INFORMATION TO USERS

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

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

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

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

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

UMI

A Bell & Howell Information Company
300 North Zeeb Road, Ann Arbor MI 48106-1346 USA
313/761-4700 800/521-0600

Université d'Ottawa · University of Ottawa

Learning Recursive Definitions in Prolog

Riverson Rios

A Thesis
Submitted to the School of Graduate Studies and Research
of the University of Ottawa in partial fulfillment of the requirements
for the degree of
Doctor of Philosophy in Computer Science
under the auspices of the Ottawa-Carleton Institute
of Computer Science

School of Information Technology and Engineering,
University of Ottawa,
Ottawa, Ontario,
Canada

© Riverson Rios
January 1998

National Library of Canada

Bibliothèque nationale du Canada

Acquisitions and Bibliographic Services

Acquisitions et services bibliographiques

395 Wellington Street
Ottawa ON K1A 0N4
Canada

395, rue Wellington
Ottawa ON K1A 0N4
Canada

Your file Votre référence

Our file Notre référence

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

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

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

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

0-612-28371-2

Canada

À ma femme.

Abstract

Inductive Logic Programming (ILP) is one of the new and fast growing sub-fields of artificial intelligence. Given a specification language, the goal is to induce a logic program from examples of how the program should work (and also of how it should *not* work). One main difficulty of ILP lies in learning recursively defined predicates. Today's systems strongly rely on a set of supporting predicates known as the background knowledge that helps define the recursive clause. The dependence on background knowledge has its drawbacks in that it is assumed that the user knows in advance what sort of predicates are required by the target definition. Predicate invention, a research topic that has received a lot of attention lately, can remedy the situation by extending the specification language with new concepts, which appear neither in the examples nor in the background knowledge, and finding a definition for them. A serious concern is that no examples of the invented predicate are explicitly given but rather of the target predicate, so learning has to be done in the absence or scarcity of examples.

This research is concerned with the problem of learning recursive definitions based on inverting clausal implication from a small data set. The aim is both to derive an autonomous learning method that can invent the recursive predicates it needs, and to implement it in an efficient manner. Experiments show that the system is capable of finding a correct definition of many relations by inventing the necessary predicates, but does not perform very well on random examples. A comparison between several similar systems that learn recursive definitions of a single predicate is shown. We also show the need for system-generated negative examples and discuss several pitfalls of predicate invention and the absence/scarcity of examples.

Acknowledgment

The completion of this thesis is due to a number of people whose help I gratefully acknowledge. Forgive me for any unintended omissions.

First of all, I am most grateful to my supervisor, Prof Stan Matwin, whom I can call a friend. I would like to say "thank you" for his friendship, guidance and support, for our long and frequent discussions on several fields. His contagious enthusiasm for research, knowledge of so many areas and sense of professionalism have definitely contributed to the success of my stay at the University of Ottawa, home of the Gee Gees. His careful reading of uncountable drafts of this thesis has improved the contents and the presentation many folds.

I must also mention Prof. Tarcísio Pequeno who fourteen years ago strongly encouraged me to follow a graduate degree in a way that would, forever, change the course of my life.

Thanks also go to the designers of several systems discussed in this work for patiently answering several probing questions like Eduardo Morales, Lionel Martin and Céline Rouveirol. Particularly, I would like to thank Stéphane Lapointe and David Aha for kindly providing us with the source code of their systems and helping us better understand their methods. Thanks also go to Rob Holte for his useful comments in several stages of this research and to the other official examiners of the thesis, Charles Ling, Franz Oppacher and Stan Szpakowicz, for carefully reviewing this work and raising so many valuable questions.

I am indebted to my colleagues at the Federal University of Ceará for the opportunity to learn a tad more about the world, to Capes for providing the necessary financial support during the last four years, to Murilo Pereira for kindly accepting to be my proxy in Fortaleza while I was away and to many friends I made in Ottawa, especially Beatriz Perez, Bete and Eliésio Malvar, Brigitte, Bruno Marsiaj, César and Carla Teixeira, Cindy Lee, Patrick Fry, Pierre Landry, Rossana Andrade and Wagner. Many thanks go to Don and Adriana for all their great support and our nice chats, and also to my family back home for years and years of inspiration.

Finalement, je voudrais remercier avec tout mon coeur celle qui a partagé ma vie, qui *est* ma vie, Lígia Coe, ma *nen*. Pour sa présence, sa vivacité, sa joie. Pour l'encouragement, l'amitié, l'appui moral qu'elle m'a démontrés tout au long de ce projet. Pour tolérer avec un sourire mes interminables soirées et fins-de-semaine à l'école, ainsi que pour son amour et patience au cours des deux dernières années ensemble. Et spécialement pour me montrer qu'il existe encore amour dans ce monde.

Table of Contents

Table of Contents i

List of Figures vii

List of Tables ix

Index xi

Chapter 1. Introduction 1

1.1 Inductive Logic Programming 1

1.2 Difficulties 3

1.3 Predicate Invention 4

1.4 The Learning System Shrinp 6

1.5 Contributions 8

1.6 Outline 11

1.7 How to Read This Thesis 12

Chapter 2. Basic Concepts 13

2.1 The Specification Language *L* 13

2.2 Deduction Theorem, Syntactic Entailment and Unification 17

2.3 Inductive Logic Programming 21

2.4 Predicate Invention, Sub-Unification and Generating Terms 24

2.5 θ-Subsumption, Generalization, Lgg and ij-Determinism 28

Chapter 3. Theoretical Background 33

3.1 ILP as a Generalization Problem 33

3.2 The Resolution Principle 36

3.3 Inverting Resolution 39
3.3.1 The V operators 40
3.3.1.1 The absorption operator 40
3.3.1.2 The identification operator 41
3.3.2 The W operators 42
3.3.2.1 The intra-construction operator 43
3.3.2.2 The inter-construction operator 44
3.3.3 Saturation 44
3.3.4 Other operators 46
3.4 Inverse Entailment 46
3.5 Inverse Implication 48
3.5.1 LOPSTER 50
3.5.2 Crustacean 52
3.5.2.1 An example 54
3.6 Discussion 58
Chapter 4. Related Work 60
4.1 Smart 60
4.2 SKILit 62
4.3 TIM 64
4.4 Chao's System 65
4.5 Force2 66
4.6 Champ 68
4.7 SIERES 70
4.8 SPILP 71
4.9 Discussion 72

4.9.1 Learning the base clause 72

4.9.2 Learning the recursive clause 73

4.9.3 Pruning the hypothesis space 74

4.10 Learning in the Absence of Negative Examples 75

4.11 Summary 79

Chapter 5. CLAM and Shrinp 80

5.1 CLAM 80

5.11 An example: learning the predicate *sum* 81

5.2 Shrinp 82

5.3 The modules 84

5.3.1 The Inducer Module 88

5.3.2 The Filter Module 89

5.3.2.1 The Left-recursive Filter 90

5.3.2.2 The Duplicate Filter 90

5.3.2.3 Superfluous-argument Filter 90

5.3.2.4 The Coverage Tester 91

5.3.3 The Predicate Inventor Module 91

5.4 The automatic generation of negative examples 93

5.4.1 The GENEX algorithm 95

5.4.1.1 The *type* mode 95

5.4.1.2 The *input determinacy* mode 96

5.4.1.3 The *input-output determinacy* mode 96

5.4.2 When not to use GENEX 97

5.4.2.1 Problems with the *io-det* mode 97

5.4.2.2 Problems with the algorithm itself 99

5.4.3 Using base clauses as negative examples 100

5.4.3.1 An example: learning a base clause for *minus* 101

5.4.3.2 Pitfalls 103

5.5 Relearning the Base Clause 105

5.5.1 An example: learning a definition for the relation *Permut* 107

5.6 Parameters and Options 107

5.6.1 Parameters 108

5.6.1.1 Maximum depth (maxdep) 108

5.6.1.2 The number of positive examples (npos) 108

5.6.2 Options 109

5.6.2.1 BADARG 109

5.6.2.2 BCNEG 109

5.6.2.3 GENEX 110

5.6.2.4 GMODE 110

5.6.2.5 NEGUSER 110

5.6.2.6 RELEARNBC 110

5.6.2.7 SHRINP 110

5.6.3 Defaults 110

5.7 Complexity Analysis 111

5.7.1 The complexity of the search for the base clause 113

5.7.2 The complexity of the search for the recursive clause 113

5.8 Discussion 114

Chapter 6. Predicate Invention 119

6.1 The Predicate Invention Task 119

6.1.1 Finding the arguments of the new predicate 122

6.2 Predicate Invention in Shrinp 123
6.2.1 Language bias 124
6.2.2 Argument structure 125
6.2.2.1 Constants 126
6.2.2.2 Repeated arguments 127
6.2.2.3 New variables 127
6.3 Argument-Binding Chains for Learning RRDs 128
6.4 Argument-Binding Chains for Learning LRDs 131
6.5 Finding Examples for the New Predicate 132
6.5.1 Problems with unnamed variables 133
6.5.2 Problems with using negative examples of the target predicate 134
6.5.3 Problems with the argument structure in the head 134
6.6 Criteria for Stopping 135
6.7 Discussion 136
Chapter 7. Experimental Results 139
7.1 Testing Shrinp on Some Predicates 139
7.1.1 A definition that is not learned 147
7.2 Experiments with GENEX 148
7.2.1 Running GENEX on two learners 149
7.2.2 Running GENEX on Shrinp 151
7.3 Evaluating the Options 154
7.4 Experiments with Random Examples 158
7.5 Experiments with CLAM 161
7.5.1 Comparison with Golem and Progol 162
7.5.2 Learning Curve 163

7.6 Discussion 164

Chapter 8. Conclusion 166

8.1 Extensions to the Method and Future Work 168

Chapter 9. References 174

Appendix A. Computing an lgg under Implication 188

Appendix B. A Session with Shrinp 193

B.1 Learning a definition on the fly 194

B.2 Demos 194

B.3 Changing options and parameters 197

List of Figures

Chapter 1

Fig 1.1 A directional graph. . . . 2

Chapter 2

Fig 2.1 t_e is an embedding term of t_1 in t_2 . . . 26

Chapter 3

Fig 3.1 Strength is inversely proportional to suitability for generalization . . . 35

Fig 3.2 R is the resolvent of C1 and C2. . . . 37

Fig 3.3 A *linear derivation* in which a clause C is recursively used several times . . . 38

Fig 3.4 The absorption operator. . . . 41

Fig 3.5 The identification operator . . . 42

Fig 3.6 The W operator. . . . 43

Fig 3.7 The intra-construction operator builds two clauses for the new predicate z . . . 43

Fig 3.8 The inter-construction operator builds one clause for the new predicate z. . . . 44

Fig 3.9 The recursive clause C resolves with two examples in different resolution chains. . . . 53

Chapter 4

Fig 4.1 SKILit's algorithm is based on four nested loops (levels). . . . 62

Fig 4.2 An example of the set of rules that SKILit uses as a language bias. . . . 63

Chapter 5

Fig 5.1 Shrinp is a descendent of three other learners. . . . 83

Fig 5.2 Shrinp's modules. . . . 85

Fig 5.3 To learn a definition of *p*, Shrinp invents two necessary predicates. . . . 86

Fig 5.4 The Inducer Module. . . . 88

Fig 5.5 The Filter Module. . . . 89

Fig 5.6 The Predicate Inventor Module. . . . 92

Fig 5.7 GENEX algorithm. . . . 98

Fig 5.8 The BCs are obtained by lgging the subterms of each group. . . . 101

Fig 6.1 Shrinp's language bias. . . . 125

Chapter 6

Fig 6.2 Types of argument-binding chains to learn right-recursive definitions. 128

Fig 6.3 Types of argument-binding chains to learn left-recursive definitions. 131

Chapter 7

Fig 7.1 Time reduction in percentage for learning PRDs with *type* and *i-det* modes. 153

Fig 7.2 Time reduction in percentage for learning LRDs and RRDs with *type* and *i-det* modes. 154

Fig 7.3 CLAM vs. Golem vs. Progol. Accuracy with five positive and ten negative examples 162

Fig 7.4 Learning curve for CLAM 164

Chapter 8

Fig 8.1 Learning a disjunctive definition. 170

List of Tables

Chapter 2

Table 2.1. The truth value of several formulas. 18

Table 2.2. Subterms, embedding terms and generating terms of *pair(s(0), pair(s(s(0)),[]))* 27

Table 2.3. The lgg of several terms under θ-subsumption 30

Chapter 3

Table 3.1. Existence of the *lgg* under three operations. 36

Table 3.2. Background knowledge, example and programs induced through IE. 47

Table 3.3. Examples of *length*. 54

Table 3.4. Cross-product of the GTs and subterms of the arguments of E+1 55

Table 3.5. Cross-product of the GTs and subterms of the arguments of E+2 55

Table 3.6. Some of the matching GTs from E+1 and E+2 and the arguments A1 and A2 of the potential base clauses they derive. 56

Table 3.7. Computing the head of the recursive clause for *length([],0)* 57

Chapter 4

Table 4.1. The result of the use of SPILP's simplification rules on two sets of definitions 72

Table 4.2. Comparison of systems that learn specifically recursive definitions of a predicate. 76

Chapter 5

Table 5.1. Examples and background knowledge for *sum* 81

Table 5.2. Definitions generated by CLAM when learning the predicate *sum* 82

Table 5.3. Subterms and potential base clauses for the predicate *minus* 103

Table 5.4. Some subterms and potential base clauses for the predicate *prefix*. 105

Table 5.5. Definition induced for *permut* before and after its base clause is relearned 108

Table 5.6. Default values for the parameters. 111

Table 5.7. Default values for the options. 112

Chapter 6

Table 6.1. Possible completions for a definition of *lEven*. 126

Table 6.2. Positive examples found for *newp1* and *newp2* 126

Table 6.3. Predicates invented through the use of the RRD argument chains 130

Table 6.4. Predicates invented through the use of the LRD argument chains 132

Table 6.5. Examples and incomplete clause for the predicate *sum* 133

Chapter 7

Table 7.1. A list of predicates and their modes. 140

Table 7.2. Positive and negative examples used in this experiment 141

Table 7.3. Definitions of the previous relations learned by Shrinp from given examples 144

Table 7.4. Golem's and Progol's accuracy and runtime on different sets of negative examples. 150

Table 7.5. Number of times out of ten that a correct definition was found alone. 151

Table 7.6. Number of final definitions learned by Shrinp given different modes. 152

Table 7.7. Groups of options used in the experiment. Option "d" is active in all groups. 155
(a = GMODE, b = GENEX, c = NEGUSER, e = BCNEG, f = RELEARNBC, g = BADARG). . . .

Table 7.8. Accuracy, runtime and correctness of the use of single options. 156
(a = GMODE, b = GENEX, c = NEGUSER, e = BCNEG, f = RELEARNBC, g = BADARG). . . .

Table 7.9. Accuracy, runtime and correctness of the use of multiple options. 157
(a = GMODE, b = GENEX, c = NEGUSER, e = BCNEG, f = RELEARNBC, g = BADARG). . . .

Table 7.10. Accuracy, correctness and runtime on different numbers of positive examples. 160

Table 7.11. A minimal training set for testing CLAM on four predicates 162

Table 7.12. Number of times a definition was induced by each learner out of ten runs 163

Index

Absorption 40
Active variables 61
Approximate rlgg computation 65
Argument-binding chain 124
Atoms 15
Background knowledge 21
Base clause 16
Basic representative set 23
Bottom set 23
BRS 23
Clause 15
Closed-World Assumption 22
Consistency 18
Constant-depth determinate clauses 125
Constructive learning 4
Coverage 20
Coverage test 23
Cross-validation 23
CWA 22
Deduction Theorem 19
Definition 16
Embedding terms 26
Empty clause 16
Expansion 49
Extensional elimination 172
Formulas 15
Generalization 28
Generating terms 26
GENEX algorithm 95
Ground clause 16
Horn clause 15
Hypothesis language 21
Hypothesis space 22
Identification 41
I-det mode 96
ij-Determinism 31
Inconsistency Theorem 19
Inductive Logic Programming 21
Inference rules 19
Input sub-unification 52
Integrity constraints 22
Intentional elimination 172
Inter-construction 44
Interpretation 17
Intra-construction 43
Inverse entailment 46
Inverse implication 48
Inverse resolution 39
Inverse substitution 20
Io-det mode 96
Language bias 24
Learner 21
Least general generalization 29
Left-recursive definition 17
lgg 29
Literals 15
Logic program 16
Logical consequence 18
Logical equivalence 19
LRD 17
Models 18
Modes 17
N-bound recursive clause 61
Necessary predicates 25

Negative examples 21
Options 109
Or-introduction 49
Output Completion Assumption 77
Parameters 108
Positive examples 21
PRD 17
Predicate invention 24
Predicate invention task 120
Predicates 14
Proof 20
Purely recursive definition 17
θ-Subsumption 28
Range-restricted clauses 79
Recursive clause 16
Recursive definition 16
Relational clichés 62
Relative least general generalization 31
Resolution chain 38
Resolution principle 36
Right-recursive definition 17
rlgg 31
RRD 17
Satisfiability 18
Saturation 44
Shrinp 80
Sketch clauses 62
Sub-clauses 28
Substitution 20
Subterms 26
Sub-unification 26
Syntactic entailment 19
Tautology 18
Terms 14
Testing set 22
T-implication 35
Training set 22
Type mode 95
Type specification 115
Unification 20
Unit clause 16
Useful predicates 24
variable literals 62

chapter **one**

1. Introduction

This chapter provides the motivation for the present research project. It starts by introducing Inductive Logic Programming, discussing its strength and the difficulties associated with it. The importance of Predicate Invention, another main topic in this work, to ILP is also presented. An outline of the whole work and suggestions on how to read it are also shown.

1.1 Inductive Logic Programming

Inductive Logic Programming (ILP) is one of the new and fast growing sub-fields in artificial intelligence ([Muggleton 92b], [Muggleton, De Raedt 94]). ILP lies somewhere between machine learning, from which it inherits an empirical approach, and logic programming, from which its theoretic foundation and representation are derived. Given a specification language, the goal is to induce a logic program from examples of how the program should work (and also of how it should *not* work). The potential of this emerging field of study has already been proven, considering the number of successful uses of ILP in domains ranging from finite-element mesh design ([Dolšak, Muggleton 92]) and a satellite fault diagnosis model ([Feng 92]) to drug design ([Muggleton, King, Sternberg 92]), deterministic parsers ([Zelle, Mooney 94]), musicology ([Dovey 95]) and others ([Muggleton 94]). An introduction to ILP is provided in [Muggleton 90], [Lavrač, Džeroski 94], [Lavrač, Džeroski, Grobelnik 91] and more recently in [Bergadano, Gunetti 96]. Several ILP applications within the European Union-sponsored ESPRIT III Project

6020 are reviewed in [Džeroski, Bratko 96].

The expressive power of ILP as opposed to classical concept learning techniques can be perceived through an example given by Ross Quinlan ([Quinlan 90]). He discusses how the concept of a pair of connected nodes in a directional graph as the one shown in Fig. 1.1 can be learned through the use of the traditional attribute-value approach ([Quinlan 86]). To describe the network, one could set a limit to the maximum number of nodes, say ten, and to the maximum number of links from a given node, say three. Any graph could then be represented in terms of thirty attributes: the three first corresponding to the nodes to which node *a* is linked, the next three to those nodes to which node *b* is connected, and so on. But, now, how to represent the concept "*two nodes not necessarily directly linked to each other*?" Any representation making use of those attributes would look awkward and extremely lengthy.

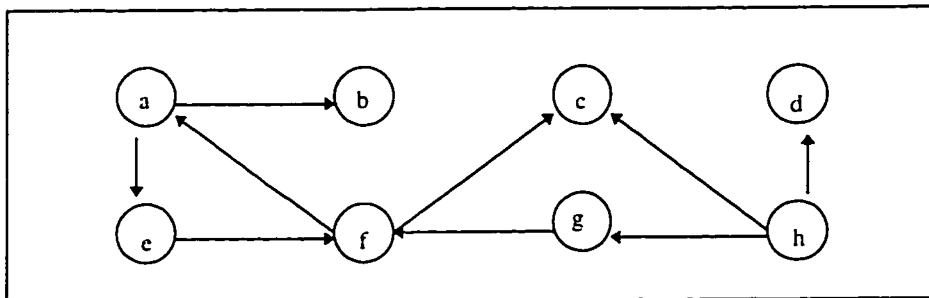


Fig 1.1. A directional graph.

On the other hand, logic languages have a nice and natural way of representing this kind of network: the use of a set of relations. For instance, the predicate *linked_to* could describe the nodes that are directly connected to each other. For the graph in Fig. 1.1, this would mean such relations as:

linked_to (a,b).
linked_to (a,e).
linked_to (e,f).

etc., assuming a closed world, that is, assuming that whichever is not explicitly stated is supposed to be false. The number of relations would correspond to the number of edges in the graph, nine in this example. Given this representation, the concept "*two nodes not necessarily directly linked to each other*" would simply be intensionally defined by the right-recursive

predicate *can_reach* below:

can_reach (A,B) :- linked_to (A,B).
can_reach (A,B) :- linked_to (A,C), can_reach (C,B).

The relation *can_reach* reads as: "two nodes are linked to each other if they are either directly connected or if the first one is directly connected to another node that is linked to the second." This example shows how much more powerful a recursive representation in first-order logic is when compared to the propositional attribute-value approach. By using the rich formalism of computational logic for both hypothesis and examples, ILP can overcome the limitations of the classical machine learning approach at the same time that it offers an ideal tool for manipulating theories through a simple addition or removal of literals and clauses.

1.2 Difficulties

One main difficulty of ILP lies in learning recursively defined predicates. As illustrated in the previous example, recursion is the principal control structure in logic programming languages such as Prolog ([Sterling, Shapiro 86]). Since terms[1], the most important data structure in those languages, are also recursively defined, it is natural that the predicates that transform them be recursive as well. The important issues in learning such predicates concern the number of base and recursive clauses, the number of recursive calls, their placement in the clause, and the choice of predicates and arguments to include in the recursive clauses. Today's systems strongly rely on a set of supporting predicates (known as the *background knowledge*) that help define the recursive clause. In the previous example, the auxiliary predicate *linked_to* was chosen to form a linear right-recursive definition of *can_reach*.

The dependence on background knowledge has its drawbacks. It assumes that the user knows in advance what sorts of predicates are to be required by the target definition. While a lack of the essential predicates results in an ineffective learning, an excess of (useless or irrelevant) predicates inevitably leads to a combinatorial explosion of the hypothesis space, thereby degrading the performance of the system. Moreover, care must be taken with respect to the

[1]Please see Chapter 2 for the definition of this and other concepts discussed here.

correctness and structure of the information in the background knowledge. For some systems, even the order of the predicates can make a difference. This problem, better known as the *background knowledge usage bottleneck*, is studied in more details in [Flener, Popelínský 94].

