventually.

In MINERVA a number of fixed parameters are used to bound the use of memory resources for different purposes: such as the size of fact memory, the depth of derivations, and the size of the partial hypothesis list. But as these bounds are independently fixed they do not always make the best use of available memory. Moreover, they are not always sufficient to ensure that the available memory is not exhausted: sometimes the execution is prematurely halted for lack of memory. This problem is compounded when SAMPLER and MINERVA are executed on the same computer as in the experiments of chapter 8 — so less memory is available to MINERVA exactly when it needs more.

If MINERVA had the introspective ability to determine memory usage and to act accordingly, performance could be improved. Indeed, this capability is probably essential to the successful integration of any learner in an intelligent software agent. But recognition alone is not sufficient, a learner must be able to modify its behaviour dynamically to allocate scarce memory resources so that performance degrades gracefully.

Integrating batch learning

In MINERVA, theory revision is prompted only by observation of a failing example. This focus on incremental revision offers advantages in responsiveness to every example and readiness for environmental interruptions. But batch learning has generally had better empirical success with accuracy and program simplicity than incremental learning. Indeed, Mooney [1992] concluded from experiments that as a learner is made "more incremental", learning time decreases but so too does accuracy and simplicity. Although MINERVA can improve program simplicity during the off-line sleeping procedure, the underlying theory is not affected — there is no inductive learning at this time. Perhaps MINERVA's accuracy performance could be improved by incorporating inductive batch learning into the sleeping procedure.

Inventing new predicates

MINERVA's ability for predicate invention is limited. When a correct concept definition necessarily depends on a concept that is missing from the language of observation [Buntine 1988] the predicate invention capability of MINERVA cannot invent a predicate to take its place. The ability to invent such predicates autonomously is probably necessary for an adequate model of "intelligent" learning, underlying the human capability for imagination and scientific discovery. Nevertheless, MINERVA's design readily supports the incorporation of predicate invention techniques including those suggested by De Raedt and Bruynooghe [1992], Hume and Sammut [1991b] and Silverstein and Pazzani [1991].

Developing heuristics

Within the design of MINERVA there is opportunity to study the effect of variations to the heuristic-guided choices for hypothesis selection. Other ways of determining clause

complexity or the value of alternative hypotheses could be considered.

Furthermore there are other places within MINERVA where attention to heuristic choices could result in better performance. Such heuristics could be based on "confidence" in the accuracy of the clauses and the predicates on which they depend; on how close each choice comes to enabling a proof; or on the number of predicates on which a clause depends. Confidence could be based on the age or frequency of use or even the heuristic value used elsewhere. This could help to reduce questions in diagnosis and revision and to improve the generalization search by encouraging generalization with the aid of background knowledge recognized to be "useful".

In particular, fewer questions might be asked in diagnosis by varying the order of questions about clause antecedents and varying the order of trying program clauses that unify with a missing atom. In MINERVA's present form the top-down version of contradiction backtracing is employed partly because it is amenable to incorporation of heuristics, but they are not employed.

As programs grow larger over the long life of a learner, it will probably be necessary to constrain the generalization search by using a more sophisticated measure of the relevance of background knowledge to a particular learning problem. This could be aided by incorporating a notion of relevance in SAMPLER too: so that examples presented to the learner successively are likely to be "close" in the program. Indeed this idea was critically important to the learning performance of MARVIN [Sammut 1981b].

Improving missing answer diagnosis

In MINERVA's missing answer diagnosis the membership interpreter is used to find answers satisfying as many as possible of the literals of the body of a clause, counting from the left. This often finds a substitution to ground the variables in the leftmost literal that is not provable, enabling a membership query to be asked about the literal instance and thus enabling better diagnosis. This could be improved further by dynamically ordering the literals so that every order is tried, and thus every ground answer to a goal that is a subset of the literals is tried, before there is a need to ask any questions about non-ground atoms.

The number of questions asked in missing answer diagnosis could be reduced even further using a selective backtracking approach like that of Pereira and Porto [1982]. In MINERVA, the variables in a literal of the body of a clause are often bound to terms by a substitution for the same variables occurring in literals to the left in the clause body. When this occurs for some literal instance for which a negative answer is given to the membership query, the search backtracks chronologically — to the subgoal immediately to the left to find alternative answers for it, even when those alternatives can not affect the bindings of the variables of the literal in question. Selective backtracking could instead find other answers for the earlier subgoals which bound those variables, thus avoiding unnecessary questions about the subgoals in between.

Although these enhancements to MINERVA's missing answer diagnosis could be made at

little cost, in most circumstances they could offer only minor performance improvement.

Interpreting theory representations

The interpreters employed in MINERVA have been developed as a compromise to satisfy goals of language expressiveness, soundness, completeness and efficiency. Unfortunately the compromise implemented in MINERVA occasionally lets it down: performance is impeded by the interpreter's inability to interpret a goal in a timely fashion. This is particularly important when derivations include non-*nvi* or non-definite clauses and when theories grow large. Further development in this area, possibly requiring restriction of the representation language and changes to the interpreters will be required for performance improvements. It should be possible and beneficial to extend the loop check mechanism already implemented in MINERVA's interpreters and diagnoser to take account of negative antecedents.

The use of a multi-valued logic like that suggested by De Raedt [1992] or that implemented in MOBAL [Morik et al. 1993] could have some advantages, although restricting expressibility in some ways. More generally, MINERVA should be able to manage contradiction in its theories — either apparent contradiction resulting from resource limitations or deliberate contradiction permitted for convenience of expression. Perhaps MINERVA could carry multiple theories, each modulated by framing conditions. Contradictory beliefs are not paradoxical to the young child nor to many intelligent adults.

10.4 Expanding the Environment

Elements of MINERVA could be useful for applications in machine learning beyond the scope of the problem framework of this thesis, such as computer program synthesis and knowledge acquisition for expert systems. But this thesis has particularly aimed at the goal of time-limited autonomous concept learning by an agent interacting with an environment. Let us look at some ways we can relax the environmental model of this work and the implications this would have for MINERVA.

Noise in the environment

A practical implementation of MINERVA in a physical environment must take account of noise. The work reported here has avoided the issue of noisy input in order to focus on other aspects of the problem.

Nevertheless, the design of MINERVA neatly supports the inclusion of the common noise-handling mechanisms described by Brazdil and Clark [1990]. A preprocessing stage could filter the input data prior to presentation as examples to MINERVA, passing on only the most representative examples — and accessing the current theory to aid the filtering process. Ranges of values for attributes could be grouped so that the coarse

grain size of the attributes hide small fluctuations in their original values. Alternatively, MINERVA could be modified to deal with random errors by incorporating a statistically defined reliability threshold in the heuristic evaluation function. Meanwhile, the support for unit clauses describing observational and exception predicates prevents small numbers of isolated errors from disproportionately affecting the theory. Furthermore, the fundamentally active nature of MINERVA permits re-testing for doubtful facts or even querying a teacher for justifications of them.

At present SAMPLER features the ability to corrupt randomly sampled examples according to a user-defined probability parameter. In future work MINERVA's performance in the presence of noise will be empirically investigated, beyond the brief attention given here, and improved.

Incomplete environment

In the design of MINERVA we have assumed that the environment is always able to answer a question, given sufficient time. This would be an unrealistic assumption for a real world environment, even when a teacher is available to answer questions. Hume and Sammut [1991b] address some of the difficulties of experimentation in an ILP framework — their solutions could be incorporated in MINERVA. Alternatively, MINERVA could be easily adapted to accept "don't know" answers by assuming the theory is presently correct on such facts during diagnosis and ignoring such facts in experimental evaluation. A more sophisticated solution could be to address the problem as part of the the noise handling mechanisms previously discussed.

Unsurprising interruptions

The MINERVA and SAMPLER duo model limited time resources as surprising interruptions to MINERVA. A richer but realistic model would have some element of predictability of interruptions: MINERVA should be able to estimate available time and to internally allocate resources to the diverse components of the learning process. This could include a capacity for lemma assertion in the program (also called *chunking* [Tambe, Newell and Rosenbloom 1990] or *operationalization* [DeJong 1988]) to increase logical redundancy in order to hasten reasoning. It could also include an ability to recognize and reason about the logical inconsistencies introduced into the model of knowledge by resource limitations.