Another problem concerns the *implication* operation. The ILP task of finding a program that behaves like the given positive examples (and *not* like the negative examples) can be seen as a *generalization task*. Since the induction of a program is defined in terms of a logical consequence (see Section 2.3), it is natural to use implication as the sole basis for generalization. The ILP task can, therefore, be "simplified" to finding a program that implies the set of given examples. Unfortunately, though, it is not a straightforward task to invert implication. Not only is implication undecidable between clauses ([Schmidt-Schauß 88]) but also between Horn clauses ([Marcinkowski, Pacholski 92]). Stronger forms of relations have been proposed such as *T-implication* ([Idestam-Almquist 93a]) and *θ-subsumption* ([Plotkin 71]). The problem with them is that either they are incomplete or do not guarantee that the result of a generalization will be a Horn clause.

From a practical point of view, the current learners that are based on inverting clausal implication (see Chapters 3 and 4) have serious limitations. The system LOPSTER ([Lapointe 92] and [Lapointe, Matwin 92]) requires that either the base clause or examples in the same resolution chain be given. Crustacean ([Aha, Lapointe, Ling, Matwin 94a and 94b]) relaxes this requirement but limits itself to pure recursion. CLAM ([Rios, Matwin 96a and 96b]) extends Crustacean by allowing left recursion too but is still dependent on the background knowledge. Finally, Smart ([Mofizur, Numao 95 and 96]) and TIM ([Idestam-Almquist 96]) are restricted to n-bound and right-recursive clauses, respectively. As shown later in Section 4.9, the inverse implication-based learners have not yet achieved results that are at the same level of those obtained by other specialized systems that learn exclusively recursive definitions of a single predicate.

1.3 Predicate Invention

There are situations when a fixed initial specification language is not enough for the learning task at hand to be successfully completed. *Constructive Learning* is the capacity to automatically

extend the specification language by adding new non-observational representation primitives ([Lapointe, Ling, Matwin 93]). This idea is today an active research topic in ILP where it basically involves the invention of new predicates.

Predicate Invention (PI) designates the process of:

(i) extending the specification language with concepts (denoted by predicates) that appear neither in the examples nor in the background knowledge and

(ii) finding a definition for these new concepts.

PI assumes that the choice of vocabulary for the given task has not yet been completely made and, thus, represents a crucial theoretical topic in learning. Once full predicate invention is efficiently attained, the background knowledge can be discarded.

[Martin, Vrain 97] see two main advantages in PI:

(i) **Simplicity:** In the best case, the new predicates correspond to the relations that were missing in the specification language and which have a particular meaning in the domain;

(ii) **Completeness:** Some learning problems can be solved only if the language is widened through the invention of new predicates. In this sense, PI allows the expressiveness of the language to be augmented without seriously sacrificing its efficiency, thereby increasing the completeness of the learning process.

The second point is supported by a theorem by Stephen Kleene ([Kleene 52]). It states that, as long as the set of positive examples is recursively enumerable, any first order learning problem can be solved by finitely adding appropriate new predicates to the original language. Although the theorem must be carefully interpreted as nothing is said about the syntactic restrictions imposed on the language, there is no doubt about the importance of PI to ILP. [Stahl 95b] discusses classes of languages for which PI is most useful.

In [Lapointe, Ling, Matwin 93], the authors show that learning that involves PI is for a large part learning of recursive definitions. A serious concern is that no examples are explicitly given of the invented predicate. Only the instances of the target predicate are available. As an example, suppose that the task is to learn a definition for the target relation *doubles*. The

expression *doubles(A,B)* succeeds if and only if the ith element of the list *B* is twice the value of the ith element of the list *A*. Let us assume that the given positive examples of the target relation are all instances of *doubles* whose first list has at most three elements taken with repetition from the set {1,2,3}. There are, therefore, forty positive examples. The given negative instances are shown below:

-doubles([],[2]),
-doubles([2,3],a),
-doubles([2,1,2],[]) and
-doubles(2,4).

Suppose, in addition, that the following potential definition of *doubles* has been induced, where *newp* is the predicate that the system is trying to create:

doubles([],[]).
doubles([A|B],[C|D]) :- doubles(B,D), newp(A,C).

If the examples of *newp* are computed by instantiating the head of the recursive clause above with the positive examples of *doubles*, only three positive examples of *newp* will be found: *newp(1,2)*, *newp(2,4)* and *newp(3,6)*. And what's worse, there will be no negative evidence as none of the negative instances of the target relation match with the heads of the definition above. This means that a recursive definition of the new predicate will have to be induced from such a sparse set of positive examples only. Very often, non-dense example sets are not sufficient for a logic program to be learned when inverse resolution is used. They generally lack what is called a basic representative set ([Ling 91a]). Hence, it is essential that techniques be developed to endow systems with means to learn recursive definitions from sparse data sets.

Given the importance of PI to ILP, it is worth noticing that none of the inverse implication-based systems mentioned in Section 1.2 support constructive learning.

1.4 The Learning System Shrinp

The research reported here focuses on the problem of learning recursive definitions based on inverting implication, especially when a small, sparse data set is given. The aim is both to derive a learning method that can invent the recursive predicates it needs, and to implement it in an

efficient manner. Two main principles have guided the development of the system Shrinp ([Rios, Matwin 98]):

1) Autonomy

The main idea to keep in mind is that Shrinp was conceived to be an *autonomous* learner. That is, no input is necessary other than the examples and the modes (see Section 2.1) of the target relation. If negative examples are missing or are incomplete, it is up to Shrinp to create them. If a supporting predicate is needed, it is up to Shrinp to invent it and if no examples exist for it, it is up to Shrinp to find them. By doing so, Shrinp is completely isolated from the outside world: no background knowledge, no domain descriptions, no external help from the user, no oracle, no type system are allowed. The goal has been to develop a system that inverts implication based solely on the available data and a strong bias. This principle makes Shrinp suitable to educational and specific learning applications.

2) Scarcity of examples

Real-world examples may be scarce or hard to obtain. If Shrinp is to be used by, say, first-year students, one would not expect them to provide all possible examples of a predicate like *length* even from a small finite alphabet. Or if Shrinp is to be used on the fly, the system has to be able to deal with the unavailability or even the absence of examples. Systems like Champ ([Kijsirikul, Numao, Shimura 92]) and Golem ([Muggleton, Feng 90]) require that a basic representative set be present in the set of training examples. But when only a few examples are available, it is very unlikely that two examples will be only one single resolution step apart not to mention in the same resolution chain. Thus, the idea is to provide the system with means to work when the number of examples, notably positive ones, is kept to a minimum.

The ability to deal with few examples and to require no external aid contributes to the system's principal applications:

1) as a specialized learner

In this case, a general-purpose learner calls Shrinp as a specialized learner of simple recursive definitions that comply with Shrinp's strong language bias, as it is more efficient in such

circumstances. As pointed out in [Flener 95], a system like Shrinp can take advantage of the knowledge that the program to be learned has to be recursive. The recursive nature of necessary invented predicates fits perfectly in this conjecture. Indeed, when a general-purpose technique detects the necessity of inventing a new predicate, it seems preferable to invoke the specialized learner for such auxiliary purposes, rather than to have the general-purpose learner do the task. Chances are the general-purpose learner will do a poor job in inventing the new relation. Only in case Shrinp fails, should the general-purpose system take over the task.

2) as a tool for teaching logic programming languages

Here the idea is to let Shrinp work as a tool for *algorithm discovery* in that beginner Prolog students would devise experiments by supplying the system with different sets of positive and negative examples of a target predicate. The advantage is that the students would not have to provide a valid set of background predicates (and possibly would not know how to build one) and could limit themselves to a small number of examples on each run. It would be tedious to enter, for instance, tens or hundreds of examples for a single predicate. One should not expect a student to enter more than two or three examples nor to patiently and correctly answer dozens of question from the system. Algorithm discovery means playing with predicates, adding examples, removing examples, changing modes, experimenting, creating, learning. Shrinp's efficiency in such a task makes it ideal for an application like this.

1.5 Contributions

The main contributions of our work are outlined below:

1) The development of an efficient method of constructive learning

This work describes a learning method that is capable of learning several classes of recursive definitions based on inverse implication. The method is the first in its class (invert implication-based methods) to invent its own supporting predicates. Learning is possible even from a very small number of sparse examples that are not necessarily in the same resolution chain. No help from the user is required, except for the training examples of the predicate to be learned. The class of learnable clauses is broader than that of the majority of other systems that learn

exclusively recursive definitions from a large number of examples. The language bias is presented in Section 6.2.1. The main topic, predicate invention, is formally defined in Section 2.4 and thoroughly discussed in Chapter 6.

2) The implementation of the method

Details of the implementation of Shrinp, written in Quintus Prolog, can be found in Chapter 5. An empirical evaluation of the system is presented in Chapter 7. A session with Shrinp is illustrated in Appendix B.

3) A comparison between several similar systems

Chapter 4 has a comparison in three dimensions regarding how some ILP systems cope with the generation of the base and recursive clauses as well as with the pruning of the hypothesis space.

4) The development of an algorithm to automatically generate negative examples

Negative examples are fundamental to the learning process in that they help discard wrong induced definitions as early as possible, thereby improving the accuracy of the solution and the efficiency of the whole process. Because of the assumption that only a few examples are available, the situation depicted in Section 1.3 is very likely to occur. To circumvent the problem, Shrinp has a special algorithm called GENEX that creates its own negative examples through the corruption of the given positive examples. Three operating modes with different degrees of reliability are provided. Although not as effective as user-given instances, the examples provided by the algorithm, introduced in Section 5.4, are enough for several important predicates to be learned alone in situations when no user-given negative examples are available—see Section 7.2.

5) The implementation of a method to eliminate incorrect base clauses

An incorrect base clause can be safely eliminated if it covers any negative example. If the base clause being tested is indeed the correct one, it should not cover any of the other candidates to be the target base clause. What this method does is to extend the user-given negative instances with potentially wrong base clauses, in hopes that in the end only the correct base clause will remain. The approach, first introduced in [Mofizur, Numao 95], is supported by a bias in the specification language that limits the learned definitions to have a unique base clause. See Section 5.4.3 for

constraints and details on the elimination of base clauses.

6) The development and implementation of CLAM

CLAM is another extension of Crustacean that learns purely recursive and left-recursive definitions. But, unlike Shrinp, it is not capable of predicate invention and, therefore, depends on background knowledge. The system is described in Section 5.1 and the result of experiments in Section 7.4.

7) The development of a method to relearn the base clause

When a purely recursive definition is extended to become a right- or a left-recursive definition, the input/output arguments of the base clause have to be relearned to reflect possible changes on the arguments of the head of the recursive clause. This is due to the addition of a new literal to the body of the recursive clause. Shrinp is one of the two inverse implication-based systems to check the correctness of the base clause after a definition is found (the other system is CLAM). If incorrect, the base clause is relearned—see Section 5.5.

8) The design of argument-binding chains

Argument-binding chains are like program templates that analyze the variables and arguments in an incomplete clause and choose some of them to compose the list of arguments of the newly invented predicates. The argument-binding chains applied by Shrinp account for several well-known left- and right-recursively defined relations. They are described in Sections 6.3 and 6.4.

9) A compilation of methods that learn from positive examples only

As explained earlier, the hunger for evidence presents a serious problem and a barrier to the usability of ILP systems. In real-life applications one cannot expect to have a large number of examples available. It is up to the learner to somehow cope with the incompleteness of the evidence or even create its own examples, particularly negative instances. Section 4.10 presents several attempts to learn from positive data only.

10) The development of clause filters

Before the negative examples are used to check the correctness of a potential definition, Shrinp

runs a series of syntactic filters in order to try to validate the definition as early as possible. The filters, described in Section 5.3.2, include tests for infinite recursion, repetition and the presence of useless arguments.

11) Extensions to the method

A final contribution of this work regards the suggestion of several directions for further work, as presented in Section 8.1.

1.6 Outline

This work is divided in nine chapters and two appendices, organized in the following way.

Chapter 2 has a somewhat detailed definition of various basic concepts regarding logic and ILP. Several important topics are defined, including clauses, recursive definitions, unification, substitution, predicate invention, necessary terms, generalization, lgg etc. Particularly, a formal definition of the ILP task can be found there.

The following chapter discusses the theoretical foundations that underpin this research. The resolution method is first defined, followed by a discussion of how to invert it. Next, inverse entailment is introduced. In the last sections, special attention is given to two systems that invert implication: LOPSTER and Crustacean, which form the basis for the present work.

Chapter 4 brings an overview of several learners that learn exclusively recursive definitions of a single predicate. The discussion is centered on three dimensions: the generation of the base clause, the induction of the recursive clause and the pruning of the hypothesis space. A special section is dedicated to learning in the absence of negative examples.

The systems Shrinp and CLAM are introduced in Chapter 5. The characteristics as well as details of Shrinp's implementation are presented. The automatic generation of negative examples is discussed in Section 5.4. Problems regarding the implementation and a complexity analysis are shown.

Due to its importance to this research, predicate invention is presented in a separate part. Chapter 6 discusses such issues as choosing an incomplete clause to be extended with a new

predicate, determining an argument structure for the call to the new predicate, finding examples and inducing a definition for the invented predicate. The idea of using argument-binding chains for selecting variables and creating new variables is described in that chapter.

In order to evaluate the implementation of Shrinp, a number of empirical experiments were run. The results are described in Chapter 7, which also shows experiments with the GENEX algorithm and the system CLAM.

Finally, Chapter 8 summarizes this research and introduces some directions for further work, suggesting possible solutions to the presented problems. The references can be found in Chapter 9.

Two appendices complete the present work. Appendix A contains the actual Prolog code that implements the least general generalization under implication of a list of terms. Appendix B shows the output of a session with Shrinp in which the user enters examples on the fly and the system induces the target relation by inventing a new predicate.

1.7 How to Read This Thesis

We suggest the chapters be read in the order in which they are presented. However, if the reader is already acquainted with the main concepts that involve ILP, Chapter 2 can be skipped. For those interested in the implementation only, we suggest a quick glance at Chapters 2 and 3 followed by a study of Chapters 5 and 6. Those who have only a general interest in the main topic of this work, predicate invention, can read Section 2.4 and proceed directly to Chapter 6. Chapter 3 should be read in all situations as it provides the basic issues that motivate the present research. The same is true for the conclusion in Chapter 8, which summarizes the whole research and explores some directions for future work.

chapter **two**

2. Basic Concepts

This chapter is devoted to logic and ILP preliminaries. This fairly lengthy chapter, which gives the definition and shows examples of the basic concepts discussed here, has been included in order to make this work as self-contained as possible. The discussion comprises five sections. We start by defining the pieces that form the language *L*, a subset of first-order logic, in which logic programs are written in this work. The other sections introduce such meaningful notions as syntactic entailment, unification and sub-unification, predicate invention, the basic representative set, generating terms, generalization, θ-subsumption and, most importantly, the ILP problem. The reader is in any event referred to such texts as [Lloyd 87], [Muggleton 90], [Chang, Lee 73], [Genesereth, Nilsson 87] and [Gallier 86] where more details can be found.

2.1 The Specification Language L

Alphabet. An alphabet Σ is a set that contains the symbols that can be used to form words and sentences in a language. The alphabet for *L* is composed of upper- and lower-case letters, numbers, punctuation symbols (*,*, *.*, *)*, *(*, *]* and *[*), connectives (¬, ∧, ∨, → and *:-*) and quantifiers (∀ and ∃).

Constants and Functions. Constant and function symbols are represented by lower-case words. T*rue*, *false* and the number *0* are also constants. A function is a mapping that relates a list of

constants, called the **arguments**, to a constant. The **arity** of a function is the number of arguments it requires. An n-ary function f is represented by $f(a_1,a_2,...,a_n)$ where $a_1,a_2,...,a_n$ are its arguments. Incidentally, constants can be viewed as 0-ary functions. While 0, a and b are examples of constants, $s(0)$ and $f(a,b)$ are examples of functions (a unary and a binary function, respectively). The function s, which is called the successor function, returns the next natural number to its argument. The numbers 1 and 2 are, thus, represented by $s(0)$ and $s(s(0))$. As a syntactic sugar, we will also be using s^n to represent the number n.

Variables. A variable is a word that starts with an upper-case letter. Examples of variables are: X, Y, Z, A, and $B2$. Once a variable becomes **instantiated** to a constant, it can be replaced by that value.

Terms. Terms are recursively defined as follows:

(i) a constant is a term.

(ii) a variable is a term.

(iii) if f is an n-ary function symbol and $t_1,t_2,...,t_n$ are terms, then $f(t_1,t_2,...,t_n)$ is a term.

(iv) nothing else is a term.

Terms are, therefore, sequences of constants, variables and functions. The latter can have as arguments other terms, in particular other functions with its arguments. So functions can be applied to other functions and so on.

NOTE

The language we are defining provides no representation for lists as customary in logic programming languages. A three-element list like $[a,b,c]$ is written in L as a dotted pair by use of the binary function *pair*. Hence, $[]$ (the empty list), $[a,b,c]$, and $[0,[s^1],a]$ are written as $[]$, $pair(a,pair(b,pair(c,[])))$, and $pair(0,pair(pair(s(0),[]),pair(a,[])))$, respectively. The use of a functional approach to represent lists makes it easy for a program to break such a structure into its smaller parts.

Predicates. A predicate symbol is a word that starts with a lower-case letter. A predicate is a mapping that maps a list of terms to one of *true* or *false*. Predicates also take a specified number of arguments, their **arity**. **Propositional logic**'s 0-ary predicates will not be addressed in this

work. We will be using the terms **relations** and predicates throughout the text as synonyms. Examples of predicate symbols are: *sum*, *reverse* and *nextTo*.

Atoms. If p is an n-ary predicate symbol and $t_1, t_2, \ldots, t_n$ are terms, then $p(t_1, t_2, \ldots, t_n)$ is called an atom.

Literals. If q is an atom, its **negation** is represented by $\sim q$. A literal is an atom (also called a positive literal) or the negation of an atom (also called a negative literal). If $vars(L)$ denotes the set of variables in a literal L, a **ground literal** is a literal L such that vars(L) = ∅. Two literals L_1 and L_2 are said to be **complementary** if $L_1 = \sim L_2$, i.e., if one is the negation of the other.

Formulas. We can recursively define a formula by the following:

(i) an atom is a formula.

(ii) if F and G are formulas, then so are $\sim F$, $F \vee G$, $F \wedge G$ and $F \rightarrow G$.

(iii) if F is a formula and x a variable, then $\forall x F$ and $\exists x F$ are also formulas.

(iv) nothing else is a formula.

Following the notation used in Prolog, we will sometimes use G :- F to denote $F \rightarrow G$.

Clause. A clause is a formula of the form:

$$Q_1 x_1\ Q_2 x_2, \ldots, Q_m x_m D$$

where D is a disjunction of literals, i.e., $L_1 \vee L_2 \vee \ldots \vee L_n$, $Q_1, Q_2, \ldots, Q_m$ are quantifiers and $x_1, x_2, \ldots, x_m$ are all the variables that appear in D. A **simplified clause** is a clause after the quantifiers are dropped.

Horn Clause. A Horn clause is a simplified clause with at most one positive atom of the form:

$$p \text{ :- } q_1, q_2, \ldots, q_n.$$

(or equivalently, the disjunction of literals $p \vee \sim q_1 \vee \sim q_2 \vee \ldots, \vee \sim q_n$). We usually give names to the parts of a Horn clause: p is the **head** while $q_1, q_2, \ldots, q_n$ form the **body** of the clause. The head p is the only positive literal in a Horn clause. When the body is empty, the symbol ":-" may be

dropped. Examples follow:

:- male(X), female(X).
:- p(a,b).
sum([], 0).
length([A|B],s(C)) :- length(B,C).
permut([A|B],C) :- permut(B,D),
select(A,D,C).

From now on we shall use the term "clause" to actually refer to Horn clauses. Whenever convenient, we shall regard a set of literals $\{L_1, L_2, ..., L_m\}$ as synonym with a clause. For instance, the clause p :- $q_1, q_2, ..., q_n$ can be expressed as $\{p, \sim q_1, \sim q_2, ..., \sim q_n\}$.

Ground Clause. A ground clause is a clause in which all literals are ground.

Unit Clause. A unit clause is a clause with one single literal.

Empty Clause. An empty clause, denoted by $\Box$, is a clause that has no literal.

Logic Program. A logic program, or program for short, is a finite set of clauses.

Definition. The set of clauses in a program P that have the same predicate q in the head is called a definition of q.

Recursive Clause. If a predicate p appears both in the head and the body of a clause, the clause is said to be recursive. Each occurrence of p in the body is known as a **recursive call** to p. **Linear recursive clauses** have only one recursive call, while **non-linear recursive clauses** have more than one call. A **binary recursive clause** is a non-linear recursive clause with two recursive calls. A **left-recursive clause** is a linear recursive clause in which the recursive call is the leftmost literal in the body. Likewise, a **right-recursive clause** is a linear recursive clause that has the recursive call as the rightmost literal in the body.

Base Clause. A base clause is a clause with an empty body that defines a base case for a recursive definition.

Recursive Definition. A definition with one base clause and one recursive clause is said to be a recursive definition. If it has more than one recursive clause, it is called a **disjunctive recursive definition**.

Purely Recursive Definition. A purely recursive definition (PRD) is a recursive definition with a two-literal recursive clause. Example:

```
length([],0).
length([A|B],s(C)) :- length(B,C).
```

Left-recursive Definition. A left-recursive definition (LRD) is a recursive definition with a left-recursive clause. Example:

```
permut([],[]).
permut([A|B],C) :- permut(B,D),
                   select(A,D,C).
```

Right-recursive Definition. A right-recursive definition (RRD) is a recursive definition with a right-recursive clause. Example:

```
doubles([],[]).
doubles([A|B], [C|D]) :- twice(A,C),
                         doubles(B,D).
```

Modes. The modes of a definition D are a list of the form $[m_1,m_2,...,m_n]$ where n is the arity of the predicate p in the head of D and each mode m_j, unlike Prolog, is labeled as either i for input or o for output. A mode m_j refers to the corresponding j^{th} argument of p. An input mode for an argument means that that argument is to be provided as input to the execution of p. An output mode means that the values returned by p are stored in those arguments. For instance, the predicate *length* has *[i,o]* as modes because it returns in the second argument the length of the list given in the first argument. In this work, the modes of a relation are restricted to be unique.

Input and Output Variables. If p :- $q_1, q_2, ..., q_n$ is a clause, then the **input variables** of the literal q_i are the variables in q_i which also appear in the clause p :- $q_1, q_2, ..., q_{i-1}$ except for those in the output arguments of the head literal p. All other variables in q_i are called **output variables**.

2.2 Deduction Theorem, Syntactic Entailment and Unification

Interpretation. Given a non-empty set D (called a **domain**), let $D_n = \{(X_1, X_2, ..., X_n) \mid X_i \in D, 1 \le i \le n\}$. The interpretation of a formula F is the assignment of a constant in D, a mapping of D_n into an element in D, and a mapping of D_n into $\{true, false\}$, respectively, to each constant, n-ary

function and *n*-ary predicate symbol in F.

Evaluation. The evaluation of a formula F over a domain D under an interpretation I is an assignment of *true* or *false* to F according to the following rules:

(i) if F is an atom, use the mapping that corresponds to the predicate symbol in F as defined in I. The terms in F are calculated by using the mapping for the functions in F;

(ii) if F is one of $\sim G$, $G \vee H$, $G \wedge H$ or $G \rightarrow H$, then use Table 2.1;

(iii) if F is of the form $\forall xG$, it is evaluated to *true* if the truth value of G is *true* for all d $\in$ D; otherwise, F is evaluated to *false*;

(iii) if F is of the form $\exists xG$, it is evaluated to *true* if there exists a d $\in$ D such that the truth value of G is *true*; otherwise, F is evaluated to *false*.

Table 2.1. The truth value of several formulas.

G	H	$\sim G$	$G \vee H$	$G \wedge H$	$G \rightarrow H$
true	*true*	*false*	*true*	*true*	*true*
true	*false*	*true*	*false*	*true*	*false*
false	*true*	*false*	*false*	*true*	*true*
false	*false*	*true*	*false*	*false*	*true*

Models. A formula F is said to be **consistent** or **satisfiable** if and only if there is an interpretation I such that F is *true* under I. If F is *true* under all interpretations, we say that F is a **valid** formula (also called a **tautology**). If there is no interpretation I for which F is *true*, F is said to be **inconsistent** or **unsatisfiable**. If an interpretation I satisfies F, we say that I is a model of F. Note that the empty clause □ is always *false* as it has no literal that can be satisfied by an interpretation.

Logical Consequence. A formula F is a logical consequence (or a **semantic entailment**) of a set of formulas S = $\{G_1, G_2, ..., G_n\}$, written as S $\models$ F, if and only if for every interpretation I, if $G_1 \wedge G_2 \wedge ... \wedge G_n$ is *true* in I, then F is also *true* in I. That is, if I is a model for S, I is a model for F. We also write $G_1, G_2, ..., G_n \models$ F to denote that F logically follows from $G_1 \wedge G_2 \wedge ... \wedge G_n$.

Theorem 2.1 (Deduction Theorem). Given a set S of formulas G_1, G_2, ..., G_n and a formula F, we have that F is a logical consequence of S if and only if $G_1 \wedge G_2 \wedge ... \wedge G_n \rightarrow F$ is valid.

Proof ($\rightarrow$) If F is a logical consequence of S, then for a given interpretation I of S, F is *true* under I. It follows from Table 2.1 that $G_1 \wedge G_2 \wedge ... \wedge G_n \rightarrow F$ is *true* under I. Now, if S is *false* in I, by Table 2.1 $G_1 \wedge G_2 \wedge ... \wedge G_n \rightarrow F$, too, is *true* in I. That is, $G_1 \wedge G_2 \wedge ... \wedge G_n \rightarrow F$ is *true* under any interpretation.

($\leftarrow$) Suppose $G_1 \wedge G_2 \wedge ... \wedge G_n \rightarrow F$ is valid, so it must be *true* under an interpretation I. Hence, if $G_1 \wedge G_2 \wedge ... \wedge G_n$ is *true* in I, F must also be *true* in I. It follows that S $\models$ F. The theorem is demonstrated.

Theorem 2.2 (Inconsistency Theorem). Given a set S of formulas G_1, G_2, ..., G_n and a formula F, F is a logical consequence of S if and only if the formula $G_1 \wedge G_2 \wedge ... \wedge G_n \wedge \sim F$ is inconsistent. In other words, to know if a formula F follows from a set of formulas, it suffices to add the negation of F to the set and check the inconsistency of the new set.

Proof. The negation of $G_1 \wedge G_2 \wedge ... \wedge G_n \rightarrow F$ is logically equivalent to $G_1 \wedge G_2 \wedge ... \wedge G_n \wedge \sim F$. By the Deduction Theorem, we have that S $\models$ F if and only if $G_1 \wedge G_2 \wedge ... \wedge G_n \rightarrow F$ is *true* under all interpretations, i.e., if and only if the negation of $G_1 \wedge G_2 \wedge ... \wedge G_n \rightarrow F$ is *false* under all interpretations.

Logical Equivalence. Two formulas C and D are said to be logically equivalent, denoted by C $\equiv$ D, if we have that C $\rightarrow$ D and D $\rightarrow$ C, i.e., if the truth values are the same under every interpretation.

Inference Rules. A relation R is called a rule of inference if there is a unique positive integer j such that for every set S of j formulas and each formula F, we can check whether the given j formulas are in the relation R with F. If so, F is called a **direct consequence** of S by virtue of R. For instance, the formula $q(a)$ is a direct consequence of the set $\{p(a), p(a) \rightarrow q(a)\}$ due to an inference rule called *modus ponens* which relates A and A $\rightarrow$ B to B for any formulas A and B.

Syntactic Entailment. A set S of formulas syntactically entails a formula F if and only if there is a sequence G_1, G_2, ..., G_n of formulas such that G_n = F and for each $1 \leq i \leq n$, either $G_i \in$ S or G_i is a

direct consequence by some inference rule of the preceding formulas in the sequence. Such a sequence is called a **proof** of F from S. When this is the case, we write S $\vdash$ F.

Substitution. A substitution is a finite set of the form $\{X_1/t_1, X_2/t_2, ..., X_n/t_n\}$ where $X_1, X_2, ..., X_n$ are variables and $t_1, t_2, ..., t_n$ are terms, $t_i \neq X_i$, $1 \leq i \leq n$. Moreover, no two elements in the set have the same variable before the slash symbol (i.e., the variables are distinct). Greek letters are used to name substitutions. The result of applying a substitution $\theta = \{X_1/t_1, X_2/t_2, ..., X_n/t_n\}$ to a literal L, denoted by Lθ, is a literal L' obtained by simultaneously replacing each occurrence of X_i in L by the term t_i, $1 \leq i \leq n$. To illustrate, let $\theta = \{X/a, Y/f(b), Z/X\}$ and L = $p(X,g(Y,c,X),Z)$. Then L' = Lθ = $p(a,g(f(b),c,a),X)$.

Inverse Substitutions. The process of replacing terms by variables in a literal L such that, for any substitution θ, we have that L$\theta\theta^{-1}$ = L is called inverse substitution. For instance, if L = $p(X,f(X))$ and $\theta = \{X/a\}$, then $\theta^{-1} = \{a/X\}$. An inverse substitution is actually a rewriting of a literal because it depends on the decision of which terms and subterms are to be replaced by the same or different variables. As an illustration, consider M' = Mθ = $q(0,s(0),0)$ for an unknown literal M and a substitution θ. There are several choices for θ^{-1}. For instance, for as simple an inverse substitution θ^{-1} as $\{0/X\}$, M can be $q(X,s(X),X)$, $q(X,s(0),0)$ or $q(0,s(X),X)$, etc. That is, a combinatorial explosion occurs. A language bias, as defined later, can help prevent or minimize the problem. [Muggleton, Buntine 88] discusses inverse substitution in more detail.

Unification. A substitution θ is called a **unifier** for a set $\{E_1, E_2, ..., E_n\}$ of expressions if and only if $E_1\theta = E_2\theta = ... = E_n\theta$. A set is said to be **unifiable** if there is a unifier for it. Example: the set S = $\{q([a],0,A), q(X,L,L)\}$ is unifiable since the substitution $\sigma = \{X/[a], L/0, A/0\}$ is a unifier for S. But $\{r(0,s(0)), r(s(X),Y)\}$ is not unifiable because the first arguments of r cannot be unified.

Most General Unifiers. Following [Robinson 65], a substitution σ is called the most general unifier of a set S = $\{E_1, E_2, ..., E_n\}$ of expressions if and only if for all unifiers θ of S there exists a substitution δ such that $(E_i\sigma)\delta = E_j\theta$ for all $1 \leq i,j \leq n$.

Coverage. A set of clauses S is said to (extensionally) cover an example e with respect to a set of clauses K whenever it is the case that S, K $\vdash e$.

2.3 Inductive Logic Programming

The ILP problem. We start this section by defining the fundamental problem in ILP.

Given:

- E^+, a set of literals called **positive examples,**
- E^-, a (possibly empty) set of (possibly ground) literals called **negative examples** such that $E^+ \cap E^- = \varnothing$,
- L, a specification language also called the **hypothesis language** and
- K, a (possibly empty) consistent set of definitions of supporting predicates such that $K \not\vdash e^+$ and $K \not\vdash e^-$, for all $e^+ \in E^+$ and $e^- \in E^-$, known as the **background knowledge,**

Find:

- a logic program $P \in L$ that covers all examples in E^+ but none in E^-.

That is, $P,K \vdash e^+$ for all $e^+ \in E^+$ (P is complete) but for every e^- in E^-, $P,K \not\vdash e^-$ (P is consistent). Said differently, it should not be possible to find a proof of the positive examples directly from the background knowledge alone, but from a combination of this with the learned program. Likewise, it should not be possible to prove the negative examples either directly from the background knowledge or together with the learned program. As can be seen, inductive inference can be seen as a problem of inverting deduction.

A system that is capable of learning a logic program in such conditions is called a **learner**. In particular, the sets E^+ and E^- may be composed of examples of only one recursively defined target predicate, in which case the program P is to contain a recursive definition for that predicate and possibly the definition of other supporting predicates.

As seen hitherto, the choice of the language *L* defined in Section 2.1, a rich subset of first-order logic, to express the learned programs permits the use of variables, quantification and recursion. Yet *L* is restricted enough to still be computable. Learned definitions expressed as collections of Horn clauses are, therefore, executable logic programs, which explains why this

sub-field of machine learning is also known as **inductive logic programming** ([Cameron-Jones, Quinlan 93]).

The set K is said to be **extensionally defined** if all its clauses are ground literals. It can also be **intensionally defined** if it contains non-ground clauses. The language L will be used as the hypothesis language from now on.

Training and Testing Sets. The union of E^+ and E^- is known as the training set. A testing set is a list of positive and negative examples that is used to test the logic program induced by a learner.

Integrity Constraints. An integrity constraint is a non-Horn clause of the form:

$$p_1 \wedge p_2 \wedge \dots p_n \rightarrow q_1 \vee q_2 \vee \dots \vee q_m$$

where the *q*s are called the positive conditions and the *p*s, the negative conditions. They are generally used to test some conditions that a program must satisfy to be correct. For instance, the integrity constraint:

$$union(A,B,C) \wedge member(X,C) \rightarrow member(X,A) \vee member(X,B)$$

expresses the condition that the members of the union of two sets must come from one or both of the sets.

Closed-World Assumption. Under the closed-world assumption (CWA), an example that does not belong to the set of positive examples is taken as negative. This mechanism makes it possible to include only "positive knowledge" in the database and assume the falsity of negative examples by default (see [Lifschiftz 87] and [Reiter 78]).

Hypothesis Space. The set of all possible learnable programs P is called the hypothesis space. The ILP task can, therefore, be seen as a **search** on the hypothesis space expressed as a graph whose nodes are programs and whose edges are operations that modify one program to form another.

Types of Learning. A learning method can be **bottom-up** or **top-down** depending on whether the hypotheses are generalized from an initial specific hypothesis or specialized from an initial general hypothesis, respectively ([Shapiro 92]).

Coverage Test. Given a training set T, a coverage test is a check of whether S covers all positive and no negative examples of T. A positive coverage test checks only the coverage of the positive instances, whereas a negative coverage test checks the coverage of the negative instances.

Cross-validation. Given a set S of examples, cross-validation is a technique that estimates the accuracy of a learner on unseen cases by dividing S into *n* blocks. The learner is run *n* times, in each of which one block is omitted from the training data and the learned program is tested on the examples in the omitted block ([Quinlan 93]). This testing method assures that each example participates in both training and testing.

Basic Representative Set (BRS). Following [Ling 91a], the BRS of a logic program P is a set S of ground atoms obtained by taking, for each clause of P, all ground atoms in a *true* instantiation of P. For instance, a BRS for the logic program below:

```
sum([],0).
sum([A|B], C) :- sum(B,D),
                 plus(A,D,C).
```

may contain the following literals: $\{sum([],0),\ sum([s^1,s^2],s^3),\ sum([s^2],\ s^2),\ plus(s^1,s^2,s^3)\}$. BRSes can, therefore, be very small. A BRS provides a minimum amount of information about the program to be identified. But note that there may be more than one logic program whose model is the target model. For instance, there are several algorithms to perform a search on a list: sequential search, binary search etc. All logic programs that implement these algorithms have the same target model. However, depending on the BRS that is presented, only one program will be identified.

Bottom Set. Let S be a set of clauses and C a clause. If *g(L)* denotes the set of all ground literals of the language *L*, the bottom set of C under S is given by the expression:

$$\text{Bot(C,S)} = \{l \in g(L) \text{ such that } S,\sim C \models \sim l\}$$

As an example, consider the clause C1 and the set S1 below:

```
p(X) :- q(X), r(X).                        (C1)
{s(X) :- r(X),                             (S1)
 p(X) :- t(X), q(X), s(X)}
```

Then the bottom of C1 under S1 is:

$Bot(C1,S1) = \{t(a), \sim q(a), \sim r(a), \sim s(a)\}$

where a is a new constant symbol that represents all constants in L.

Language Bias. Any restriction that is imposed on the hypothesis language, such as the use of variable-free clauses (ground clauses), function-free clauses, unit clauses and so on, is regarded as a language bias. Indeed, the term *bias* refers to any basis for excluding hypotheses from the search space, other than strict completeness and consistency with the examples. Its main purpose, therefore, is to reduce the hypothesis space. A strong bias denotes a less expressive hypothesis language, i.e., a smaller hypothesis space, and as a consequence means a more efficient learning. The bias can also be expressed as a syntactic constraint on the kind of logic programs that can be learned by a particular system. The language bias defined for the system Shrinp is shown in Section 6.2.1. For further discussion on the need for a language bias please see [Mitchell 90].

2.4 Predicate Invention, Sub-Unification and Generating Terms

Predicate Invention. When the language L and the background knowledge K are not enough to provide a program P with the characteristics mentioned in the previous section, the language can be extended with newly created predicates. Predicate invention (PI) represents a shift in the language bias with the introduction of a predicate that does not occur anywhere in the examples nor in the background knowledge and whose definition must be provided by the system ([Stahl 95a], [Utgoff 86]), although, of course, the *name* of the predicate exists in the language, but *not* its definition.. It is also up to the system to find examples for the new predicate before finding its definition. See Chapter 6 for more on PI.

A question arises here. Are all invented predicates really necessary to the success of a learning task? Following the following definitions of useful and necessary predicates (taken from [Ling 91b]), they are not.

Useful Predicates. A new predicate is classified as useful if its invention and use do not affect the learnability of a program. As such, useful predicates can be eliminated, although their presence makes the program more compact. For instance, given the program below (we have

ignored the variables for simplicity):

p1 :- q, r, r.
p2 :- q, r, r, s.
p3 :- t, q, r, r, q, r, r.

a useful predicate *u* can be invented that rewrites the program in a more compact way, by extracting the common part of the clauses:

p1 :- u.
p2 :- u, s.
p3 :- t, u, u.
u :- q, r, r.

but the program is learnable even in the absence of that predicate.

Necessary predicates. A new predicates is necessary if, without it, the program is not learnable (according to some criterion of success). Necessary predicates are *essentially recursive*. If not, all their occurrences in the body can be replaced by its definition and still the program would be correct[2]. As an example, consider the program below where *newp* is a newly created predicate:

p(A,B) :- p(B), newp(A).
newp(a).
newp(b).
newp(c).
newp(d).

The program can be equivalently rewritten *without* that predicate, which proves that it is not necessary:

p(a,B) :- p(B).
p(b,B) :- p(B).
p(c,B) :- p(B).
p(d,B) :- p(B).

On the other hand, the definition of the relation *subset* below, which tests if the first argument is a subset of the second, is incomplete without the call to *newPred8* (the predicate

[2]This does not mean that *all* invented predicates are recursive, though, but only that for an invented predicate to be considered necessary for a learning session, it *must* be recursive.

member). Note that *newPred8* is recursively defined.

```
subset([],[A|B]).
subset([A|B],[C|D]) :- newPred8(A,[C|D]),
                       subset(B,[C|D]).
newPred8(A,[A|B]).
newPred8(A,[B,C|D]) :- newPred8(A,[C|D]).
```

The invention of necessary predicates overcomes the weakness of many concept learning systems in which all necessary background knowledge has to be supplied ([Ling 91b]).

Subterms and Embedding Terms. Given two terms t_1 and t_2, t_1 is said to be a subterm of t_2 if and only if there exists a third term t_e and a substitution θ such that a term identical to t_2 (up to a substitution to yield the variables in t_2) is obtained after θ is applied by substituting t_1 for a variable in t_e. The term t_e is called an embedding term of t_1 in t_2. For example, given $t_2 = f(a,g(b))$ and $t_e = f(a,V)$, a term $t_1 = g(b)$ is a subterm of t_2 if the substitution $\theta = \{V/g(b)\}$ is applied to t_e, which yields t_2 (see illustration in Fig 2.1). Table 2.2 presents the several subterms and embedding terms of *pair(s(0),pair(s(s(0)),[]))*, i.e. the list *[1,2]*. The last row has a trivial embedding term.

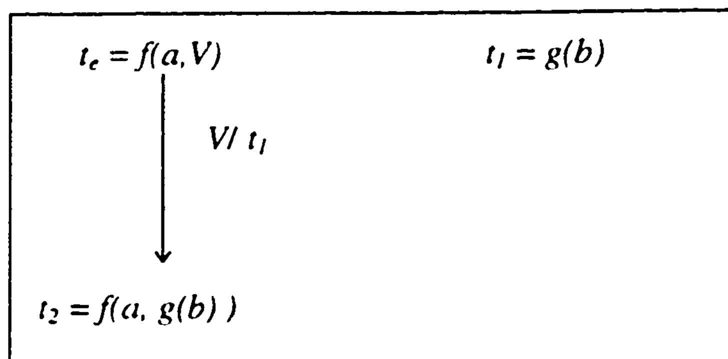


Fig 2.1. t_e is an embedding term of t_1 in t_2.

Sub-unification. Following the definition in [Lapointe, Matwin 92], if there is an embedding term for a term t_1 with respect to a term t_2, then t_1 is said to be **sub-unifiable** in t_2. For instance, $t_1 = s(0)$ is sub-unifiable in $f(s(s(0)),a)$ since substituting t_1 for X in $t_e = f(s(X),a)$ produces t_2. Unification is a special case of sub-unification with the embedding term being a variable. See also [Lapointe, Ling, Matwin 93] for details.

Generating Terms. Let t^V be a term where a variable V occurs exactly once. If an embedding

term t_c can be generated (up to a substitution) by *n-1* recursive substitutions of t^V for V in t^V, t^V is called a *generating term (GT) of* **depth** *n* (or a GT-n) of t_c. For example, $t^V = s(V)$ is a generating term of depth three of *s(s(s(X)))* as two substitutions of t^V for *V* are necessary to make it identical to *s(s(s(X)))* up to a substitution that replaces *V* by *X*. It is also a GT-1 of *s(X)* as no substitutions are necessary to make them identical (except for replacing *V* by *X*). In other words, a generating term describes how an embedding term can be recursively obtained from a term by repeating the same substitution.

A generating term is described in Shrinp by a sequence of operations G = $g_1, g_2, \ldots, g_n$. Each g_i is of the form $f_i(arg_i)\text{-}depth_i$, for $1 \le i \le n$, where f_i is a function, arg_i a number less than or equal to the arity of f_i, and $depth_i$ the number of times the arg_i-th argument of f_i is to be recursively obtained before g_{i+1} is executed. The operations of G are applied from left to right on a term t_1 in order to retrieve a particular subterm of t_1. For instance, the GT *f(2)-1* returns the subterm *V* in $t_1 = f(a, V)$ by accessing the second argument of the function *f* once.

Table 2.2. Subterms, embedding terms and generating terms of *pair(s(0), pair(s(s(0)),[ ]))*.

Subterm	Embedding term	Generating Term
0	*pair(s(X), pair(s(s(0)),[ ]))*	*pair(1)-1,s(1)-1*
0	*pair(s(0), pair(s(s(X)),[ ]))*	*pair(2)-1,pair(1)-1,s(1)-1, s(1)-1*
s(0)	*pair(X, pair(s(s(0)),[ ]))*	*pair(1)-1*
s(0)	*pair(s(0), pair(s(X),[ ]))*	*pair(2)-1,pair(1)-1,s(1)-1*
s(s(0))	*pair(s(0), pair(X,[ ]))*	*pair(2)-1,pair(1)-1*
[ ]	*pair(s(0), pair(s(s(0)),X))*	two GTs: *pair(2)-1,pair(2)-1* *pair(2)-2*
pair(s(s(0)),[ ])	*pair(s(0), X)*	*pair(2)-1*
pair(s(0), pair(s(s(0)),[ ]))	*X*	*none*

Table 2.2 shows the generating terms of all subterms of the term t_1 = *pair(s(0),pair(s(s(0)),[ ]))*. The subterm *s(0)*, for example, appears twice in t_1 and, thus, has two embedding terms. The GT of the second embedding term of *s(0)*, t_2 = *pair(2)-1, pair(1)-1, s(1)-1*, tells that to retrieve *X* in the embedding term it suffices to first get the second argument of *pair* once in t_2,

which returns *pair(s(X),[ ])*. Then, *pair(1)-1* tells to get the first argument of *pair*, which yields *s(X)*. Finally, *s(1)-1* accesses *X*. The subterm *0* also has two embedding terms that are accessed through different GTs. Note that *[]* has only one subterm, but nonetheless can be retrieved in two different ways: through a sequential or a recursive application of *pair(2)*. The first has a depth of 1 and the latter, a depth of 2. Trivial subterms like *pair(s(0),pair(s(s(0)),[]))* have no GTs as no decomposition is required to access the variable *X*.

Most Specific Generating Terms. The most specific generating term of a subterm, or **msgt** for short, is a generating term G for which there is a substitution θ such that for all other GTs H, it is true that $H = G\theta$.

2.5 θ-Subsumption, Generalization, Lgg and ij-Determinism

θ-Subsumption. A clause C θ-subsumes a clause D, denoted by $C \preceq D$, if and only if there is a substitution θ such that $C\theta \subseteq D$. In this case, we say that C is a **generalization under θ-subsumption** of D. A clause is said to be **reduced** when it is not θ-subsumed by any of its **sub-clauses**, that is, by any proper subset of itself.

The operator $\preceq$ defines a lattice on the family of clauses. The lattice implies the existence of a unique (up to a substitution) least upper bound and a unique greatest lower bound.