Using a helpful teacher

The model of the environment could also be made more "learner-friendly". There is no doubt that if a software learning agent is to approach the learning skill demonstrated every day by humans, it must be able to learn from a wide variety of sources and also to understand the strengths and limitations of those sources. Active experimentation in an environment is not enough: dialogue with a helpful and benevolent teacher will be

necessary. MINERVA must be able to ask "why?" and to accept explanatory answers. Access to encyclopaedic information sources might also be required to enable learning from the collected human experience. The technologies of intelligent tutoring systems [Sleeman 1983, Sleeman, Hirsh, Ellery and Kim 1990] and foundational knowledge representation [Guha and Lenat 1994] could contribute.

One might also envisage learning from peers. What if two agents, provisioned with MINERVA's learning ability and placed in distinct environments could be brought together to share their knowledge? Each agent in turn could take the role of SAMPLER to tell what they know, although one might expect them to converse in a more expressive language than is possible with the external environment. The learners' theories may be mutually inconsistent on some points: perhaps they could resolve their differences by further experimentation in the environment or by justifying their conclusions in terms of the explanatory power or simplicity of the hypotheses which imply them.

10.5 Concluding Remarks

We started this work with a question: how do intelligent beings learn? Whilst it would be overstating the achievements to suggest that we have comprehensively answered the question, we have contributed a novel model of autonomous learning that is supported by theoretical and empirical performance results. We have shown how an agent can use observations of an environment to reason about events which are not observed; to predict the consequences of actions; and to form an internal model of its world. We have demonstrated the feasibility of our approach to learning for implementation in the flexible, intelligent autonomous agents of the future.

Bibliography

Angluin, D.: 1990, Negative results for equivalence queries, *Machine Learning* **5**(2), 121–150.

Apt, K. R. and Bezem, M.: 1990, Acyclic programs, *Proceedings of the 7th International Conference on Logic Programming.*

Bain, M.: 1991a, Experiments in non-monotonic first-order induction, *in* S. Muggleton (ed.), *Proceedings of the International Workshop on Inductive Logic Programming ILP-91.*

Bain, M.: 1991b, Experiments in non-monotonic learning, *in* L. A. Birnbaum and G. C. Collins (eds), *Machine Learning: Proceedings of the Eighth International Workshop*, Morgan Kauffman.

Bain, M. and Muggleton, S.: 1990, Non-monotonic learning, *in* J. E. Hayes-Michie and E. Tyugu (eds), *Machine Intelligence 12*, Oxford University Press.

Banerji, R. B.: 1991, Learning theoretical terms, *in* S. Muggleton (ed.), *Proceedings of the International Workshop on Inductive Logic Programming ILP-91.*

Bergadano, F., Gunetti, D. and Trinchero, U.: 1993, The difficulties of learning logic programs with cut, *Journal of Artificial Intelligence Research* **1**, 91–107.

Bol, R. N., Apt, K. R. and Klop, J. W.: 1991, An analysis of loop checking mechanisms for logic programs, *Theoretical Computer Science* **86**, 35–79.

Bratko, I.: 1990, *Prolog Programming for Artificial Intelligence*, 2nd edn, Addison-Wesley.

Brazdil, P. and Clark, P.: 1990, Learning from imperfect data, *in* Brazdil and Konolige (eds), *Machine Learning, Meta-Reasoning and Logics*, Kluwer.

Buntine, W.: 1988, Generalized subsumption and its application to induction and redundancy, *Artificial Intelligence* **36**, 149–176.

Cain, T.: 1991, DUCTOR: A theory revision system for propositional domains, *in* L. A. Birnbaum and G. C. Collins (eds), *Machine Learning: Proceedings of the Eighth International Workshop*, Morgan Kaufmann.

Cameron-Jones, R. M. and Quinlan, J. R.: 1993, Avoiding pitfalls when learning recursive theories, Available by ftp at ftp.cs.su.oz.au, apparently submitted to IJCAI 93.

Carbonell, J. G. and Gil, Y.: 1987, Learning by experimentation, *in* P. Langley (ed.), *Proceedings of the Fourth International Workshop on Machine Learning*, Morgan Kaufmann, California.

Carbonell, J. G. and Gil, Y.: 1990, Learning by experimentation: The operator refinement method, *in* Y. Kodratoff and R. Michalski (eds), *Machine Learning: An Artificial Intelligence Approach Volume III*, Morgan Kaufmann.

Cavedon, L.: 1991, Acyclic logic programs and the completeness of SLDNF-resolution, *Theoretical Computer Science* **86**, 81–92.