As an example of generalization under θ-subsumption, consider the following set of clauses. It is easy to see that, while A_3 is a generalization under θ-subsumption of A_2 ($A_3 \subseteq A_2$ with $\theta = \{\}$), A_4 generalizes not only A_2 but also A_1 and A_3 with substitutions *{X/regis}*, *{X/liane}* and *{X/regis}*, respectively.

mortal(liane) :- human(liane).	(A_1)
mortal(regis) :- human(regis).	(A_2)
mortal(regis).	(A_3)
mortal(X).	(A_4)
mortal(X) :- human(X).	(A_5)
mortal(father(Y)) :- human(Y).	(A_6)

Generalization Under Implication. By the Deduction Theorem, a clause C implies a clause D, denoted by $C \rightarrow D$, whenever $C \models D$. C is called a generalization of D under implication. Note

that implication is a weaker relation than θ-subsumption. That is, if a clause C θ-subsumes a clause D, we can conclude that C implies D, but the converse is not always true. For instance, A_4 above θ-subsumes A_2. Besides, it implies A_2 (see Table 2.1 above). On the other hand, A_5 implies A_6 but does not θ-subsume A_6.

The generalization under implication does not induce a lattice on the set of formulas but only a partial order.

Least General Generalization. Following the definition given by Gordon Plotkin at Edinburgh ([Plotkin 71]), the least general generalization (**lgg**) under θ-subsumption of a clause C is a clause D such that D ≤ C and D is less general than all other clauses E such that E ≤ C, that is, E ≤ D for all E. The lgg, therefore, somehow represents all generalizations and is the greatest lower bound in the lattice defined by the θ-subsumption operator. Below is the recursive algorithm to compute the lgg of two clauses C and D (adapted from [Lavrač, Džeroski 94]):

(a) if C and D are terms:

(i) lgg(C,D) = C if C = D;

(ii) lgg($f(t_1,t_2,\ldots,t_n)$, $g(s_1,s_2,\ldots,s_n)$) = f(lgg(t_1,s_1), lgg(t_2,s_2), ..., lgg(t_n,s_n)) if f and g are the same function;

(iii) otherwise, lgg(C,D) = X, a new variable that represents the lgg.

(b) if C and D are atoms:

(iv) lgg($p(t_1,t_2,\ldots,t_n)$, $q(s_1,s_2,\ldots,s_n)$) = p(lgg(t_1,s_1), lgg(t_2,s_2), ..., lgg(t_n,s_n)) if p and q are the same predicate;

(v) otherwise, lgg(C,D) = X.

(c) if C and D are literals:

(vi) lgg(C,D) is given by (iv) and (v) if C and D are positive literals;

(vii) lgg(C,D) = ~lgg(~C,~D) if C and D are negative literals;

(viii) lgg(C,D) is undefined if C and D have different polarity.

Now we can define the lgg of a set of clauses. The lgg of a set $\{C_i : 1 \leq i \leq n\}$ of clauses is a clause D such that D = lgg(C_i) for all $1 \leq i \leq n$. For instance, consider again the clauses A_1 through A_6 shown before. A_4 is a generalization of both A_1 and A_2 since $A_4\theta_1 \subseteq A_1$ and $A_4\theta_2 \subseteq A_2$ for $\theta_1 = \{X/liane\}$ and $\theta_2 = \{X/regis\}$, but is not their lgg as A_4 is more general than the correct lgg, A_5, since $A_4\theta_3 \subseteq A_5$ for $\theta_3 = \{\}$. [Muggleton, Feng 90] notes that the lgg of a set of clauses may grow exponentially in the number of clauses in the set.

The lgg of several terms is shown in Table 2.3.

Table 2.3. The lgg of several terms under θ-subsumption.

term A	**term B**	**lgg (A, B)**
a	*a*	*a*
pair(a,[])	*[]*	*X*
pair(a,[])	*pair(a,[])*	*pair(a,[])*
pair(a,[])	*pair(b,[])*	*pair(X,[])*
pair(a,[])	*pair(b,pair(a,[]))*	*pair(X,Y)*
pair(a,pair(b,pair(c,[])))	*pair(a,pair(b,[]))*	*pair(a,pair(b,X))*
s(0)	*0*	*X*
s(0)	*s(s(0))*	*s(X)*

Lgg Under Implication. In a similar way, one can define lgg under implication: a least general generalization under implication of a clause C is a clause D such that D → C and D is less general than all other clauses E such that E → C, that is, E → D for all E. Due to the simplicity and decidability of the algorithm to compute generalizations under θ-subsumption, this relation has been given more attention than its counterpart. The computation of an lgg under implication is more complicated as one has to take into account the bindings that have being already made to pairs of terms that appeared before. For instance, the lgg under implication of the terms *f(a,b,a)* and *f(d,b,d)* is *f(X,b,X)*. The algorithm is supposed to "remember" that *lgg(a,d) = X* when it sees the pair *a* and *d* for the second time in the third argument. Moreover, the implication is undecidable between Horn clauses ([Marcinkowski, Pacholski 92]).

On the other hand, the computation of the lgg of *f(a,b,a)* and *f(d,b,d)* under Plotkin's technique is relatively easier. It suffices to choose a new variable whenever two terms disagree, which yields *f(X,b,Y)*. This simplicity, nonetheless, sometimes results in an over-generalization. Consider, for instance, the clauses B1 and B2 next (borrowed with modifications from [Idestam-Almquist 95]):

$is_natural(s^1)$:- $is_natural(0)$. (B1)
$is_natural(s^3)$:- $is_natural(s^2)$. (B2)

While their lgg under θ-subsumption yields the clause B3 below, the same operation under implication returns a clause, B4, that is clearly less general than B3. As a matter of fact, B4 happens to be more appropriate a definition of a predicate to recognize a natural number than B3. To learn recursive clauses, generalization under θ-subsumption is not very adequate after all, as this example shows.

is_natural(s(X)) :- is_natural(Y). (B3)
is_natural(s(X)) :- is_natural(X). (B4)

Unless otherwise stated, the use of lgg hereafter will always refer to the lgg under θ-subsumption. The Prolog code to compute an lgg under implication of a list of terms is given in Appendix A.

Relative Least General Generalization. Given a set of background clauses K and a clause C such that K $\not\vdash$ C, we say that D is the lgg under θ-subsumption of C relative to K, often called the **rlgg** under θ-subsumption, if D is the lgg of C within the θ-subsumption lattice for which K,D $\vdash$ C. A clause E is the rlgg of a set of clauses S with respect to K if K,E $\vdash$ S. As first noticed by Plotkin ([Plotkin 71]), for some S and K, the rlgg may be undefined or produce an infinite clause. However, for ground clauses the rlgg is defined. The use of rlgg henceforth refers to the rlgg under θ-subsumption, unless otherwise stated.

ij-Determinism. Very often, the clauses produced by the rlgg operation are extremely long. They may also contain nonproductive variables in the sense that they do not depend on any of the variables that appear before their first occurrence in a clause. In order to restrict the size of the rlggs, the idea of ij-determinism was introduced in the system Golem ([Muggleton, Feng 90]). A

literal L in a clause C is said to be **determinate** if its output variables have at most one possible ground substitution, given that the input variables of L have already been instantiated when L is evaluated. A clause is said to be determinate[3] if all its literals are determinate. The **degree** of a variable V in L is the number of variables in L on which V depends, that is, that are bound when L is evaluated. The **depth** of a variable V is recursively defined as zero, if V belongs to the head of the clause; or one plus the depth of the input variables of the literal where V first appears in a clause as an output variable, otherwise. A clause C is said to be ij-determinate if it is determinate and the literals in its body have a depth of at most *i* and a degree of at most *j*.

Hence, ij-determinism is a language bias that imposes the limits *i* and *j* to the body of a clause. For small values of *i* and *j*, two for instance, many classic predicates are ij-determinate (such as *quicksort, reverse, multiplication, partition* etc.). Moreover, note that it is possible to redefine non ij-determinate clauses in an ij-determinate form ([Muggleton, Feng 90]).

[3]Determinate clauses can be evaluated by a Prolog interpreter without backtracking.

chapter **three**

3. Theoretical Background

The *resolution method* ([Robinson 65]) is a very well-known, complete inference system that forms the basis of Prolog[4]. The completeness means that whenever we have C $\models$ D for two clauses C and D, it is also true that C $\vdash$ D through the use of the method (and vice-versa). A strong theoretical basis for the construction of learning systems is possible if the resolution rule is inverted. Unfortunately, though, inverting resolution is not a straightforward task. It involves the intricacies of generalization.

This chapter provides the theoretical background for this work. It starts by explaining why implication is more appropriate to generalization than other operations. The chapter also presents attempts to invert resolution that make use of θ-subsumption and implication. The pros and cons of each method are shown as well. Section 3.4.2.1 contains an example of Crustacean's execution when trying to learn a purely recursive definition. The main concepts dealt with here have been defined in the previous chapter.

3.1 ILP as a Generalization Problem

The main operations in ILP are *generalization* and *specialization*. Recall from the previous

[4]Actually, Prolog relies on a special variant of resolution called *SLD-resolution* ([Siekman, Wrightson 83]), which enjoys such nice properties as completeness, correctness and, especially, efficiency.

chapter that the ILP problem consists in finding a program P that is more general than a number of given examples E^+. In a logical framework, this fact can be written as $P \models E^+$. Additionally, a set E^- of negative instances can also be given that prevents the program from being over-generalized. For P to be correct, we must have that $P \not\models E^-$. Some learning systems accept a set K of background information that helps defining P. The presence of background knowledge changes the goal to be $K,P \models E^+$ and $K,P \not\models E^-$. By the Deduction Theorem (Section 2.2), these expressions can be rewritten as $K \wedge P \rightarrow E^+$ but not $K \wedge P \rightarrow E^-$. There is, therefore, a close relationship between generalization and *implication*: generalization can be seen as a problem of inverting implication.

Generalization can also be defined in terms of θ-subsumption, as discussed in Section 2.5, which was shown to be stronger a relation than implication. So, it is possible to have $E \models F$ for two clauses E and F but not $E \leq F$. The generalization under θ-subsumption is, thus, *incomplete* and may not be sufficient for learning recursive definitions (see explanation in that section). Additionally, the *lgg* under θ-subsumption of a set of clauses may grow exponentially in the number of clauses ([Muggleton, Feng 90]) and can result in an over-generalization.

Considering all these facts, one may believe that implication is the basis for generalization in ILP. Nonetheless, θ-subsumption has been given far more attention than its counterpart for these main reasons:

1. θ-subsumption is more tractable than implication;
2. It allows us to demonstrate some nice properties with respect to the lattice that it defines. Such a structure is not conveyed by implication;
3. Generalization under θ-subsumption is *decidable* ([Robinson 65] and [Laag, Nienhuys-Cheng 93]) and can be implemented, in practice, reasonably efficiently in spite of being NP-complete;
4. Implication is *undecidable* not only between clauses ([Schmidt-Schauβ 88]) but also between Horn clauses ([Marcinkowski, Pacholski 92] in spite of a false claim in [Niblett 88] to the

contrary), as a natural consequence of the undecidability of first-order logic[5].

Because of the deficiencies of θ-subsumption, it is desirable to make the step from θ-subsumption to the more powerful implication, possibly trying to circumvent the undecidability problem. In this respect, a different kind of implication, called *T-implication*, was introduced in [Idestam-Almquist 93a and 95].

T-implication offers a "good approximation" to implication by being a stronger relation. At the same time, it is a weaker relation than θ-subsumption. The great merit of T-implication is the existence of an algorithm to decide whether a clause T-implies another. As a consequence, the *lgg* under T-implication of a finite set of clauses is defined. Unfortunately, though, from a theoretical point of view T-implication cannot be blindly applied as the basis for generalization (see Fig. 3.1, which summarizes the discussion here). The *lgg* of Horn clauses under T-implication is not guaranteed to be a Horn clause, in which case the result of a generalization might not be expressible in Prolog.

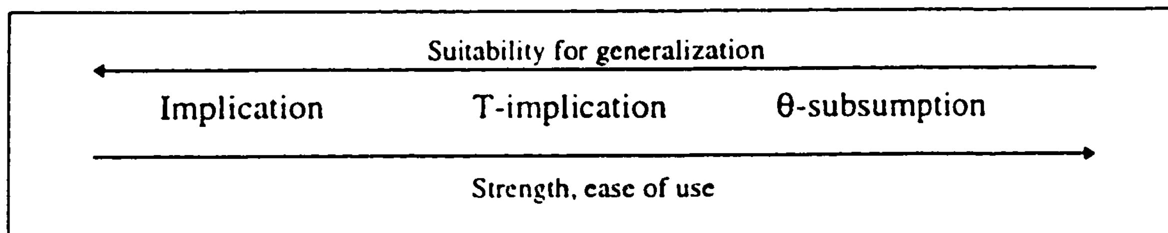


Fig 3.1. Strength is inversely proportional to suitability for generalization.

The existence of an algorithm to compute the *lgg* under these operations is summarized in Table 3.1 (adapted from [Nienhuys-Cheng, DeWolf 96b]) where "✓" means a positive answer and "✗" a negative answer.

[5]Although [Neinhuys-Cheng, DeWolf 96a] gives a proof that an lgg under implication is computable for any finite set of clauses that contains at least one non-tautologous function-free clause (among other, not necessarily function-free clauses).

Table 3.1. Existence of the *lgg* under three operations.

Operation	Horn Clauses	General Clauses
Implication	✗	✓ for function-free clauses[6]
T-implication	✓ (but the result may not be a Horn clause)	✓
θ-subsumption	✓	✓

A fast algorithm to test the implication between two clauses, which in most cases reduces the test to a θ-subsumption test although still running the risk of never stopping, is shown in [Gottlob 87]. [Niblett 88] investigates in more details the decidability of generalization under implication and θ-subsumption in the presence/absence of background knowledge.

3.2 The Resolution Principle

The construction of a machine that is able to reason (infer) by itself is a Western culture's ancient dream ([Rios 89)]. It was not until 1965, when Robinson published his Resolution Inference System ([Robinson 65]), that this dream has been effectively realized. He developed a unique inference rule that did not require any extra set of formulas other than the one that comprises the problem. The path to *automatic theorem proving*, and as a consequence to Prolog, was made easier.

Robinson noted that it suffices to have two clauses C1 and C2 with complementary literals L1 ∈ C1 and L2 ∈ C2 that can be made equivalent via *unification*. The *resolution rule* involves the production of a new clause R composed of the literals in both C1 and C2 except for the complementary literals, which are discarded. Without loss of generality, if we assume that no two clauses share the same variables, then C1 and C2 have no common variables. Then, we have that $L1\theta_1 = \sim L2\theta_2$, with $\theta = \theta_1\theta_2$ the most general unifier of L1 and L2. The same substitutions θ_1 and θ_2 are applied to the remaining literals in both C1 and C2, respectively, resulting in the clause below.

[6]The existence of an lgg under implication of a finite set of clauses is still an open problem. Because of the undecidability of the implication between clauses, it is clear that the lgg in these circumstances is in general not computable.

$(C1 - \{L1\})\theta_1 \cup (C2 - \{L2\})\theta_2$ (R)

R is called the *resolvent* of C1 and C2 with C1, C2 $\models$ R. The derivation of R is depicted in Fig. 3.2.

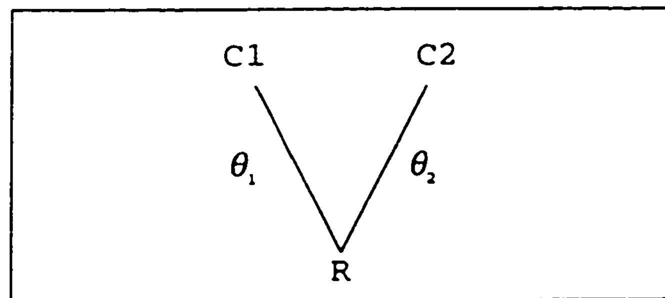


Fig 3.2. R is the resolvent of C1 and C2.

To test if a formula F logically follows from a set S of formulas, the resolution principle is employed in the following manner. By the Inconsistency Theorem (Theorem 2.2), F logically follows from S if and only if the new set S' formed by adding the negation of F to S is unsatisfiable. The resolution principle first checks whether S' has the empty clause □. If it does, S' is unsatisfiable. If not, the next thing to check is whether □ can be derived from S' by applying the resolution rule a finite number of times on members of S'. After two clauses in S' are resolved together, the resolvent R is added to S' and a new resolvent is generated, possibly using R[7]. The process continues until two unit clauses are resolved that yield the empty clause, in which case S' is unsatisfiable and, as a consequence, S $\models$ F.

As an example, consider the set S = {*(p(X,Y) :- q(X),r(Y))*, *q(a)*, *r(b)*} and the formula F = *p(a,b)*. The set S' is composed of the clauses:

1 *p(X,Y) :- q(X),r(Y)*

2 *q(a)*

3 *r(b)*

4 *:- p(a,b)* This is the negation of F

By resolving the clauses indicated at the right, the following sequence of resolvents is produced.

[7]A resolution method that always resolves with the most recently produced resolvent is called a *Linear Resolution* ([Loveland 70], [Luckham 70]).

The last resolvent being the null clause implies that S $\models$ F.

5 *:- q(a), r(b)*	Resolving 4 against 1
6 *:- r(b)*	Resolving 5 against 2
7 □	Resolving 6 against 3

The choice of the pair of clauses to resolve grows exponentially with the number of resolvents and the size of S. Various strategies, like linear resolution, are analyzed in the literature ([Loveland 78], [Casanova 87], [Chang, Lee 73]). With the development of Prolog ([Kowalski 72, 74, 79], [Colmerauer, Roussel 93]), a programming language based on resolution, the possibility of using this method in an efficient manner was shown to be feasible.

When one of the clauses C1 and C2 is recursive, it can be reused several times in a same derivation, i.e., in the same *resolution chain*. In Fig. 3.3, C is first resolved with A_1 and then continually with the resolvents A_2, A_3 etc. until the final clause A_n is derived.

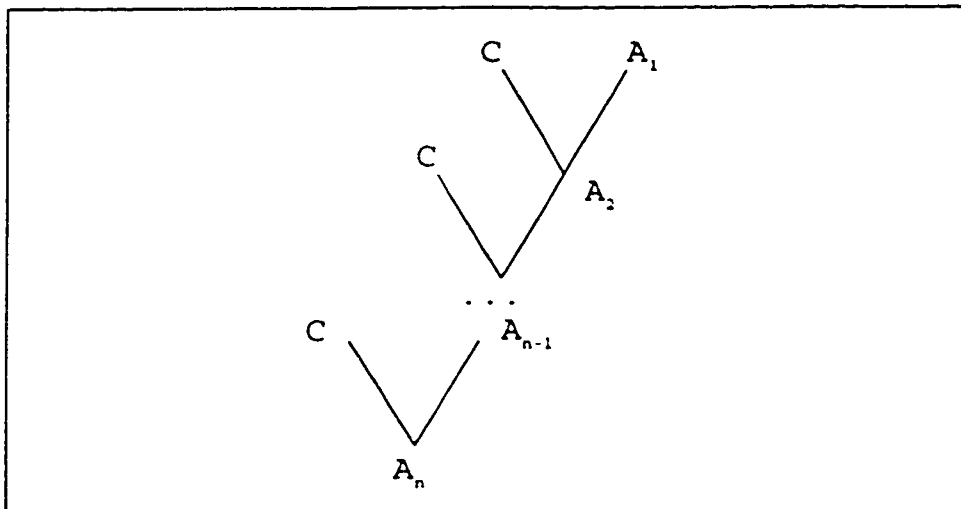


Fig 3.3. A *linear derivation* in which a clause C is recursively used several times.

For instance, the purely recursive clause *last_of(A,[C|D]) :- last_of(A,D)* can be resolved several times with F = *last_of(e,[r,o,s,e])* before the clause *last_of(e,[e])* is reached. This predicate, with modes *[i,i]* (that is, *last_of* has two input arguments), checks if its first argument is the last element of the list in the second argument.

1 *last_of(A,[C|D]) :- last_of(A,D)*

2 *:- last_of(e,[r,o,s,e])* Negation of F

3 *:- last_of(e,[o,s,e])* Resolving 1 against 2

4 *:- last_of(e,[s,e])* Resolving 1 against 3

5 *:- last_of(e,[e])* Resolving 1 against 4

This last resolvent is an instance of the base clause of this relation, *last_of(A,[A])*. When the base clause is resolved against that resolvent, the null clause is found:

6 □ Resolving 5 against the base clause

3.3 Inverting Resolution

The first system to try to invert resolution was Duce ([Muggleton 87]) which was restricted to 0-arity predicates (i.e., propositional logic)[8]. It was only in 1988 in a seminal work, which presented the system CIGOL ([Muggleton, Buntine 88]), that the inverse resolution problem was essentially introduced. Since then, research on inverse resolution abounds: [Wirth 89], [Rouveirol, Puget 90], [Muggleton 91], [Rouveirol 92], [Banerji 92] and [Flach 93] to name but a few. Various methods are discussed in [Ling, Narayan 91].

Given the V-like resolution rule shown in Fig. 3.2, the task of inverse resolution can be stated as: "how to get to one of C1 or C2 given R and the clause on the other arm of the V?" Algebraically speaking, the problem is expressed as an attempt to invert the equations below:

$$R = (C1 - \{L1\})\theta_1 \cup (C2 - \{L2\})\theta_2 \qquad \text{(E1)}$$

$$L1\,\theta_1 = \sim L2\ \theta_2 \qquad \text{(E2)}$$

Two problems arise concerning the *uniqueness* of the solution. Consider, for instance, the clauses below where the literal *s* appears as negative in C3 and positive in C4:

p :- q, r, s. (C3)
s :- b, c. (C4)

[8]An analysis of the system can also be found in [Muggleton 91].

which resolve, after eliminating *s*, as:

p :- *q*, *r*, *b*, *c*. (R2)

Now, suppose we are given only R2 and C4 and are asked to induce C3. Then there will not be a unique solution to this problem. All clauses below are correct solutions:

p :- *q*, *r*, *s*, *b*.
p :- *q*, *r*, *s*, *c*.
p :- *q*, *r*, *s*, *b*, *c*.

The uniqueness problem occurs primarily when we take into account *n*-ary predicates with $n > 0$ (i.e. first-order relations). The next example is borrowed from [Muggleton, Buntine 88]. Given the two clauses below:

:- *heavier(hammer, feather).* (C5)
:- *denser(hammer, feather), larger(hammer, feather).* (R3)

Several possibilities exist for a clause that resolves with C5 to produce R3. They go from very specific, like C6 below, to very general, like C8, depending on which terms are turned into variables. The clause C7 is a somewhat intermediate solution.

heavier(hammer, feather) :- denser(hammer, feather), larger(hammer, feather). (C6)
heavier(hammer, Y) :- denser(hammer, Y), larger(hammer, Y). (C7)
heavier(X, Y) :- denser(X, Y), larger(X, Y). (C8)

To cope with the problem, CIGOL and other systems use two kinds of operators: the *V* and the *W* operators, explained in the next section.

3.3.1 The V operators

Depending on whether the clause with the positive complementary literal is known or not, two operators are defined: *absorption* and *identification*.

3.3.1.1 The absorption operator

Due to [Sammut, Banerji 86], this operator builds the clause with the negated literal (clause C2) from C1 and R in Fig. 3.2. Here, the body of C1 is completely contained within the body of R.