Chan, D.: 1988, Constructive negation based on the completed database, *in* R. A. Kowalski and A. Bowen (eds), *Logic Programming: Proceedings of the Fifth International Conference and Symposium*, MIT Press, pp. 111–125.

Charniak, E. and McDermott, D.: 1986, *Introduction to Artificial Intelligence*, Addison-Wesley.

Cheng, P. C.-H.: 1991, Modelling experiments in scientific discovery, *in* J. Mylopoulos and R. Reiter (eds), *Proceedings of the Twelfth International Joint Conference on Artificial Intelligence*, Vol. 2, Morgan Kaufmann, California.

Clocksin, W. F. and Mellish, C. S.: 1981, *Programming in Prolog*, Springer.

Covington, M. E.: 1985a, Eliminating unwanted loops in Prolog, *ACM SIGPLAN Notices* **20**(1), 20–26.

Covington, M. E.: 1985b, A further note on looping in Prolog, *ACM SIGPLAN Notices* **20**(8), 28–31.

Craw, S. and Sleeman, D.: 1991, The flexibility of speculative refinement, *in* L. A. Birnbaum and G. C. Collins (eds), *Machine Learning: Proceedings of the Eighth International Workshop*, Morgan Kauffman.

Davis, R.: 1979, Interactive transfer of expertise: Acquisition of new inference rules, *Artificial Intelligence* **12**, 121–157.

De Raedt, L.: 1992, *Interactive Theory Revision: An Inductive Logic Programming Approach*, Academic Press.

De Raedt, L. and Bruynooghe, M.: 1992, Interactive concept learning and constructive induction by analogy, *Machine Learning.*

De Raedt, L., Bruynooghe, M. and Martens, B.: 1991, Integrity constraints and interactive concept learning, *in* L. A. Birnbaum and G. C. Collins (eds), *Machine Learning: Proceedings of the Eighth International Workshop*, Morgan Kauffman.

De Raedt, L., Lavrač, N. and Džeroski, S.: 1993, Multiple predicate learning, *in* S. Muggleton (ed.), *Proceedings of the International Workshop on Inductive Logic Programming ILP-93*, Josef Stefan Institute, Technical Report, Slovenia.

Dean, T. and Boddy, M.: 1988, An analysis of time-dependent planning, *Proceedings of the Seventh National Conference on Artificial Intelligence*, Morgan Kaufmann, pp. 49–54.

DeJong, G.: 1988, An introduction to explanation-based learning, *in* H. E. Shrobe and the American Association for Artificial Intelligence (eds), *Exploring Artificial Intelligence: Survey Talks from the National Conferences on Artificial Intelligence*, Morgan Kaufmann, California.

Dietterich, T. G. and Michalski, R. S.: 1986, Learning to predict sequences, *in* R. S. Michalski, J. G. Carbonell and T. M. Mitchell (eds), *Machine Learning: An Artificial Intelligence Approach Volume II*, Morgan Kaufmann.

Džeroski, S. and Bratko, I.: 1992, Handling noise in inductive logic programming, *in* S. Muggleton and K. Furukawa (eds), *Proceedings of the International Workshop on Inductive Logic Programming ILP-92*, Institute for New Generation Computer Technology, ICOT Technical Memorandum: TM-1182, Tokyo.

Edmondson, R.: 1988, Exceptions: A proposal to remedy floundering, *in* J. S. Gero and R. Stanton (eds), *Artificial Intelligence Developments and Applications*, Elsevier, pp. 183– 193.

Flavell, J. H.: 1987, *Cognitive Development*, Prentice-Hall.

Foo, N., Rao, A., Taylor, A. and Walker, A.: 1988, Deduced relevant types and constructive negation, *in* R. A. Kowalski and A. Bowen (eds), *Logic Programming: Proceedings of the Fifth International Conference and Symposium*, MIT Press, pp. 126–139.

Forbus, K.: 1984, Qualitative process theory, *Artificial Intelligence* **24**, 85–168.

Gärdenfors, P.: 1988, *Knowledge in Flux*, MIT Press, Cambridge, Massachusetts.

Georgeff, M. P. and Wallace, C. S.: 1984, A general selection criterion for inductive inference, *Technical Report 44*, Department of Computer Science, Monash University, Clayton, Victoria.

Gil, Y.: 1991, A domain-independent framework for effective experimentation in planning, *in* L. A. Birnbaum and G. C. Collins (eds), *Machine Learning: Proceedings of the Eighth International Workshop*, Morgan Kaufmann.

Ginsburg, H. and Opper, S.: 1979, *Piaget's Theory of Intellectual Development*, 2 edn, Prentice-Hall, Englewood Cliffs, NJ, USA.

Goldman, S. A. and Sloan, R. H.: 1994, The power of self-directed learning, *Machine Learning* **14**, 271–294.

Gregory, R. L. (ed.): 1987, *Oxford Companion to the Mind*, Oxford University Press.

Guha, R. and Lenat, D. B.: 1994, Enabling agents to work together, *Communications of the ACM*.

Hamakawa, R.: 1991, Revision cost for theory refinement, *in* L. A. Birnbaum and G. C. Collins (eds), *Machine Learning: Proceedings of the Eighth International Workshop*, Morgan Kauffman.

Hayes-Roth, F.: 1983, Using proofs and refutations to learn from experience, *in* R. S. Michalski, J. G. Carbonell and T. M. Mitchell (eds), *Machine Learning: An Artificial Intelligence Approach*, Tioga Publishing Company, Palo Alto, Cal., USA.

Hume, D. and Sammut, C.: 1991a, Applying inductive logic programming in reactive environments, *in* S. Muggleton (ed.), *Proceedings of the International Workshop on Inductive Logic Programming ILP-91*.

Hume, D. and Sammut, C.: 1991b, Using inverse resolution to learn relations from experiments, *in* L. A. Birnbaum and G. C. Collins (eds), *Machine Learning: Proceedings of the Eighth International Workshop*, Morgan Kaufmann.

Idestam-Almquist, P.: 1993, Generalization under implication by recursive anti-unification, *Proceedings of the Tenth International Conference on Machine Learning*, Morgan Kaufmann.

Jung, B.: 1993, On inverting generality relations, *in* S. Muggleton (ed.), *Proceedings of the International Workshop on Inductive Logic Programming ILP-93*, Josef Stefan Institute, Technical Report, Slovenia.

Kedar, S. T., Bresina, J. L. and Dent, C. L.: 1991, The blind leading the blind: Mutual refinement of approximate theories, *in* L. A. Birnbaum and G. C. Collins (eds), *Machine Learning: Proceedings of the Eighth International Workshop*, Morgan Kauffman.

Klahr, D.: 1994, Children, adults and machines as discovery systems, *Machine Learning* **14**, 313–320.

Kuhn, T. S.: 1970, *The Structure of Scientific Revolutions*, 2nd edn, University of Chicago Press, Chicago, USA.

Kulkarni, D. and Simon, H. A.: 1989, The role of experimentation in scientific theory revision, *in* A. M. Segre (ed.), *Sixth International Workshop on Machine Learning*, Morgan Kaufmann.

Lapointe, S. and Matwin, S.: 1992, Sub-unification: A tool for efficient induction of recursive programs, *in* D. Sleeman (ed.), *Machine Learning: Proceedings of the Ninth International Workshop*, Morgan Kaufmann, California.

Ling, C.: 1991, Non-monotonic specialization (preliminary version), *in* S. Muggleton (ed.), *Proceedings of the International Workshop on Inductive Logic Programming ILP-91.*

Ling, X. C. and Narayan, M. A.: 1991, A critical comparison of various methods based on inverse resolution, *in* L. A. Birnbaum and G. C. Collins (eds), *Machine Learning: Proceedings of the Eighth International Workshop*, Morgan Kauffman.

Lloyd, J. W.: 1987a, Declarative error diagnosis, *New Generation Computing* **5**(2), 133–154.

Lloyd, J. W.: 1987b, *Foundations of Logic Programming*, 2nd edn, Springer-Verlag.

Michalski, R. S.: 1983, A theory and methodology of inductive learning, *in* R. S. Michalski, J. G. Carbonell and T. M. Mitchell (eds), *Machine Learning: An Artificial Intelligence Approach*, Tioga.

Minton, S., Carbonell, J. G., Knoblock, C. A., Kuokka, D. R., Etzioni, O. and Gil, Y.: 1990, Explanation-based learning: A problem solving perspective, *in* J. Carbonell (ed.), *Machine Learning: Paradigms and Methods*, MIT Press/ Elsevier.