Hence, the body of C1 is "absorbed" by a part of R. The V-shaped representation is shown in Fig. 3.4.

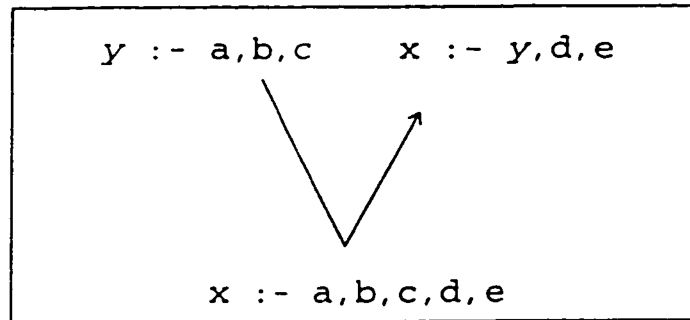


Fig 3.4. The absorption operator.

To illustrate its use, consider the clauses C9 and R4 below:

alive(X) :- breathes(X), haspulse(X). (C9)
happy(liane) :- breathes(liane), haspulse(liane), smiles(liane). (R4)

the absorption operator produces a correct solution by using the substitution $\theta = \{X/liane\}$:

happy(liane) :- alive(liane), smiles(liane) (C10)

It is common in the systems that implement absorption to build the generalization C2 from the positive example R and background clause C1. From the pair of equations E1 and E2 shown earlier, the clause C2 can be found by computing:

$$C2 = (R - (C1 - \{L1\})\theta_1\theta_2^{-1} \cup \{\sim L1\}\theta_1\theta_2^{-1} \quad \text{(E3)}$$

given that $L2 = \sim L1\theta_1\theta_2^{-1}$. Different solutions depend on the choice of θ_1, θ_2^{-1} and the literals from both C1 and C2 other than L1 and L2. But as discussed in Section 2.2, a combinatorial explosion occurs when searching for an inverse substitution like θ_2^{-1}. What CIGOL and other systems like IRES ([Rouveirol, Puget 89]), ITOU ([Rouveirol 91] and LFP2 ([Wirth 89]) do is to impose a syntactic bias on the language, such as the use of Horn clauses, unit clauses etc., which limits the number of literals in a clause. The more restrictive the bias, the more simplified equation E3 becomes.

3.3.1.2 The identification operator

The identification operator is applied when the clause with the positive complementary literal

(C1 in Fig. 3.2) is not given. Clearly, both C2 and R have to have the same head predicate. To find C1, it suffices to search for a literal in the body of C2 that does not occur in R. This operator is stronger than the previous one in that the head of the generated clause is not the same as the head of the other two clauses (see illustration in Fig. 3.5).

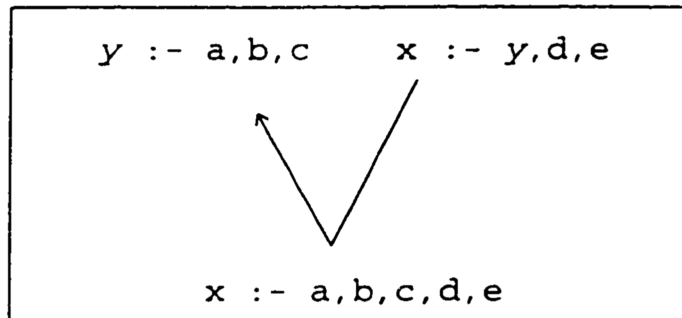


Fig 3.5. The identification operator.

For instance, given C10 and R4 as before, the identification operator produces a correct solution by using the substitution $\theta = \{X/liane\}$ and *alive* as the head literal. Observe that none of the previous clauses C10 and R4 has *alive* in their heads.

alive(liane) :- breathes(liane), haspulse(liane).

3.3.2 The W operators

By combining together two Vs back to back, the W operator is obtained. Assume that C1 and C2 resolve on a common literal L within clause A to produce B1 and B2, respectively (see Fig. 3.6). L is resolved away as it appears neither in B1 nor in B2. A new predicate must be "invented" by the W operator in order to build clause A. That is why this operator is frequently claimed to be constructive.

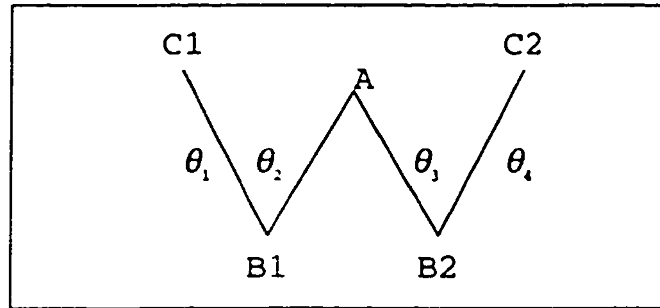


Fig 3.6. The W operator.

As before, a language bias has to be defined by each system in order to cope with the combinatorial explosion. The solution depends on the choice of the inverse substitutions, the variables for the new predicate, the syntactic structure of the clauses etc. If, for instance, C1 and C2 are assumed to be unit clauses, A can be computed by the expression:

$$A = B1\theta_2^{-1} \cup \{L\} \text{ or}$$

$$A = B2\theta_3^{-1} \cup \{L\} \text{ or}$$

$$A = B \cup \{L\}$$

where B is a common generalization of clauses B1 and B2. Two scenarios exist depending on the polarity of the literal L and are treated by two distinct operators: *intra-construction* and *inter-construction*.

3.3.2.1 The intra-construction operator

In *intra-construction* operator, the literal L of Fig. 3.6 is assumed negative in A (and, hence, positive in both C1 and C2) like *z* in Fig. 3.7. This operator not only invents L. It also produces two new clauses from B1 and B2 that have L as head.

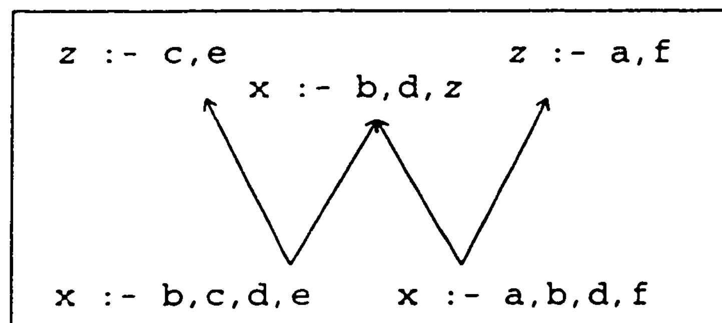


Fig 3.7. The intra-construction operator builds two clauses for the new predicate *z*.

The following example is taken from [Rouveirol, Puget 90]. Consider the clauses:

grandfather(X,Z) :- father(X,Y), father(Y,Z). (B3)
grandfather(A,C) :- father(A,B), mother(B,C). (B4)

Employing the intra-construction operator yields the clause A1 below. Note that *newp* is a new predicate created by the use of this operator. Two new, non-recursive clauses are produced for *newp*.

grandfather(L,N) :- father(L,M), newp(M,N). (A1)
newp(Y,Z) :- father(Y,Z). (C11)
newp(B,C) :- mother(B,C). (C12)

3.3.2.2 The inter-construction operator

Here the literal L of Fig. 3.6 is assumed positive in A (and negative in both C1 and C2). In the W-shaped illustration below, *z* is invented after common literals in B1 and B2 are extracted. Observe how different clauses are interestingly invented by this and the previous operator from the same pairs of clauses *x :- b,c,d,e* and *x :- a,b,d,f*.

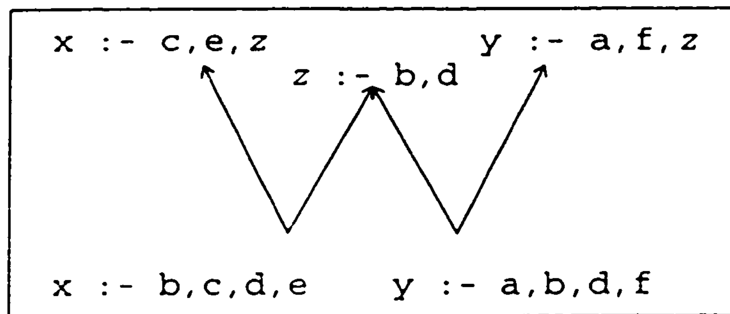


Fig 3.8. The inter-construction operator builds one clause for the new predicate *z*.

Given the same clauses B3 and B4 as above, this operator builds the three clauses below where *newp2* is a newly invented predicate. Note how they differ from A1, C11 and C12 above.

newp2(L,N) :- father(L,N). (A2)
grandfather(X,Z) :- father(Y,Z), newp2(X,Y). (C13)
grandfather(A,C) :- mother(B,C), newp2(A,B). (C14)

3.3.3 Saturation

A useful operator is proposed in [Rouveirol 90] in order to solve a limitation of absorption. The problem is that absorption is *destructive* in that the complementary literals are dropped. Not

having those literals anymore, the new clauses cannot reuse them on further generalizations. An example taken from [Rouveirol, Puget 90] illustrates the problem. Consider the two background clauses C15 and C16 and the positive example C17 below:

A :- B, C, E. (C15)
B :- C, D. (C16)
F :- C, D, E. (C17)

Note that the first two clauses share a variable in their bodies, *C*, so both could be used to perform absorption with C17. The absorption of C17 with C16 removes the literals *C* and *D* from C17, yielding clause C18 below. But this prevents one more absorption to be tried with the first clause, which would yield clause C19. As can be seen, not all possible generalizations can be found as the outcome depends on the order of application of this operator.

F :- B, E. (C18)
F :- A. (C19)

Saturation is an absorption-like operator that keeps all replaced literals in the body of the clause, enclosing them between brackets. So they can be used in subsequent steps, which allows alternative solutions to be discovered later. Referring again to the clauses above, the saturation of C17 and C16 would be:

F :- [C, D], B, E. (C20)

Note that *C* and *D* are kept in the clause. Applying saturation again, we obtain clause C21 as wished.

F :- [C, D, E, B], A. (C21)

This clause represents in a compact way all possible generalizations of clause C17. These generalizations can be obtained by removing some of the bracketed literals. The sixteen possible clauses are:

F :- A.
F :- C,A. *F :- D,A.* *F :- E,A.* *F :- B,A.*
F :- C,D,A. *F :- C,E,A.* *F :- C,B,A.* *F :- D,E,A.* *F :- D,B,A.* *F :- E,B,A.*
F :- C,D,E,A. *F :- C,D,B,A.* *F :- C,E,B,A.* *F :- D,E,B,A.*
F :- C,D,E,B,A.

So saturation overcomes the destructive nature of absorption, so that further generalizations can be carried out as they do not depend on the choice of the clauses involved in the first steps. The saturation of a positive example e is a least general clause that is logically equivalent to e with respect to a set of clauses.

3.3.4 Other operators

Two other operators defined in the system Duce are only mentioned here.

1. *Truncation* - A special case of the W operators when B1 and B2 are the empty clause $\square$ (and A, C1 and C2 are unit clauses).
2. *Dichotomization* - An operator that searches for common symbols within the bodies of both the positive and negative examples.

3.4 Inverse Entailment

Mode-Directed Inverse Entailment (IE) is an inductive inference rule proposed in [Muggleton 95] that is a direct consequence of the definition of the ILP problem. Recall from Section 2.3 that an induced program P is correct if $K \wedge P \models e^+$ (and $K \wedge P$ is consistent) for any background knowledge K and any positive example e^+ that is not logically implied by K alone. By the Deduction Theorem (Section 2.2), this condition can be expressed as:

$$K \wedge \sim e^+ \models \sim P.$$

If we interchange both sides, we obtain:

$$\sim K \vee e^+ \models^{-1} P$$

or better,

$$K \rightarrow e^+ \models^{-1} P \qquad \text{(E4)}$$

which is known as the *inverse entailment* of the original expression.

As an example of this inference rule, consider the clauses induced for the examples and

the background knowledge shown in Table 3.2 (taken from [Muggleton 95]).

Table 3.2. Background knowledge, example and programs induced through IE.

K	e^+	P
{*anim(X) :- pet(X),* *pet(X) :- dog(X)*}	*nice(X) :- dog(X).*	*nice(X) :- dog(X), pet(X), anim(X).*
{*sentence([],[])*}	*sentence([a,a,a],[]).*	*sentence([a,a,a],[]) :- sentence([],[]).*

By using the notion of bottom set from Section 2.3, we can come up with a procedure to compute the induced program. A program P is derived by the IE rule from an example e^+ and background knowledge K if P θ-subsumes some ground clause G that is a disjunction of literals in the bottom set of e^+ under K. For instance, let e1 and K1 be as follows:

{*pet(X) :- cat(X),* (K1)
cuddly-pet(X) :- small(X), fluffy(X), pet(X)}
cuddly-pet(X) :- fluffy(X), cat(X). (e1)

Let *garfield* be a constant symbol that stands for the variable *X*. The bottom set of *e1* under K1 is given by:

Bot(e1,K1) = {*small(garfield), ~fluffy(garfield), ~cat(garfield), ~pet(garfield)*}.

Therefore, clause C22 below is inversely entailed from K1 and e1:

small(X) :- fluffy(X), cat(X) (C22)

because it subsumes the ground clause C23 below. Note that C23 is a disjunction of three literals that appear in the bottom set above.

small(garfield) :- fluffy(garfield), cat(garfield) (C23)

The IE approach has led to the successful development of a general-purpose system called Progol ([Muggleton 95]). For each example, Progol induces a clause C that is later used to guide a search through clauses on the θ-subsumption lattice that subsume C. A refinement operator inspired by Shapiro's MIS system ([Shapiro 83]) helps constrain the search. Also, mode declarations must be provided by the user for all predicates involved (target and background

relations). The modes indicate the input/output nature of both the head and body literals of the target clause. Types must also be given and defined for all arguments. All clauses constructed by Progol are depth-bounded (Section 2.5).

Expression E4 suggests that it is possible to induce a program from background knowledge and positive examples only. Even so, negative examples are often necessary to rule out over-general clauses. A new version of Progol is presented in [Muggleton 96]. Progol4.2 uses a Bayes' method for the posterior probability of hypothesis given positive examples only in order to guide the search. Experimental results show that Progol4.2's predictive accuracy when learning from positive instances only is comparable to learning from a mixture of positive and negative instances.

So the idea behind Inverse Entailment is to move away from resolution-proof theory— and as a consequence from the problem of inverting implication—, and approach the ILP task from the viewpoint of model-theory, which underlies proof, by regarding inductive inference directly as the inverse of deduction.

In a recent publication, [Yamamoto 96] proved that Inverse Entailment is not complete by showing some correct programs that cannot be induced by this inductive inference rule. Nevertheless, if restrictions are imposed on the class of derivable hypotheses, IE can be made complete (see [Yamamoto 97] and [Furukawa et al. 97]). Incidentally, a complete method of inverting implication between clauses that is based on Inverse Entailment using the Deduction Theorem of Section 2.2 does exist, but only if the clauses are limited to function-free definite clauses ([Muggleton 95]). The problem of inverting implication between clauses in general is discussed in the next section.

3.5 Inverse Implication

Inverse implication has lately become one of the main trends in ILP, especially due to its fundamental role in inducing recursive clauses. If implication between clauses can be properly inverted, the completeness of inverse resolution will be enhanced since θ-subsumption is used in place of clausal implication. The good news is that methods to deal with some of the problems with clausal implication have been proposed, which led to the recent development of various

learning systems based on inverting implication. This section analyzes one such method and presents three inverse implication-based learners that form the basis for the present research.

Implication's undecidability problem can be overcome by reducing this operation to the more tractable θ-subsumption operation. [Idestam-Almquist 95] shows an interesting technique for carrying out the reduction that is based on the notions of expansion and or-introduction as explained below.

Consider again clause R as defined earlier in Section 3.2:

$$(C1 - \{L1\})\theta_1 \cup (C2 - \{L2\})\theta_2 \qquad \text{(R)}$$

If we assume that both θ_1 and θ_2 are empty, which means that no substitutions have been done in the resolution of C1 and C2, and that C1 - {L1} = C2 - {L2}, which corresponds to the assumption that every literal in R is inherited from both C1 and C2, we have a way to induce the two parent clauses. It suffices to make:

$C1 = R \cup \{L1\}$ and

$C2 = R \cup \{\sim L1\}$.

Note that L2 = ~L1 given the assumptions above. In this case, we say that C1 and C2 are obtained from R by an *or-introduction* of the literal L1 ([Idestam-Almquist 93b]). If a set of literals is given, several pairs of parent clauses can be constructed for each literal in the set. Resolution can, as a consequence, be inverted by using this operation as shown in [Idestam-Almquist 95].

An *expansion* of a clause C with respect to a set of literals S is a clause E that is the lgg under θ-subsumption of a set of clauses or-introduced from C by S. The expansion of a clause is logically equivalent to the clause itself. That is, implication can be reduced to θ-subsumption if we replace a clause by its expansion. Consider, for instance, clauses C24 and C25 below (taken from [Idestam-Almquist 95]):

p(f(X)) :- p(X). (C24)
p(f(f(f(a)))) :- p(a). (C25)

It is clear that C24 $\rightarrow$ C25 but we do not have that C24 $\preceq$ C25. Now consider the following set of clauses that are or-introduced from clause C25:

$p(f(f(f(a)))) \text{ :- } p(f(f(a))), p(a).$ (C26)
$p(f(f(f(a)))), p(f(f(a))), p(f(a)) \text{ :- } p(a).$ (C27)
$p(f(f(f(a)))), p(f(f(a))) \text{ :- } p(f(a)), p(a).$ (C28)

The lgg under θ-subsumption of {C26, C27, C28} is clause C29 below.

$p(f(X)), p(f(f(f(a)))) \text{ :- } p(a), p(X).$ (C29)

But now it is clear that C24 θ-subsumes C29, the expansion of clause C25. So, by using or-introduction implication can be made decidable through a reduction to θ-subsumption. But how general is this technique?

It would be nice if every generalization under implication could be reduced to a generalization under θ-subsumption. An expansion E of a clause C is called a *complete expansion* or a *finite self-saturation* of C if and only if, for every clause D, D $\preceq$ E whenever D $\rightarrow$ C. Unfortunately, because some clauses have infinitely many distinct generalizations under implication, complete expansions do not exist for all clauses ([Muggleton, Page 94]) so this technique is not general enough.

The framework for generalization under implication is computationally expensive. Therefore, restrictions have to be imposed on the hypothesis language in order to make inverse implication efficient and useful. Two main attempts have been made to develop learning systems that are capable of inverting implication under a heavy bias. These learners, precursors of the present work, are introduced in the next two sections (see also the discussion on CLAM in Section 5.1). Other inverse implication-based methods are described in Sections 4.1 and 4.3 and also in [LeBlanc 94] and [Muggleton 95]. General properties of implication between clauses can be found in [Muggleton 92a].

3.5.1 LOPSTER

The first system to pursue learning recursive definitions from a few examples was LOPSTER. The idea, described in [Lapointe 92] and in [Lapointe, Matwin 92], is to use inverse implication

on the examples in order to induce the recursive clause. The assumption is that either the base clause is a member of the training set, or the examples are in the same resolution chain. Their work introduced such concepts as sub-unification and generating terms. LOPSTER is capable of learning both purely recursive and left-recursive definitions.

To learn a purely recursive definition, LOPSTER first builds all generating terms for each argument of the positive examples. Then the system constructs the recursive clause in the following way. The msgt (Section 2.4) of each argument will derive the arguments in the head, while the generating variables will form the recursive call. For instance, to learn a definition of *append*, given two examples: the base clause *append([ ],L,L)* and *append([a,b,c], [l,t], [a,b,c,l,t])*, LOPSTER computes the msgt for each of the three arguments, obtaining *[X1|V1]*, *V2* and *[X2|V3]*, respectively. The head and the body of the recursive clause will be *append([X1|V1], V2, [X2|V3])* and *append(V1, V2, V3)*, respectively, resulting in the following definition:

append([ ],L,L).
append([X1|V1], V2, [X2|V3]) :- append(V1,V2,V3).

The variable *X2* is later replaced by *X1* since they have identical sequences of instantiations (*a*, *b* and *c*) when this definition is executed on the second example.

The induction of a left-recursive definition is done in a similar way, by applying the same method to the input variables of the examples. This results in an incomplete clause that looks like the one below. Here the target is to learn a definition for *multiply* given as examples the base clause *multiply(X,0,0)* and $multiply(s^2,s^3,s^6)$. The modes are *[i,i,o]*.

multiply(V1,s(V2),V3) :- multiply(V1,V2,W3).

To bind the variable *W3* to the output variable in the head, *V3*, LOPSTER searches the literals in the background knowledge. Eventually, it will come across the literal *plus(V1,W3,V3)* and the correct definition is found that covers all positive examples and no negative examples:

multiply(X,0,0).
multiply(V1,s(V2),V3) :- multiply(V1,V2,W3), plus(V1,W3,V3).

The algorithm is based on choosing two examples at a time to do the learning. It should always be possible to find their msgt if they are in the same resolution chain. In case they are not in the same chain of recursive calls or the induced definition does not pass the coverage test, two other examples are randomly chosen and the same process is re-applied.

The reason why right-recursive definitions (RRDs) cannot be induced lies in the type of sub-unification used by LOPSTER. For learning purely recursive definitions (PRDs), the system makes use of ordinary sub-unification, but to learn left-recursive definitions (LRDs) a variant called *input sub-unification* is used. *Input sub-unification* differs from the ordinary operation in that it considers only the input arguments of the examples. So the head and the recursive call will be completed only with respect to the input arguments. As seen before, the output variables are bound through the addition of relations from the background knowledge. Because there is no variant for inducing right-recursive clauses, only PRDs and LRDs can be learned.

The approach has pros and cons. The main drawback was mentioned earlier: the user must give the system either the base clause itself or two clauses that are just a few steps away in the resolution chain (like P1 and P2 in Fig. 3.9). That is, the user has to have a good idea in advance of how the target clause looks. The advantage is that the algorithm is extremely efficient and requires just a few examples. A comparison with two previous learners, FOIL with determinate literals ([Quinlan 91]) and Golem ([Muggleton, Feng 90]), shows that LOPSTER requires fewer examples and takes much less time than the other two systems. In the experiment, hand-tailored examples were used.

3.5.2 Crustacean

The system Crustacean ([Aha, Lapointe, Ling, Matwin 94a and 94b) is a major advance compared to the previous attempt. It relaxes LOPSTER's requirement that the examples be in the same resolution chain or that the base clause be known. The system can work with randomly selected and completely independent examples like E4 and E5 in Fig. 3.9 that are in different resolution chains. The technique used combines the subterms of the positive examples in order to generate a base clause that must not cover any negative examples. The generating term of the base clause is then used to further generate the two-literal recursive clause. The system, therefore,

requires no further knowledge of the base clause such as FORCE2 (cf. Section 4.5) and many approaches based on θ-subsumption do. But, as a consequence, Crustacean can only learn PRDs.