Mitchell, T. M.: 1982, Generalization as search, *Artificial Intelligence* **18**, 203–226.

Mooney, R. J.: 1992, Batch versus incremental theory refinement, *AAAI Spring Symposium on Knowledge Assimilation*, American Association for AI, Standford, California.

Mooney, R. J. and Ourston, D.: 1991, Constructive induction in theory refinement, *in* L. A. Birnbaum and G. C. Collins (eds), *Machine Learning: Proceedings of the Eighth International Workshop*, Morgan Kaufmann.

Morik, K., Wrobel, S., Kietz, J.-U. and Emde, W.: 1993, *Knowledge Acquisition and Machine Learning: Theory, Methods and Applications*, Academic Press.

Mozetic, I.: 1987, The role of abstractions in learning qualitative models, *in* P. Langley (ed.), *Proceedings of the Fourth International Workshop on Machine Learning*, Morgan Kaufmann.

Muggleton, S.: 1991, Inductive logic programming, *New Generation Computing* **9**(4), 295–318.

Muggleton, S.: 1992, Inverting implication, *in* S. Muggleton and K. Furukawa (eds), *Proceedings of the International Workshop on Inductive Logic Programming ILP-92*, Institute for New Generation Computer Technology, ICOT Technical Memorandum: TM-1182.

Muggleton, S.: 1994, Predicate invention and utilization, *Journal of Experimental and Theoretical Artificial Intelligence* **6**(1), 121–130.

Muggleton, S. and Buntine, W.: 1988, Machine invention of first-order predicates by inverting resolution, *in* J. Laird (ed.), *Proceedings of the Fifth International Conference on Machine Learning*, Morgan Kaufmann.

Muggleton, S. and Feng, C.: 1990, Efficient induction of logic programs, *Proceedings of the First Conference on Algorithmic Learning Theory*, Tokyo.

Muggleton, S. and Page, D.: 1994, Self-saturation of definite clauses, *in* S. Wrobel (ed.), *Proceedings of the International Workshop on Inductive Logic Programming ILP-94*, Gesellschaft für Mathematik und Datenverarbeitung mbH, Technical Report 237, D-53754 Sankt Augustin, Germany.

Muggleton, S. and Quinlan, R.: 1993, personal communication.

Muggleton, S., Srinivasan, A. and Bain, M.: 1992, Compression, significance and accuracy, *in* D. Sleeman (ed.), *Machine Learning: Proceedings of the Ninth International Workshop*, Morgan Kaufmann, California.

Naish, L.: 1985, Automating control for logic programs, *Journal of Logic Programming* **3**, 167–183.

Naish, L.: 1986, *Negation and Control in PROLOG*, Springer-Verlag.

Naish, L.: 1992, Declarative diagnosis of missing answers, *New Generation Computing* **10**(3), 255–285.

Nayak, A., Pagnucco, M., Foo, N. and Kwok, R.: 1995, Entrenchment and retractability: A preliminary report, *in* X. Yao (ed.), *Eighth Australian Joint Conference on Artificial Intelligence*, World Scientific, pp. 219–226.

Nienhuys-Cheng, S. H. and Flach, P. A.: 1991, Consistent term mappings, term partitions, and inverse resolution, *in* Y. Kodratoff (ed.), *Machine Learning — EWSL-91 Proceedings*, Springer-Verlag.

Nute, D.: 1985, A programming solution to certain problems with loops in Prolog, *ACM SIGPLAN Notices* **20**(8), 32–37.

O'Rorke, P., Morris, S. and Schulenburg, D.: 1989, Theory formation by abduction: Initial results of a case study based on the chemical revolution, *in* A. M. Segre (ed.), *Sixth International Workshop on Machine Learning*, Morgan Kaufmann.

Paakki, J.: 1994, Effective algorithmic debugging for inductive logic prgramming, *in* S. Wrobel (ed.), *Proceedings of the International Workshop on Inductive Logic Programming ILP-94*, Gesellschaft für Mathematik und Datenverarbeitung mbH, Technical Report 237, D-53754 Sankt Augustin, Germany.

Pazzani, M. J.: 1989, Detecting and correcting errors of omission after explanation-based learning, *in* N. S. Sridharan (ed.), *Eleventh International Joint Conference on Artificial Intelligence*, Morgan Kaufmann.