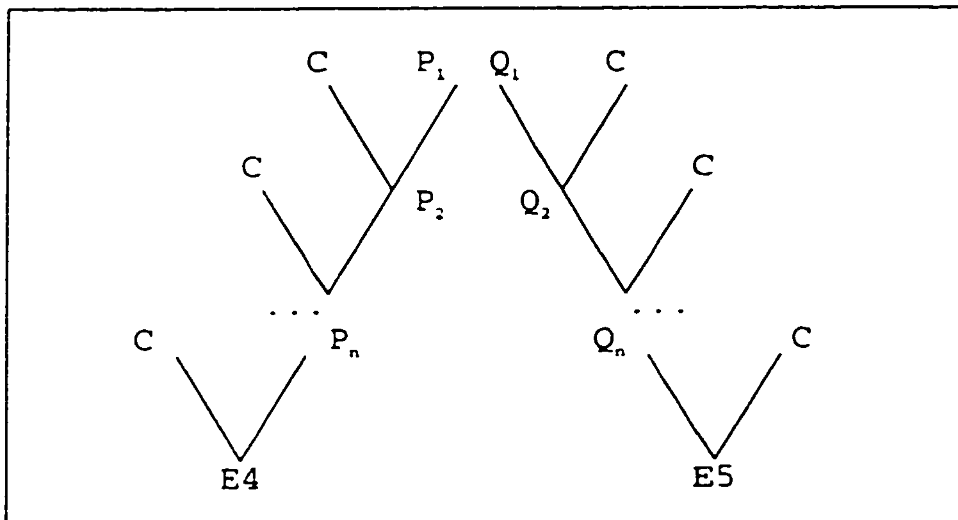


Fig 3.9. The recursive clause C resolves with two examples in different resolution chains.

Crustacean's complexity hinges enormously on the combination of the subterms. The space of combinations grows exponentially with the number of positive examples, the arity of the target predicate and the complexity of the arguments. To reduce the search through this space, constraints are imposed that discard combinations with different or useless depth. The effect of such constraints is observed by the significant reduction on the space of combinations to merely 143 from 14,225 combinations in ten relations tested.

The method is very simple. First, Crustacean generates all combinations of subterms of the arguments of the positive examples. Then, for each argument, Crustacean computes the *lgg* of those combinations satisfying the aforementioned constraints. This computation results in a small number of tentative base clauses. There is a chance that these clauses are too general due to the *lgg* operation. Crustacean discards the wrong ones by testing if they are consistent with the negative examples. The remaining clauses are, therefore, incomplete definitions of the target predicate. In the next step, Crustacean finds the recursive clause corresponding to each one of them, forming a set of PRDs. Finally, the system runs each definition on all given examples, reporting as induced clauses those that pass a coverage test.

The induction of the recursive clause for a given base clause is based on the generating term of that base clause. Crustacean produces the head of the recursive clause by *lgging* the result yielded by the application of the GT on all positive examples. Once finding the head, the system

induces the recursive call by applying the GT exactly once to the head.

When compared with Golem on randomly selected examples, Crustacean obtained a higher accuracy. The reason for that is that Crustacean has a strong language bias while Golem is a general-purpose ILP learner. Moreover, the number of positive examples used, five, was too small for the rlgg operation used by Golem to produce a correct definition.

3.5.2.1 An example

Special attention is now given to Crustacean as it is the basis for the present research. The next example shows how a purely recursive definition for the relation *length* is learned by the system. This relation, which has modes *[i,o]*, returns the number of elements in a list. We will show how the base clause is first learned and how its generating term is used to induce a recursive clause. We are going to use Shrinp's internal representation, which makes use of the function *pair*, as defined in Section 2.1, to represent lists. An analysis of the complexity of the algorithm to compute the base clause can be found in Section 5.7.1. The given sets of examples are shown in Table 3.3.

Table 3.3. Examples of *length*.

Positive Examples	Negative Examples
(E+1) *length(pair(a,pair(b,[ ])),s(s(0)))*	(E-1) *-length(pair(e,[ ]), s(s(0)))*
(E+2) *length(pair(c,[ ]),s(0))*	(E-2) - *length([ ], s(s(s(0))))*

Crustacean starts by computing the subterms, embedding terms and generating terms (GTs) of all arguments of the positive examples. For each example separately, the cross product of its subterms and generating terms is calculated in order to find combinations with the same depth. Non-matching depths are not allowed because they mean different numbers of derivation steps to prove the same example, which is impossible. The computation yields the following combinations for the first example (Table 3.4). Valid combinations are marked with a "✓". Eighteen out of 24 matches are found.

GTs with a depth of zero define no operation and are indicated by the tag "none". They match any GT because the absence of operations means that the term can be left as is no matter

what the depth of the other arguments is, i.e., no matter how deep a derivation proof is. The combination of the subterms *a* and *s(0)*, for instance, is accepted because their GTs, *pair(1),1* and *s(1),1*, have the same depth, one.

Table 3.4. Cross-product of the GTs and subterms of the arguments of E+1.

	Subterms of arg.2 ⇒	*s(s(0))*	*s(0)*	*0*	*0*
Subterms of arg.1	**GT, depth**	none, 0	s(1), 1	s(1), s(1), 1	s(1), 2
pair(a,pair(b,[]))	none, 0	✓	✓	✓	✓
a	pair(1), 1	✓	✓	✓	
pair(b,[])	pair(2),1	✓	✓	✓	
b	pair(2), pair(1), 1	✓	✓	✓	
[]	pair(2), pair(2), 1	✓	✓	✓	
[]	pair(2), 2	✓			✓

Note that the subterm *[]* of the first argument has two GTs. The same is true for the subterm *0* in the second argument. What this means is the existence of two sequences of operations to get to these subterms: one of depth one and the other of depth two. Because they have different GTs, they must be considered independently. The matches for the second example are shown in Table 3.5.

Table 3.5. Cross-product of the GTs and subterms of the arguments of E+2.

	Subterms of arg.2 ⇒	*s(0)*	*0*
Subterms of arg. 1	**GT, depth**	none, 0	s(1), 1
pair(c,[])	none, 0	✓	✓
c	pair(1), 1	✓	✓
[]	pair(2), 1	✓	✓

The next thing Crustacean does is to find the cross-product of the valid matches found for all examples. The number of groups of subterms is proportional to the number of examples, the arity of the relation and the number of subterms of the arguments. To reduce the number of possibilities, which in this example equals 108, the following constraints are used:

1. For a given argument, the GTs must be the same, i.e., the sequence of operations and selected arguments must be identical (so the same operations will be performed for that argument in all examples)

2. To guarantee generality, the summed depth of the GTs of a group of subterms must be at least two. Lower depths lead to overly specific clauses: a depth of zero implies an instance of the base clause while a depth of one implies one single recursive call.

3. As before, "none" matches any GT.

The constraints considerably reduce the search through the space of combinations. In the current case, the reduction is as high as 82% as only nineteen matches comply with the constraints. Once a group of subterms matches, a potential base clause is induced by computing the *lgg* across the arguments. Table 3.6 shows some of the combinations that match. A1 and A2 refer to the first and second arguments, respectively.

Table 3.6. Some of the matching GTs from E+1 and E+2 and the arguments A1 and A2 of the potential base clauses they derive.

GT, depth (E+1)		GT, depth (E+2)		Subterms (E+1)		Subts (E+2)		lgg	
A1	**A2**	**A1**	**A2**	**A1**	**A2**	**A1**	**A2**	**A1**	**A2**
none,0	s(1),1	none,0	s(1),1	*pair(a,pair(b,[]))*	*s(0)*	*pair(c,[])*	*0*	*pair(A,B)*	*C*
none,0	s(1),2	none,0	none,0	*pair(a,pair(b,[]))*	*0*	*pair(c,[])*	*s(0)*	*pair(A,B)*	*C*
pair(1),1	s(1),1	pair(1),1	s(1),1	*a*	*s(0)*	*c*	*0*	*A*	*B*
pair(2),2	none,0	none,0	none,0	*[]*	*s(s(0))*	*pair(c,[])*	*s(0)*	*A*	*s(B)*
pair(2),2	s(1),2	pair(2),1	s(1),1	*[]*	*0*	*[]*	*0*	*[]*	*0*
pair(2),2	s(1),2	none,0	none,0	*[]*	*0*	*pair(c,[])*	*s(0)*	*A*	*B*

For instance, in the first line the base clause induced from the subterms *length(pair(a,pair(b,[])),s(0))* and *length(pair(c,[]),0)* is *length(pair(A,B),C)*. Other generated candidates to base clause, not including the ones shown in the previous table, are: *length(pair(A,B),0)*, *length([],s(A))*, *length([],A)* and *length(A,0)*.

The next step tests the set of base clauses against the given negative examples, discarding those that cover any example. After the check, only three candidates remain: *length([],0)*,

length(pair(A,B),0) and *length(A,0)*. This new set is used for the generation of recursive clauses, which is done, for each base clause separately, in the following way. First, each positive example is decomposed *d*-1 times, where *d* equals the depth of the GT for that example in the matched combination. The decomposition sequence starts with each example and at each step the generating term is applied to the previous decomposed literal. The system generates the head of the recursive clause by computing the lgg of the set of decomposed literals obtained from all positive examples. The sequence that corresponds to the base clause *length([],0)* is indicated in Table 3.7. The head for this base clause is, therefore, *length(pair(A,B), s(C))*. The second example undergoes no decompositions as it has a depth of one.

Table 3.7. Computing the head of the recursive clause for *length([],0)*.

	Example E+1	**Example E+2**
Depth	2	1
GT	pair(2), s(1)	pair(2), s(1)
Original	*length(pair(a,pair(b,[])),s(s(0)))*	*length(pair(c,[]),0)*
1st decomposition	*length(pair(b,[])),s(0))*	no decomposition
lgg	*length(pair(A,B), s(C))*	

To build the recursive call, the "winning" GTs are applied again, this time to the head. The operation yields the term *length(B,C)*, which will be used as the clause body. Finally, the induced purely recursive definition for this particular base clause is formed:

length([ ], 0). (D1)
length(pair(A,B), s(C)) :- length(B,C).

The PRDs derived from the other two potential base clauses are induced in the same way. They are D2 and D3 below:

length(pair(A,B), 0). (D2)
length(pair(A,B), s(C)) :- length(pair(A,B),C).

length(A, 0). (D3)
length(pair(A,[ ]), s(0)) :- length([ ],0).

All definitions are individually checked for consistency with the sets of examples before being reported as correct. In this example, only definition D1 above passes the coverage test and is the (correct) definition learned by Crustacean for the *length* relation.

3.6 Discussion

This chapter showed different approaches to generalization in the ILP context. The discussion was centered on methods that invert resolution through θ-subsumption as opposed to those that invert implication. It was shown that the attempts to invert resolution have serious drawbacks. However efficient, the use of *lgg* under θ-subsumption may produce lengthy and over-generalized clauses. Among the facts that make θ-subsumption unsuitable for learning recursive definitions is that, for a clause C to θ-subsume a clause D given a set S of clauses, C has to be used exactly once in the derivation of $S \vdash C \rightarrow D$. Implication, on the other hand, is a stronger relation but has been traditionally given less attention mainly due to its undecidability.

The V and W operators were proposed to help invert resolution. Systems like CIGOL and LFP2 that use these operators do overcome such problems as the uniqueness of the solution but at the expense of a heavy language bias. Besides that, the absorption operator is *destructive* as explained in Section 3.3.3. More important, the way predicate invention is realized by the W operator does not allow for the constructive learning of recursive concepts as the invented predicate is at most useful, but not necessary. What these operators actually provide is merely a *rewriting* of a set of clauses in a more compact, readable way. The same is true for the decompositions constructed by the system INDEX ([Flach 93]). They are used to restructure a database through the creation of new relations. However, the system is not capable of inventing recursive predicates.

Attempts to invert the more powerful implication achieved different results. By finding structural regularities in terms of the given examples, both LOPSTER and Crustacean are the first systems to successfully invert implication, which is made possible, in both cases, from a small data set without any extra help from the user. As a major constraint, while Crustacean has been proven to be complete with respect to purely recursive definitions (see [Aha, Lapointe, Ling, Matwin 94a]), it is restricted to that class of logic programs. The system CLAM (Section

5.1) extends Crustacean by allowing left recursion but is not capable of inventing new predicates. An approach reported in ([Muggleton 95]) proves that it is possible to invert implication of any two clauses but the method is too costly and restricted to function-free clauses.

The system Shrinp, to be presented in detail in Chapter 5, is an extension of Crustacean that has, among other characteristics, the ability to learn several classes of recursive definitions from a few examples. The presence of background knowledge, as expected by LOPSTER and CLAM, is not necessary as the system is capable of inventing recursive predicates. While CIGOL and LFP2 rely on human intervention to evaluate the invented predicates, Shrinp requires no further help from the user besides the training set and the modes of the target relation.

Section 3.4 introduced inverse entailment. Although it is a more general approach than inverse implication, inverse entailment has various setbacks. First, it suffers from incompleteness as shown in [Yamamoto 97]. Also, Progol requires a reasonable number of positive examples in order to learn in the absence of negative examples. In several cases, negative examples must be provided anyway. The mode declarations are so detailed that the user is almost sure of how the target clause will look like. For a recursive clause, the user not only has to provide the modes for the head but also for the recursive call in the body of the clause (besides the background knowledge). Shrinp's input is by far simpler and shorter.

It has to be said that generalization only plays a part of any learning system. Of equal importance in practice are questions about the search space involved, how the hypotheses are formed and what control mechanisms are designed. All of these issues are discussed in the next chapter.

chapter **four**

4. Related Work

The breakthrough event in ILP was the introduction in the early 1980s of the Model Inference System-MIS ([Shapiro 83]). It represents the first successful approach to the inductive synthesis of logic programs from examples after the popularization of Prolog as a programming language. Since then, several other systems have been proposed ranging from specialized to general-purpose learners. This chapter is dedicated to describing the techniques used by systems specialized in learning recursive definitions of a single predicate only. Such systems are characterized by dealing with noise-free training data and by learning relations that implement some common list-processing or arithmetic function such as *member*, *append* and *plus*. In general they can more efficiently learn a recursive definition than general-purpose learners.

The chapter also provides a comparison between the different systems with respect to how they cope with the generation of the base and recursive clauses as well as the pruning of the hypothesis space. Three other important systems that are based on inverse implication have already been presented in Section 3.5. Several attempts to learn in the absence of negative examples are discussed at the end of the chapter.

4.1 Smart

An extension of Crustacean was proposed in [Mofizur, Numao 95] (a more detailed explanation can be found in [Mofizur, Numao 96]). The system Smart learns recursive definitions with one

base clause and one *n-bound recursive clause*, as explained later. The induction of the base clause is inspired by Crustacean but the approach is not based on inverse implication but rather on a top-down fashion. Like David Aha and colleagues' system (Section 3.5.2), Smart generates the lgg of the cross product of all subterms of the arguments of the positive examples. The novelty is the idea to eliminate wrong base clauses by using as negative examples the other candidates to the target base clause. This approach is only possible because of Smart's language bias, which restricts the learned definitions to having only one base clause. At the end of this quadratic algorithm, the correct target base clause has a great chance of being found.

The induction of the recursive clause is done in a top-down fashion similar to that used by the system CLAM (Section 5.1), i.e., by adding literals from the background knowledge. The process stops as soon as a clause is found that is consistent with the positive and negative examples. A recursive call can be added at any step, which allows for both linear and non-linear recursive definitions to be learned.

To constrain the increasing number of combinations of variables in a newly added literal, Smart uses the notion of *active variables*. A variable is said to be active in an incomplete clause if it has been used only once in this clause. A variable becomes inactive the moment it is used as input to a determinate literal L as soon as L is introduced. The output variables of L are newly introduced active variables that "represent" the meaning conveyed by its (inactive) input variables. By doing so, Smart limits the number of variables that can be arguments of a newly added literal to only those which are still active. Given a bound n, this number is kept to a maximum of n active variables. Hence, the name n-bound recursive clauses.

Smart learns a much larger class of predicates than its predecessor, Crustacean, as a result of the search on the hypothesis space of literals from the background knowledge. Nonetheless, the system has a serious drawback. It is not capable of learning all classes of left-recursive definitions (LRDs) since the base clause is not relearned as a result of the addition of the literals. Section 3.5.3 shows the example of a predicate that Smart would fail to learn.

The experiments reported in [Mofizur, Numao 95] involved only the use of carefully selected examples. No comparison with randomly generated examples or other systems was

provided. The experiments confirmed the ability to learn such classic relations as *quicksort*, *append* and *reverse*.

4.2 SKILit

The system SKILit ([Jorge, Brazdil 95]) uses a method called *iterative bootstrap induction* to learn several classes of recursive clauses. The clauses learned at each step serve as input to the next step, hence the name iterative. The method involves four nested levels, or loops (see Fig. 4.1). The outer level refines a set of clauses, called a theory. The initial theory is, by default, the set of positive examples. The clauses generated at each step are added to the background knowledge K, which is then used as input to the next step. In a sense, it is as though the theory were refined over time. The last theory found, when no more refinements are possible, is taken as the set of induced clauses.

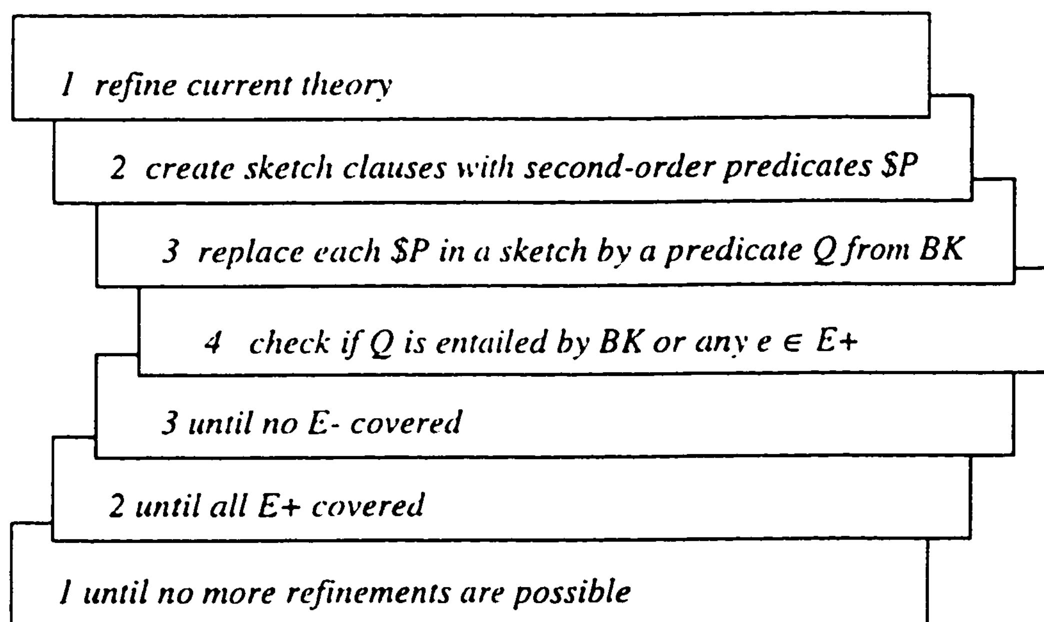


Fig 4.1. SKILit's algorithm is based on four nested loops (levels).

At level two, *sketch clauses* are continually generated until all the positive examples are covered. Sketch clauses are clauses that resemble FOCL's *relational clichés* ([Silverstein, Pazzani 91]) in that they have *variable literals*, i.e., literals whose name and arity are not yet known (also known as second-order literals). It is at level three that predicates in K are substituted for the variable literals in each sketch clause. This is implemented by a breadth-first

search on the space of combinations of predicates. For each chosen literal, the system searches the correct combinations of variables. To restrict the combinatorial explosion of this search, two constraints are employed. First, a literal is chosen only if it is semantically entailed by K or any of the positive examples—that constitutes the fourth level *per se*. The other constraint is imposed by a *dcg*-like (*definite clause grammar*, [Pereira, Warren 80]) set of rules that tells what predicates are acceptable at each place in a clause. This way, only literals of the same type as the variable literal can replace it.

The dcg used is a simplified version of the grammar presented in [Cohen 94]. As an illustration, Fig. 4.2 shows that the body of a program for the predicate P can be one or more decomposition literals followed by one test literal, one or more recursive calls to P and a composition literal.

```
body(P)       :- decompose, test, recursion(P), compose.
decompose  :- decompose_lit.
decompose  :- decompose_lit, decompose.
recursion(P) :- P.
recursion(P) :- P, recursion(P).
```

Fig 4.2. An example of the set of rules that SKILit uses as a language bias.

SKILit has several advantages. As no restrictions are imposed on the size of the theories, the induced definitions can have an unlimited number of clauses, as long as they do not cover any negative examples (guaranteed by level 3). That means that it is possible for more than one base clause to be learned. Also, depending on how the dcg is written, a single clause can have more than one recursive clause, which allows for such predicates as *quicksort* to be learned. The dcg also serves to constrain the search for predicates in level 3. As a result, SKILit learns all classes of recursive definitions as constrained by the grammar when given a reasonable number of training examples.

However, the system has some drawbacks as well. The search at the third level becomes more and more time-consuming as K grows larger, even with the aid of the dcg. In addition, the fourth level, not detailed in [Jorge, Brazdil 95], involves another loop that is clearly exponential. Besides, for a small number of randomly selected examples, it may be that no recursive clauses

are generated, as experiments done by the authors show. On relations like *append* and *delete*, none of the induced clauses, the number of which was not specified, was recursive. No definition of *append* can be considered a valid definition at all if it is not recursive. Even so, the accuracy on the testing set, whose contents are unknown, was surprisingly high, around 75% in both cases, when only two positive examples were used. The number of times the correct definition was induced in the experiments is not known. The system is actually known for its low accuracies against arbitrary sets ([Flener, Yilmaz 97]). A more recent version of SKILit that uses a Monte Carlo method for verifying consistency ([Jorge, Brazdil 96], see also discussion in Section 4.10) suffers from the same problem and requires voluminous input from the user, which is quite surprising as usually the larger the evidence, the higher the accuracy on unseen cases.

4.3 TIM

Another system that learns recursive definitions of a single predicate is TIM, The Induction Machine ([Idestam-Almquist 96]). TIM is limited to learning linear RRDs only. Unlike Crustacean, which analyzes the structure of the positive examples, TIM bases its analysis on structural regularities of the saturations of these examples (see Section 3.3.3). The computation of the saturations and this search may well be costly, but in a comparison with Crustacean, TIM obtained very promising results.

The regularities are expressed in terms of paths of variables from the head of a saturation down to the literals in its body, known as *path structures*. By definition, in a path structure the output variable of a literal serves as input to the next literal. The fact that the saturations are restricted to ij-determinate clauses guarantees that at least one such path structure exists in every saturation. Some of the path structures contain repetitions of the same pattern or define a direct relationship between the input arguments in the head and a literal in the body. These will lead to instantiations of the recursive and the base clauses, respectively. The base clause is induced by computing the lgg of the instantiations of the base clause. Likewise, the target recursive clause is learned by finding the lgg of the instantiations of the recursive clause as implied by the path structures. But before the lgg is computed, TIM adds to the end of each saturation the instantiation of the base clause for that saturation. After the computation of the lgg, this new

literal will become the recursive call, which explains why the class of predicates learnable by TIM is restricted to linear RRDs only.