Pazzani, M. J., Brunk, C. A. and Silverstein, G.: 1991, A knowledge-intensive approach to learning relational concepts, *in* L. A. Birnbaum and G. C. Collins (eds), *Machine Learning: Proceedings of the Eighth International Workshop*, Morgan Kauffman.

Pereira, L. M. and Porto, A.: 1982, Selective backtracking, *in* Clark and Tarnlund (eds), *Logic Programming*, Academic Press.

Plotkin, G. D.: 1970, A note on inductive generalisation, *in* B. Meltzer and D. Michie (eds), *Machine Intelligence 5*, Elsevier North-Holland.

Plotkin, G. D.: 1971a, *Automatic Methods of Inductive Inference*, PhD thesis, University of Edinburgh, Scotland.

Plotkin, G. D.: 1971b, A further note on inductive generalisation, *in* B. Meltzer and D. Michie (eds), *Machine Intelligence 6*, Elsevier North-Holland.

Plümer, L.: 1990, *Termination Proofs for Logic Programs*, number 446 in *Lecture Notes in Artificial Intelligence*, Springer.

Politakis, P. G.: 1985, *Empirical Analysis for Expert Systems*, Pitman.

Poole, D.: 1993, Logic programming, abduction and probability — a top-down anytime algorithm for estimating prior and posterior probabilities, *New Generation Computing* **11**, 377–400.

Poole, D. and Goebel, R.: 1985, On eliminating loops in Prolog, *ACM SIGPLAN Notices* **20**(8), 38–40.

Popper, K.: 1969a, *Conjectures and Refutations*, 3rd edn, Routledge and Kegan Paul, London.

Popper, K.: 1969b, Science: Conjectures and refutations, *Conjectures and Refutations*, 3rd edn, Routledge and Kegan Paul, London, chapter 1.

Quinlan, J. R.: 1990, Learning logical definitions from relations, *Machine Learning* **5**(3), 239–266.

Quinlan, J. R.: 1991, Determinate literals in inductive logic programming, *in* J. Mylopoulos and R. Reiter (eds), *Proceedings of the Twelfth International Joint Conference on Artificial Intelligence*, Vol. 2, Morgan Kaufmann, California.

Rajamoney, S. A.: 1989, Exemplar-based theory rejection: An approach to the experience consistency problem, *in* A. M. Segre (ed.), *Sixth International Workshop on Machine Learning*, Morgan Kaufmann.

Richards, B. and Mooney, R.: 1991, First-order theory revision, *in* L. A. Birnbaum and G. C. Collins (eds), *Machine Learning: Proceedings of the Eighth International Workshop*, Morgan Kaufmann.

Richards, B. and Mooney, R.: 1992, Learning relations by pathfinding, *Proceedings of the Tenth National Conference on Artificial Intelligence*, AAAI Press/ MIT Press.

Richards, B. L. and Mooney, R. J.: 1995, Automated refinement of first-order horn-clause domain theories, *Machine Learning* **19**(2), 95–131.

Rissanen, J.: 1978, Modeling by shortest data description, *Automatica* **14**, 465–471.

Rissanen, J.: 1983, A universal prior for integers and estimation by minimum description length, *Annals of Statistics* **11**(2), 416–431.

Robinson, J. A.: 1965, A machine oriented logic based on the resolution principle, *Journal of the ACM* **12**(1), 23–41.

Rouveirol, C.: 1991a, ITOU: Induction of first order theories, *in* S. Muggleton (ed.), *Proceedings of the International Workshop on Inductive Logic Programming ILP-91*.

Rouveirol, C.: 1991b, Semantic model for induction of first order theories, *in* J. Mylopoulos and R. Reiter (eds), *Proceedings of the Twelfth International Joint Conference on Artificial Intelligence*, Vol. 2, Morgan Kaufmann.

Rouveirol, C.: 1994, Flattening and saturation: Two representation changes for generalisation, *Machine Learning* **14**, 219–232.

Rouveirol, C. and Puget, J. F.: 1990a, Beyond inversion of resolution, *Proceedings of the Seventh International Conference on Machine Learning*, Morgan Kaufmann.

Rouveirol, C. and Puget, J.-F.: 1990b, A simple solution for inverting resolution, *in* K. Morik (ed.), *EWSL-89: Proceedings of the Fourth European Working Session on Learning*, Pitman and Morgan Kaufmann.

Russell, S. J.: 1989, *The Use of Knowledge in Analogy and Induction*, Morgan Kaufmann.

Salzberg, S., Delcher, A., Heath, D. and Kasif, S.: 1991, Learning with a helpful teacher, *in* J. Mylopoulos and R. Reiter (eds), *Proceedings of the Twelfth International Joint Conference on Artificial Intelligence*, Morgan Kaufmann.

Sammut, C.: 1981a, Concept learning by experiment, *in* A. Drinan (ed.), *Proceedings of the Seventh International Joint Conference on Artificial Intelligence*, Morgan Kaufmann.

Sammut, C.: 1981b, *Learning Concepts by Performing Experiments*, PhD thesis, University of NSW, Sydney, Australia.

Sammut, C. and Banerji, R. B.: 1986, Learning concepts by asking questions, *in* R. S. Michalski, J. G. Carbonell and T. M. Mitchell (eds), *Machine Learning: An Artificial Intelligence Approach Volume II*, Morgan Kaufmann.

Shapiro, E. Y.: 1981, Inductive inference of theories from facts, *Research Report 192*, Yale University, USA.

Shapiro, E. Y.: 1982, Algorithmic program debugging, *Research Report 237*, Yale University, USA.

Silverstein, G. and Pazzani, M. J.: 1991, Relational cliches: Constraining constructive induction during relational learning, *in* L. A. Birnbaum and G. C. Collins (eds), *Machine Learning: Proceedings of the Eighth International Workshop*, Morgan Kauffman.

Sleeman, D. H.: 1983, Inferring student models for intelligent computer-aided instruction, *in* R. S. Michalski, J. G. Carbonell and T. M. Mitchell (eds), *Machine Learning: An Artificial Intelligence Approach*, Tioga.

Sleeman, D., Hirsh, H., Ellery, I. and Kim, I.-Y.: 1990, Extending domain theories: Two case studies in student modeling, *Machine Learning* **5**(1), 11–37.

Smith, D. E., Genesereth, M. R. and Ginsberg, M. L.: 1986, Controlling recursive inference, *Artificial Intelligence* **30**, 343–389.

Stahl, I. and Weber, I.: 1994, The arguments of newly invented predicates in ILP, *in* S. Wrobel (ed.), *Proceedings of the International Workshop on Inductive Logic Programming ILP-94*, Gesellschaft für Mathematik und Datenverarbeitung mbH, Technical Report 237, D-53754 Sankt Augustin, Germany.

Stepp, R. E., Whitehall, B. L. and Holder, L.: 1988, Towards intelligent machine learning algorithms, *in* Y. Kodratoff (ed.), *ECAI 88 Proceedings*, Pitman, London.

Sterling, L. and Shapiro, E.: 1986, *The Art of Prolog: Advanced Programming Techniques*, MIT Press, Cambridge, Massachusetts.

Stickel, M.: 1984, A prolog technology theorem prover, *New Generation Computing* **2**, 371–83.

Tambe, M., Newell, A. and Rosenbloom, P. S.: 1990, The problem of expensive chunks and its solution by restricting expressiveness, *Machine Learning* **5**(3), 299–348.

Taylor, K.: 1993, Inverse resolution of normal clauses, *in* S. Muggleton (ed.), *Proceedings of the International Workshop on Inductive Logic Programming ILP-93*, Josef Stefan Institute, Technical Report, Slovenia.

Thom, J. A. and Zobel, J.: 1990, NU-Prolog reference manual, *Technical Report 86/10*, Department of Computer Science, University of Melbourne.

Van-Lehn, K.: 1990, *Mind Bugs: The Origins of Procedural Misconceptions*, MIT Press.

Vere, S.: 1977, Induction of relational productions in the presence of background information, *Fifth International Joint Conference on Artificial Intelligence*, Morgan Kaufmann.

Wilkins, D. C. and Tan, K.-W.: 1989, Knowledge base refinement as improving an incorrect, inconsistent and incomplete domain theory, *in* A. M. Segre (ed.), *Sixth International Workshop on Machine Learning*, Morgan Kaufmann.

Wogulis, J.: 1991, Revising relational domain theories, *in* L. A. Birnbaum and G. C. Collins (eds), *Machine Learning: Proceedings of the Eighth International Workshop*, Morgan Kaufmann.