The search for common structural regularities can be expensive. The construction of the saturations requires the computation of the rlgg of the positive examples relative to the background knowledge. Even with the use of ij-determinate literals, the computational complexity of this process is doubly exponential. The search for the common path structures is even more time-consuming. In order to cope with some of these problems, TIM uses a technique called *approximate rlgg computation* ([Idestam-Almquist 96]) that efficiently finds the rlgg, although not always yielding the correct answer.

The results of empirical experiments ran with TIM are encouraging. In a comparison with Crustacean, TIM got a better accuracy on 100 randomly selected testing examples of five predicates (*append*, *delete*, *last_of*, *member* and *plus*) when given five positive and ten negative examples for training. While Crustacean achieved an average accuracy of 97.6% on *last_of*, for instance, TIM obtained 100%. However, the execution times for Crustacean were significantly lower due to the larger hypothesis space searched by TIM.

4.4 Chao's System

A system developed by Li-Jen Chao ([Chao 94]) uses a rather different approach, in which only a subset of the positive examples is used to decide whether to induce a recursive or a non-recursive definition. A test on the number of sub-unifiable arguments tells what kind of definition is to be learned. If they all sub-unify, a purely recursive definition is pursued. Otherwise, the system starts a top-down search for literals in the background knowledge. The hypothesis space is restricted by imposing a maximum size on a clause and by the use of clause schemes.

The schemes define what sort of predicates to look for and how to constrain the argument binding. They correspond to definitions that start by binding either the input or the output variables in the head of the clause, before the recursive call is made. They also allow clauses that have a splitting literal followed by two recursive calls and a merge literal—a *divide-and-conquer* approach. Once a clause is shaped, it is executed on the positive examples in order to yield the base clause. The system, as a consequence, learns the classes of linear and two-clause binary

recursive definitions.

4.5 Force2

The system Force2 ([Cohen 93 and 95a]) learns a restricted class of recursive definitions known as the class of "closed" linear-recursive ij-determinate definitions composed of a base and a recursive clause. The recursive call must have no new variables—so it is "closed" in the sense of the variables that appeared before it in the clause (see [Cohen 93]). The ij-determinism is essential in that it makes the result of computing the rlgg unique and of polynomial size with respect to the number of predicates in the background knowledge.

The use of the relative lgg implies the presence of (extensionally defined) background knowledge with the definition of supporting predicates. Moreover, two functions must be defined by the user: BASECASE and MAXDEPTH. Given a clause, the function BASECASE tells whether it is an example of the base case of the recursion. The function MAXDEPTH returns an upper bound in the depth of the target program. The former is used to discard incorrect base clauses, while the latter helps detect infinite recursion.

As an illustration, the definitions below for these two functions can be used to learn the predicate *append*. The function MAXDEPTH sets MD - 1 as the maximum length of the first argument of *append*, where MD is a user-provided constant. Note that in *MD = LA + 1* below both MD and LA are bound when this expression is evaluated, so it should not be interpreted as an attribution. The second function tells that only candidates to the base clause that have a null list as the first argument can be accepted.

MAXDEPTH(append(A,_,_),MD) :- length(A,LA), MD = LA + 1.
BASECASE(append([],_,_)).

The method works as follows. First, Force2 divides the positive examples into two disjoint sets: B and R. The first set, B, contains those examples for which BASECASE is true. Their rlgg will form the tentative base clause BC (the rlgg is unique). Next, the rlgg of the examples in R for which BASECASE is false is computed, forming the tentative recursive clause RC. Continuing with the example of learning *append*, the tentative clauses might be BC and RC

below where *components(A,B,C)* is a relation that returns in *B* the head and in *C* the tail of the list *A*:

append(A,B,C) :- components(B,D,E),null(A),A=C. (BC)
append(A,B,C) :- components(A,D,E),components(C,F,G), D=F. (RC)

if the background knowledge contains the definition of the relations *null(X)*, which is the same as *A=[ ]*, and *components(A,B,C)*, which tries to unify *A* with *[B|C]*. Note that BC is incorrect and RC lacks the recursive call.

The next step is the search for the recursive call. There are several possibilities of building it according to the combinations of variables in RC. The system tries each one of them until a clause is learned that covers no negative examples. For instance, adding the (correct) recursive call *append(E,B,G)* to RC above yields:

append(A,B,C) :- components(A,D,E), components(C,F,G), D=F, append(E,B,G). (RC')

Once RC is extended, Force2 tries to correct both the base clause and the recursive clause. This is done by picking each positive example at a time and computing its rlgg with either BC or RC. This choice will depend on which set it belongs to, either B or R. After an example is rlgged with RC', the system simulates the execution of the new RC' on it. This leads to a recursive call to *append* as the last literal in RC'. The recursive call is again treated as a positive example that may belong to either B or R. If BASECASE finds that it belongs to R, the same process is applied, that is, the rlgg is computed and a new call to *append* is analyzed; but in case BASECASE tells that it belongs to the set B, Force2 rlggs BC against it, producing a new base clause, BC'. The correction of the base and the recursive clauses is, therefore, carried out at the bottom of a sequence of executions of the recursive clause. When this process is applied to all positive examples, a new definition is found. The coverage test assures that it is the correct one.

To illustrate, suppose BC and RC' as above and the example *e = append([2,1], [], [2,1])* from R. Its rlgg with RC' does not change RC'. When executing RC' on *e*, the learner comes across the new non-BASECASE call *e' = append([1], [], [1])*. RC' is now rlgged with e', which does not change it. The new call now is *e'' = append([], [], [])*, for which BASECASE is true. Now, BC is generalized to:

append(A,B,C) :- null(A), B=C. (BC')

The new definition, formed by the clauses BC' and RC', covers all positive and no negative examples. Therefore, a correct definition of *append* is successfully found. But what happens when an incorrect recursive call is added to the original RC? Incorrect definitions are discarded by the fact that they either loop, always generating the same recursive call, or over-generalize the base clause. The former is detected by the function MAXDEPTH and the latter by the presence of negative examples. This way, with a sufficient number of examples, Force2 is capable of correctly learning programs restricted to the class discussed earlier.

One of the main strengths of this system is that it is proven to learn against any distribution of examples in polynomial time[9] without using queries. Nonetheless, the method suffers from a problem similar to LOPSTER's. While LOPSTER requires the base clause to be given as an example, Force2 needs a function to distinguish potential examples of a base case. This serious concern decreases the number of possible applications of the system as the user is supposed to know the base case in advance.

[Cohen 93] reports that the system performed well in a comparison with FOIL4 ([Quinlan 90]). It obtained an accuracy of 97.4% when given 21 random examples of the predicate *multiply*. In the same conditions, FOIL4 got 93.9%. The accuracy was computed by using cross-validation. Experiments also show that the time taken by Force2 is linear with the number of examples. Furthermore, it is less sensitive to the distribution of random examples than FOIL4.

4.6 Champ

The system Champ ([Kijsirikul, Numao, Shimura 92]) is one of the first systems to invent new necessary predicates. It learns all kinds of linear recursive definitions when provided with background knowledge; otherwise, only left-recursive definitions are induced. Champ first uses a FOIL-like approach ([Quinlan 91]) to generate from the given examples an incomplete recursive clause. An algorithm called Discriminant-Based Constructive Induction (DBC) is responsible for

[9] So Force2 is a PAC-learning algorithm according to Valiant's theory of learnability ([Valiant 84], [Džeroski, Muggleton, Russell 92] and [Kietz 93]).

adding a new literal to that clause in such a way that the new clause completely discriminates between positive and negative examples. Champ also finds a definition for the new predicate by using the same method recursively. That explains why the complete discrimination is mandatory.

The discrimination is achieved by the correct choice of variables for the new predicate. The variables employed are guaranteed to be *a* minimal set of variables that occur in the incomplete clause that completely discriminates the examples. In order to do that, the DBC algorithm initially creates a new literal with all the variables in the clause. Then, by repeatedly removing from the new literal a variable at a time, it checks whether the clause still covers all positive and no negative examples. If so, the irrelevant variable is permanently discarded. For the sake of efficiency, *the* minimal set of variables is not pursued. Being linear, DBC does not test all the possible combinations of variables in the incomplete clause. Consider, as an example, the literal *newp(x,y,z,w)* and suppose that both *newp(y,z,w)* and *newp(x,w)* discriminate between positive and negative examples, while *newp(z)* does not. The algorithm tests the variables from left to right. So, *newp(x,z)*, which has fewer variables than *newp(y,z,w)* will never be considered since x is first removed from the original literal. Hence, the DBC algorithm saves time at the expense of not yielding the best (shortest) new predicate as the algorithm is sensitive to the order of the variables. The final number of variables is a local minimum.

Champ works as follows. First, the Selective Inductive Component, called Cham, tries to induce a clause that passes the coverage test. For that purpose, Cham makes use of literals in the background knowledge and special operations, called *refinement operations* ([Shapiro 83]), that allow for instantiating a head variable to a function, unifying variables and adding a literal to a clause. To choose a literal, Cham applies a FOIL-like approach that combines information gain and likelihood ([Kijsirikul, Numao, Shimura 91]). If no clause can be found, the best incomplete clauses generated, as given by another information gain-based heuristics, are selected to be specialized with a new predicate. The DBC algorithm provides the choice of variables in the new predicate as well as its positive and negative examples. At this point, the system calls itself recursively to learn a definition of the new predicate. The same process is then reapplied to the new predicate, which may require the definition of a new sub-predicate, and so on. The chapter on predicate invention comments more on the DBC algorithm.

Champ's main drawback is that it needs a complete set of ground examples (up to a certain level of recursion) for the DBC algorithm to work. If some examples are missing, the system may end up with a final set of variables that is larger than the set that would have been produced from a complete set of examples. As a consequence, this situation may ruin the process of finding examples for the new predicate and, therefore, the search for its definition. Besides, Champ still relies on user-supplied predicates. In case the background knowledge is not provided, the Selective Inductive Component will return nothing but purely recursive clauses and, as a result, only left-recursive clauses will be induced by Champ, a critical limitation.

Experiments reported by the authors show that the system is capable of learning such predicates as *reverse*, *union* and *grandmother*. For each of them, a new predicate was invented: *concat*, *member* and *female*, respectively.

4.7 SIERES

A system that combines both the use of background knowledge and the invention of predicates is presented in [Wirth, O'Rorke 92]. Named SIERES, the system is based on inverse resolution and on a generalization operation called the *least common anti-instance*, or *lca*[10]. It works by first computing the lca of the positive examples. This results in a list of clauses that are tested for over-generalization. Over-general clauses are syntactically detected (and then discarded) by looking for unbound output variables. The specialization process of the remaining clauses is achieved through the addition of literals from the background knowledge. The search is guided by the need to bind all variables in the body of a clause in a similar way to CLAM's restrictions (cf. Section 5.1). Some heuristic constraints, which are based on argument dependency graphs, help prune the search space. Such constraints also help select the arguments of a new predicate whenever no specialization is found that fulfills these constraints. This way, several classes of recursive definitions can be learned.

Predicate invention in SIERES is attained in a Champ-like fashion: first, the examples of the new predicate are found by executing the current clause on the positive examples and keeping

---

[10]The lca is a generalization operation equivalent to the lgg. If a clause A is more general than a clause B, B is called the *anti-instance* of A, hence the name lca ([Lassez, Maher, Marriot 88]).

track of the instantiations of the call to the new predicate. Once the examples are gathered, SIERES is recursively called to learn a definition for the new predicate. There is no need for negative examples as a closed world is assumed.

One of the problems the system has lies in the number and type of examples it is given. In general, a large number of examples of the target relation have to be given for a new predicate to be invented. Another drawback is that the method employed to discover over-general clauses is not sufficient to discard all types of incorrect definitions. As a result, definitions can be learned that are incorrect since only the positive examples are used to test a definition.

4.8 SPILP

SPILP ([Martin, Vrain 97]) is a top-down system that is also capable of predicate invention. The language bias restricts the clauses to being function-free and to having at most two literals in the body. An initial ground theory, which acts as background knowledge, is given to the system. The aim is to complete the theory by iteratively adding a clause of one of the two possible forms. The process, which may need the introduction of new predicates, ends when all positive examples and no negative examples are covered.

There are two types of clauses that can be added:

p( ) :- q( ). (Type *)
p( ) :- q( ), newp( ). (Type **)

where p is the target predicate and q is chosen from the current theory, $p \neq q$. The selected literal q is the one that covers the most number of positive examples and the least number of negative examples. Clause * is added only if it covers no negative examples of p. Otherwise, clause ** is built by the introduction of the new predicate *newp* whose definition has to be found. The arguments of *newp* are formed by the union of the variables occurring in both p and q. The examples of the new relation are computed by running clause ** on the examples of p. When a definition for *newp* is later found, unused arguments are dropped. A FOIL-like approach is used to select the next clause to be specialized.

One interesting aspect of the algorithm is that only those two types of clauses can be

learned. Given that $p \neq q$ and $newp \neq q$, can SPILP learn recursive definitions? The answer is yes, although this is indirectly done. After a program is learned, the system applies a series of *simplification rules* to eliminate useless new predicates that were incidentally built. It is the result of this operation that may yield recursive clauses. For instance, consider Table 4.1.

Table 4.1. The result of the use of SPILP's simplification rules on two sets of definitions.

S1	S2	S1'	S2'
p( ) :- q( ), newp1( ).	*r( ) :- s( ), newp2( ).*	*p( ) :- q( ), p( ).*	*r( ) :- s( ), r( ), r( ).*
newp1( ) :- p( ).	*newp2( ) :- r( ), newp3( ).*		
	newp3( ) :- r( ).		

The sets of clauses S1 and S2 will give rise to the two sets S1' and S2', respectively, after being simplified. The new predicates, *newp1*, *newp2* and *newp3*, are eliminated. Note that S1' is right-recursive while S2' is non-linear. As for left-recursive definitions, recall that SPILP uses a function-free language. Therefore, this class of clauses is not produced by the system lest the learned definitions may lead to an infinite recursion.

4.9 Discussion

The systems presented above are now analyzed in three dimensions according to how the base and the recursive clauses are learned and how the hypothesis space is reduced. Due to their importance to this work, the learners LOPSTER, Crustacean and CLAM are also incorporated in this analysis.

4.9.1 Learning the base clause

Different approaches have been used to learn the base clause. LOPSTER is the only system that does not learn a base clause, as it must be explicitly given one. Crustacean, CLAM and Smart can be grouped together as they all combine subterms of the positive examples in order to induce a set of tentative base clauses. Actually, Smart generates its own negative examples in order to discard incorrect base clauses and, thus, induces only one base clause. If necessary, the base clause found by CLAM is relearned.

Force2 requires the definition of a function, BASECASE, that tells whether an example is a valid specialization of the base clause. The rlgg of the examples satisfying that function produces the base clause. Chao's system executes the induced definitions on the positive examples until the base clause is found. Only two systems learn the base clause at the same time as they search for the recursive clause, SKILit and TIM. In the process of refining a theory, a base clause is hopefully encountered by SKILit. TIM relies on simple path structures in the saturations whose lgg produces the base clause. As Champ requires a complete set of ground examples, which certainly contains the base clause, the system will eventually come across it. SIERES expects the base clause to be given or derived as a result of the lca operation. Finally, base clauses for SPILP are clauses of type * (see Section 4.8).

4.9.2 Learning the recursive clause

A top-down search for the recursive clause is common in all systems but Crustacean, which only learns purely recursive definitions. LOPSTER tries to find a link between the output variables of the recursive call and the output variables in the head of the recursive clause, by trying predicates in the background knowledge. A similar approach is pursued by SIERES, except that a new predicate can also be used to specialize a clause and the search is guided by argument dependency graphs. SKILit employs clauses with second-order predicates that will be later replaced by predicates from the background knowledge. The type of the next literal to be added has to agree with the rules of a definite-clause grammar. Without the help of such a grammar, the systems Smart and CLAM add any type of predicates to an incomplete clause. Chao's system also makes use of this strategy, but narrows the search to the sort of predicates suggested by clause schemes.

Force2 is helped by the rlgg of the positive examples for which BASECASE is false. All it has to do is to look for the right combination of variables in a unique recursive call added to the end of the clause produced by the rlgg operation. TIM also adds the recursive call to the end of the clause. The literal added is incidentally the base clause for a given saturation. When the saturations are lgged, the newly added literal will become the recursive call. Champ uses information gain to select literals from the background knowledge and to induce an incomplete

clause that is later specialized with a newly invented predicate. SPILP's recursive clauses are generated after the learned definitions are resolved together as a result of the application of the simplification rules.

4.9.3 Pruning the hypothesis space

As implemented by most of the systems shown before, the hypothesis space employed to generate a recursive clause is huge. It comprises all the combinations of literals in the background knowledge plus the target predicate itself, which can appear more than once in a clause. Another issue concerns the combinations of functions and variables on a new literal that depend on the variables already in use in a clause and the possible addition of new variables.

Let k denote the size of the background knowledge, a the arity of the target predicate, n the maximum arity of a literal ($n \geq a$) and m the maximum number of literals in the body of a recursive clause. Considering the target predicate, there are $k + 1$ possible relations to fill a literal spot. Each one of the n arguments in this literal can be chosen from the arguments in the head of the clause or in the previous literals or it can be a new variable. This gives $in + a + 1$ possibilities for each argument in the ith literal. It follows that to add up to m literals to a clause, the number of possible combinations is bounded by the exponential expression below. Note that the actual hypothesis space is actually infinite as general (complex) terms are accepted as arguments of literals.

$$\sum_{i=1}^{m} (k+1)^i (in + a + 1)^n$$

To search through this enormous space exhaustively is counter-productive. A language bias is defined by each system in order to cope with the problem. LOPSTER uses the instantiations in the head of the recursive clause obtained from the positive examples to guide the search. CLAM restricts the combination of variables in the new literals by using a series of rules. Smart restricts itself to those variables which are still active but imposes no restriction on the literals, as opposed to SKILit, which uses a definite-clause grammar and predicate types to direct the search, and Chao's system, whose schemes tell what sort of predicates to look for next. His system also sets a maximum limit to the size of a clause. SIERES uses a rather analogous

approach that is guided by schemes represented in the form of dependency graphs. Some over-general clauses are syntactically discarded by the system.

Force2 uses the rlgg of the positive examples to form an initial incomplete clause, thereby saving the time that would be spent on trying other combinations. A similar approach is done by the system TIM. The saturations of the positive examples also behave as initial incomplete clauses. The search carried out by the last two systems involves the addition of only one new literal, namely the recursive call. In turn, Champ relies on information gain to select the next predicate from the background knowledge. The number of variables in the new predicate is restricted to a minimal set by a discrimination-based algorithm. An information-gain approach is also pursued by SPILP. The algorithm has to take into account that the search space is iteratively widened by the introduction of new predicates.

This comparative study is summarized in Table 4.2. Crustacean is the only learner that does not need background knowledge. On account of this and of not being able to invent new predicates, it is limited to learning PRDs only. Currently, the majority of the systems rely on a large amount of data and on the background knowledge. Predicate Invention (PI) is attained by just three systems but none of them has means to deal with a small number of examples. The few systems that do so are limited to purely and left-recursive clauses and just one base clause and one recursive clause. Champ appears twice in the table because when it is executed without background relations, it loses the ability to learn right-recursive predicates.

The new learner proposed in this work, Shrinp (see Chapter 5), learns linear recursive clauses from a small number of examples and yet is able to invent its own predicates without relying on user-supplied information.

4.10 Learning in the Absence of Negative Examples

The previous sections showed how a system like Champ is successful in inventing predicates due exactly to an abundance of examples of the target predicate. Many other systems require a large number of instances to induce recursive definitions. Evidence-hungry approaches present a serious problem and a barrier to the usability of ILP systems as for some applications there are not enough evidence available. There are situations when the learner will have to somehow cope

with the incompleteness of the evidence or even create its own examples, particularly negative instances. The purpose of this section is to expose several attempts to learn from positive data only.

Table 4.2. Comparison of systems that learn specifically recursive definitions of a predicate.

System	# of exs.	K	Linear Recursion			Non lin. recurs.	# of BC	# of RC	PI	Search constraint
			PRD	LRD	RRD					
LOPSTER	few	Y	x	x			1	1	N	variable instantiations
Crustacean	few	N	x				1	1	N	subterms of same depth
CLAM	few	Y	x	x			1	1	N	variable use
Smart	many	Y	x	x	x	x	1	1	N	n-bound clauses
SKILit	many	Y	x	x	x	x	≥1	≥1	N	definite-clause grammar
TIM	many	Y			x		1	1	N	lgg of saturations
Chao's	many	Y	x	x	x	binary	1	1	N	clause schemes
Force2	many	Y			x		1	1	N	rlgg of E^+
Champ	many	Y	x	x	x		≥1	≥1	Y	information gain/
	many	N	x	x			≥1	≥1	Y	discrimination
SIERES	many	Y	x	x	x	x	1	1	Y	dependency graph
SPILP	many	Y			x	x	≥1	≥1	Y	information gain
Shrinp	few	N	x	x	x		1	1	Y	argument-binding chain

The system Smart (see Section 4.1) picks the correct base clause of the target definition from a series of tentative base clauses, by checking which one does not cover the other base clauses, as though the latter were system-generated negative examples. The well-known system FOIL ([Quinlan 90]) requires that the positive and at least some negative examples be supplied. If not, the latter are computed using the Closed-World Assumption (CWA). [De Raedt & Bruynooghe 93] presents the system CLAUDIEN, which induces clausal theories from databases by also making use of the CWA. One interesting aspect of the system SIERES, discussed in Section 4.7, is that, besides employing the CWA, the system dispenses the need for negative

examples for detecting over-general clauses. Over-generalization is syntactically recognized by looking for unbound output variables in a clause. In many learning situations, however, the CWA approach is not practical since it forces the user to give a complete set of positive examples. FOIL, for instance, is reported to learn the predicate *reverse* from 341 positives and 92,796 negatives ([Quinlan, Cameron-Jones 93]). To learn a recursive definition, FOIL needs near-complete sets of positive instances. Only non-recursive definitions are easily learned from sparse training cases.

Some FOIL-derived learners pursued distinct approaches. FFOIL ([Quinlan 96]) is a limited version of that system that learns definitions of functional relations. Similar to GENEX's *i-det* mode (cf. Section 5.4.1.2), it considers negative the examples that have the same sequence of input arguments but a different output argument as an element of the set of positive examples. A somewhat identical procedure is implemented by FILP ([Bergadano & Gunetti 93]), except that *multiple output* (the learning of predicates with more than one output mode) is also possible. One interesting aspect is that the system queries the user about the "polarity" of missing examples, i.e., examples the system comes across that are not members of the given sets of examples. [Mooney & Califf 95] introduced the notion of *implicit negatives*. Their system FOIDL assumes that all possible instantiations of the output arguments of a predicate for a given sequence of input values already appear as positive examples. Outputs of learned clauses are taken as negative examples if they have a different output sequence for the same input of any example in E^+, the so-called *Output Completion Assumption* (OCA). Functional relations are a special case of this method.

An alternative is presented by a more recent version of the system SKILit (cf. Section 4.2). The new system, presented in [Jorge, Brazdil 96], uses a Monte Carlo approach named MONIC for verifying consistency. The idea is to use integrity constraints and run each of them against randomly generated logical consequences (called queries) of a potential target definition. If all of these facts are satisfied by the constraints, the definition is accepted. Incorrect definitions are discarded if any constraint is violated. In fact, the integrity constraints, explained in Section 2.3, work as non-ground negative examples in that they help the system reject over-general programs. So SKILit-MONIC still needs to be provided with negative examples except that they

are not expressed in the usual manner, i.e., explicitly, but in a form that for some applications requires from the user of the system more effort to be prepared. Another problem with SKILit-MONIC is that, although correct, the system is incomplete as a direct consequence of the Monte Carlo strategy. No matter how high the number of queries, one can never be 100% sure of the consistency of a learned definition. Moreover, like its predecessor the system needs a great amount of positive examples to achieve a reasonable accuracy.

A technique that is also similar to the approach pursued in this work is implemented in the system PAL ([Morales 92 and 97]). It generates examples based on perturbing arguments of the given examples. All combinations of one, two etc. arguments of a predicate are considered for being corrupted. Restrictions are defined that prune the search space. PAL requires, for instance, that the domains of the arguments be known (e.g., the chess pieces can be pawn, bishop, king etc.). Moreover, application-dependent rules (such as the fact that in chess two kings cannot be in adjacent positions), which are in fact integrity constraints, help the generation of the examples. Needless to say, the number of examples grows exponentially with the number of arguments to be perturbed. PAL relies on the user to accept or reject an example and even to decide if it is positive or negative.

The method introduced in this work differs from all the others. Our system Shrinp uses a special algorithm, called GENEX, to deal with the absence of negative examples (cf. Section 5.4). None of the systems above generate negative examples based on the incompleteness of the set of positive examples. Indeed, the majority presupposes some form of completeness of that set either by the CWA or the OCA. As a result, chances are that they would not properly work in a situation where the number of positive examples is very low. The use of functional relations, moreover, is only part of the features offered by the GENEX algorithm, which requires no further input from the user. The system PAL requires a lot of additional information about the examples, including the domains of each argument and application-dependent rules to help it generate the new examples. Further help from the user is necessary to tell if a newly created example is positive or negative. GENEX, on the other hand, can be easily adapted to any problem, as it requires no further knowledge about the problem. Shrinp's design restricts the input from the user to the given examples. No other information is necessary, be it the domains involved or the

validity of an example. Although not as robust as PAL due to the little information it is given, GENEX still does very well in creating new, powerful negative examples. Moreover, GENEX works in a combination of three operating modes that provide the new examples with different degrees of reliability and power. Shrinp's filters also work as implicit negative instances in that they serve to discard incorrect definitions.

A more general approach is pursued by the general-purpose system Progol (Section 3.4) which learns from positive data only by using inverse entailment. A recent version, Progol4.2 ([Muggleton 96]), uses a Bayesian framework to avoid some of the pitfalls of the previous version. As long as all clauses are *range-restricted*[11], Progol4.2 can estimate the probability distribution of positive and negative examples by constructing a Stochastic Logic Program from the given examples, background knowledge and some additional information such as the types of the arguments. But as experimental results demonstrate, Progol's accuracy increases when it uses the examples from GENEX only—see the report on experiments in Section 7.3.

4.11 Summary

This chapter discussed the characteristics of several systems that learn exclusively recursive definitions of a single predicate. Special attention was given in the survey to the generation of the base and recursive clauses and the pruning of the hypothesis space. A number of different approaches to learn from positive data only were presented. The reader interested in other similar surveys is referred to such texts as [Deville, Lau 94] and more recently [Flener, Yilmaz 97]. More literature reviews can also be found in Sections 6.1 and 6.7.

Section 4.9 briefly introduced the constructive learning system Shrinp as the first inverse implication-based system to learn several classes of recursive definitions from a small number of examples without further input from the user. The system can work with positive examples only as it can create its own negative examples. The algorithm to do so does not presuppose any form of completeness of the training set. Shrinp is described with details in the next chapter.

---

[11] A clause is said to be range-restricted if every variable that occurs in its head also appears in its body.

chapter **five**

5. Clam and Shrinp

So far we have seen a number of specialized learners that learn exclusively definitions of one predicate. We have only briefly introduced the systems Shrinp and CLAM. This chapter is mainly dedicated to an explanation of Shrinp's main characteristics, the implementation, the algorithm, the parameters and options, and the automatic generation of negative examples, among other details. It is shown at the end of the chapter an analysis of the complexity of the algorithm. Due to the importance to this work, the invention of new predicates is covered in a separate chapter. Experiments with both systems are also shown separately. Let us start with a brief discussion of the system CLAM.

5.1 CLAM

CLAM ([Rios, Matwin 96a and 96b]) is an extension of Crustacean whose main point is the ability to efficiently induce PRDs and LRDs by looking for common subterms in a few positive examples and making use of background knowledge. Although allowing more than two literals in the body of a recursive clause, CLAM still has a strong language bias. Only definitions with one base clause and one recursive clause can be induced.

Like Crustacean, CLAM first tries to induce a series of tentative purely recursive definitions of the target predicate. If they fail the coverage test, they are specialized by a top-

down search on the space of combination of literals from the background knowledge in a way similar to Smart's approach (Section 4.1). However, the combinatorial explosion of variables in the new literals is restrained by the use of the following rules:

- The input variables of a literal must be the output variables of the previous literal or an input variable from the head of the clause;
- At least one output variable of the head must appear in the last literal of the clause;
- The output variables of a literal have to be either the output variables of the head or new variables;
- Every variable must be used at least twice.

These rules also play an important role in the system in that they make it easy to correct the base clause in a learned definition. As explained in Section 5.5, the mechanism for relearning the base clause is necessary as the addition of new literals to the right of the recursive call may intervene in the result of the recursive clause. The approach is, nevertheless, applicable only to left-recursive definitions.

5.1.1 An example: learning the predicate *sum*

CLAM's algorithm is explained through an example that has as target relation the predicate *sum*. The background knowledge and training set given to CLAM are shown in Table 5.1. The relation *plus* with modes *[i,i,o]* adds the two first arguments and returns the result in the third argument.

Table 5.1. Examples and background knowledge for *sum*.

Positive examples	**Negative examples**	**Background knowledge**
sum($[s^2,0]$, s^2).	*sum*($[s^1,0]$, 0).	*plus(0,A,A).*
sum($[s^1,s^1]$, s^2).	*sum*($[s^2]$, s^1).	*plus(s(A),B,s(C)) :-*
	sum($[\,]$, s^2).	*plus(A,B,C).*

CLAM searches for a common pattern in the positive examples and finds a single base clause that is consistent with the negative examples. Its generating term gives rise to the purely

recursive definition D1 (Table 5.2) which is rewritten as the incomplete definition D2. When adding a literal to D2, CLAM will eventually pick *plus(E,A,D)* from the background knowledge and add it to the end of that definition forming definition D3. To relearn the base clause, this definition is shifted, yielding D4. When D4 is tested for coverage of the first positive example, the sequence of calls *sum([s^2,0],s^2)*, *sum([0],0)* and *sum([],0)* is observed. As the same base clause is obtained for the second example, the *sum([],0)* is kept as is and definition D3 is reported as correct.

Table 5.2. Definitions generated by CLAM when learning the predicate *sum*.

Definition D1	Definition D2	Definition D3	Definition D4
sum([ ], 0).	*sum([ ], 0).*	*sum([ ], 0).*	*sum([ ], 0).*
sum([A|B],s(C)) :-	*sum([A|B],D) :-*	*sum([A|B],D) :-*	*sum([A|B],D) :-*
sum(B,C).	*sum(B,E).*	*sum(B,E),*	*plus(E,A,D),*
		plus(E,A,D).	*sum(B,E).*

Shrinp can learn exactly the same definition without the need for background knowledge. To do so, it invents the relation *plus* and finds its definition. See Section 7.1. We now introduce the system Shrinp.

5.2 Shrinp

Shrinp stands for **SH**ift, **R**ight recursion and **IN**vention of **P**redicates. It was so named after CLAM's first version's inability to learn right-recursive definitions. Shrinp extends that system mainly by inventing the predicates it needs, thereby disposing of the background knowledge, and by allowing for right-recursive clauses to be learned as well. In the process of finding the examples of a new predicate a shifting of the literals in an incomplete clause is occasionally carried out, hence the letters "*sh*". Shrinp has evolved from the three learners described in Section 3.5, as shown in Fig. 5.1. That means that the system inherits their characteristics, especially the inverting implication-based approach.

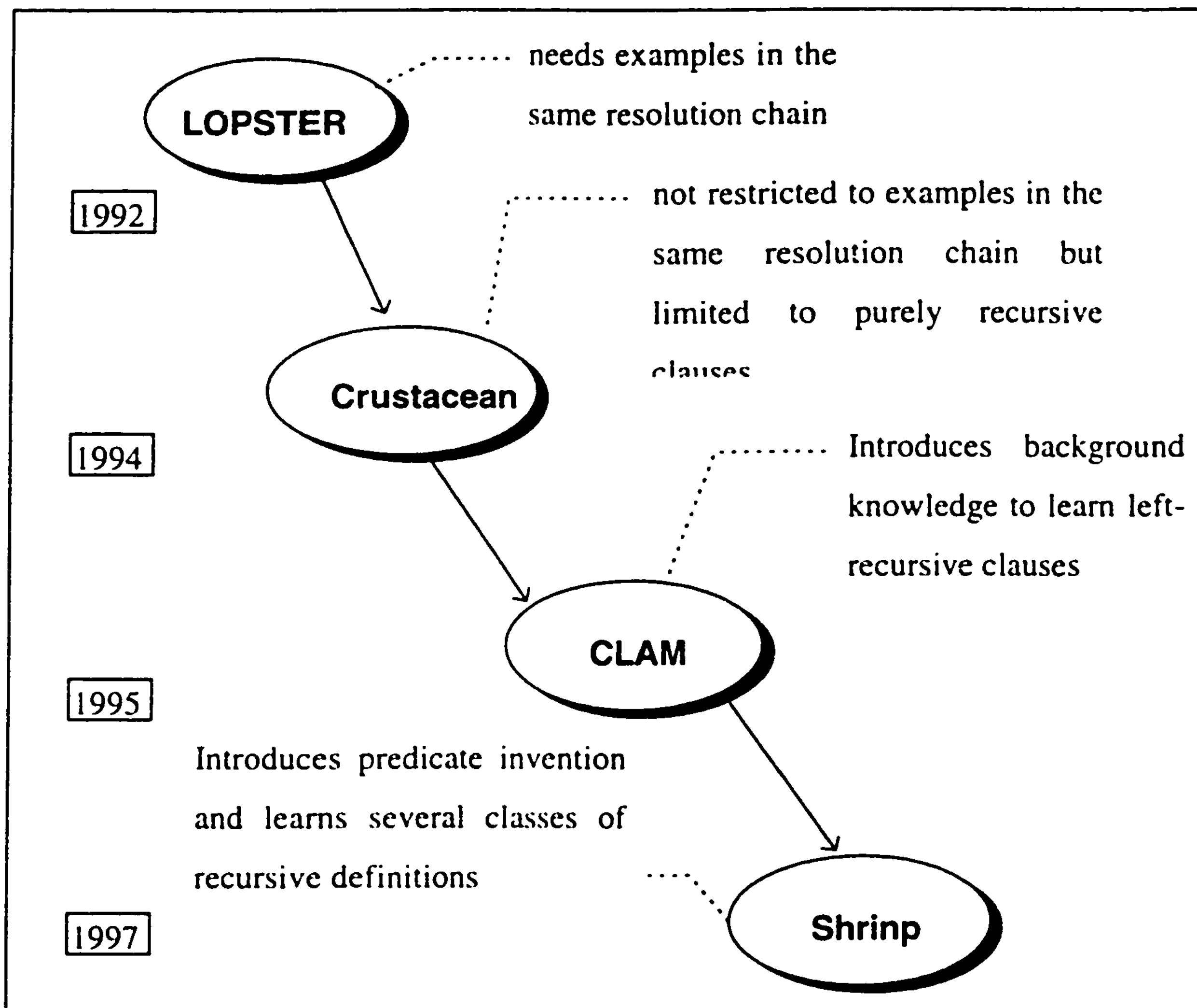


Fig 5.1. Shrinp is a descendent of three other learners.

The main characteristics of the system were already mentioned in Section 1.4, namely autonomy and the ability to learn from a few sparse examples that lie in non-intersecting resolution paths. Other characteristics are:

- The ability to learn purely, left- and right-recursive linear clauses
- The use of argument-binding chains to guide the search for the arguments in the newly invented predicate
- The ability to relearn the base clause.

There are many ways in which Shrinp differs from the other systems. To start with, it is based on inverting implication, what brings it closer to LOPSTER, Crustacean and CLAM. But

not only is it more powerful than those systems in terms of being able to learn several kinds of recursive definitions, it is also and most significantly capable of learning constructively. In this regard, it is comparable to Champ, SIERES and SPILP, introduced in Chapter 4 (see also Table 4.2), but there are many points in which Shrinp contrasts with those systems.

First of all, Shrinp does not require that *all* the ground instances of the target predicate be given as do Champ and SIERES. One of its main qualities is exactly the ability to learn from very small data sets. Secondly, Shrinp is able to learn several types of linearly recursive clauses without using background information. Champ is limited to learning exclusively purely and left-recursive clauses when no background knowledge is available. Background knowledge is also needed by SIERES and, in the form of an initial theory to be completed, by SPILP, which is limited to RRDs. In addition, Shrinp has to deal with the possible absence of negative examples of the new predicate for reasons explained later in Section 5.4. This situation will hardly ever happen to Champ and SIERES due both to the substantial number of examples required by them and to the use of the CWA by SIERES. So Shrinp is the first constructive system that learns from a small number of instances several types of recursive clauses without any help from the user.

5.3 The modules

Shrinp can be roughly divided into three main modules: the *inducer*, the *filter* and the *predicate inventor*, as depicted in Fig. 5.2. The first is responsible for the user interface: it accepts the examples and modes from the user and outputs the results. The definitions learned by the inducer, all of which are purely recursive definitions (PRDs), are passed onto the Filter Module, which tests each of them for endless loops, coverage etc. The program stops when a definition passes the screening, yielding the learned clauses. If none does, the filtered definitions are regarded as potential right-recursive or left-recursive definitions (RRDs or LRDs), that need to be completed through the addition of a new literal. It is the Predicate Inventor Module that is in charge of providing the missing literal and a definition of this new relation by recursively calling the Inducer after computing examples for the new relation.

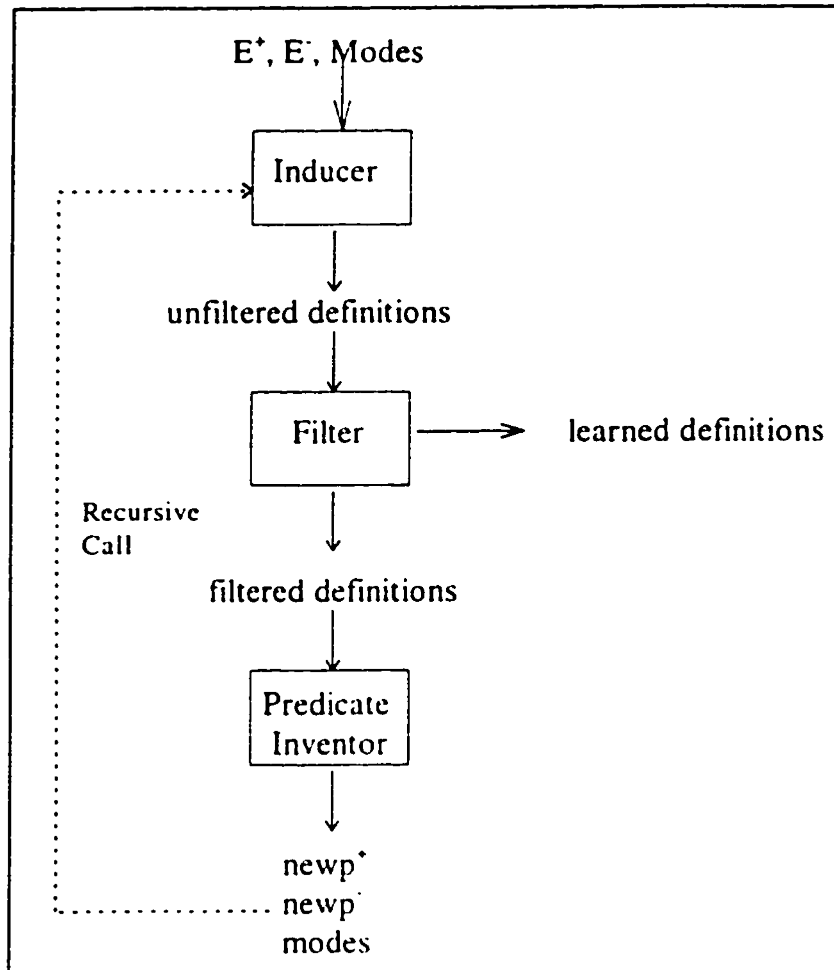


Fig 5.2. Shrinp's modules.

As can be seen, the process can be repeated indefinitely. An incomplete definition, that is, one that is not complete nor consistent with the training set, gives rise to a recursive call to Shrinp, which produces another incomplete definition that forces a new call to Shrinp, and so on recursively. The sequence of calls stops when the last definition in the chain of recursive calls is a correct PRD. In the best scenario, once such a definition is found in the bottom of the calls, all the previous incomplete definitions will get completed one by one, until the target predicate is learned. At this point, Shrinp outputs all the definitions found and stops.

The process is illustrated by the figure below. Firstly, Shrinp receives the examples of a relation *p*. The first clauses it induces (i) do not cover all positive examples. Therefore, an incomplete clause is formed and the examples of a new predicate, *newp*, are found and passed to the second activation of Shrinp. Likewise, this new recursive call in step (ii) finds an incomplete definition of *newp*. It calls Shrinp recursively again, handing the system the examples of a newer

relation, *newp'*. Finally, in step (iii) the system succeeds in learning a purely recursive definition of *newp'*, thereby breaking the sequence of recursive calls. In the next step, the activation of Shrinp in level 2 regains control and runs a coverage test on the definitions of *newp* and *newp'*. The test succeeds and, as a result, the two definitions are returned to the first activation of Shrinp, which tests the coverage of the definition of the target predicate *p* on the sets of examples *p+* and *p-* given the definitions of the auxiliary predicates *newp* and *newp'*. In step (v), the learned definitions pass the test and are output to the user. At the end of the process, Shrinp will have invented two necessary predicates without any help from the user.

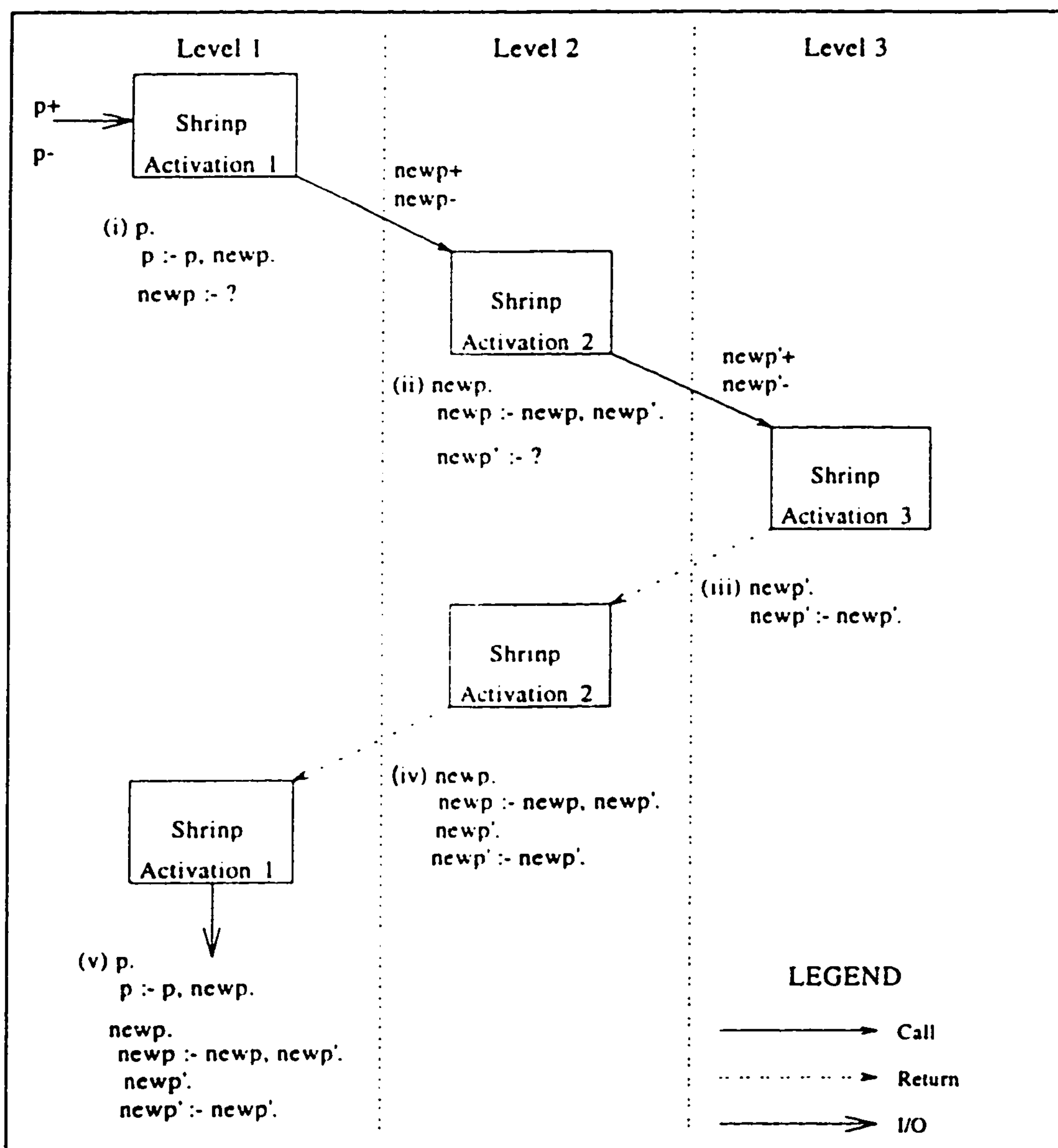


Fig 5.3. To learn a definition of *p*, Shrinp invents two necessary predicates.

In reality, the process is more complex. An activation of Shrinp can return several

definitions of a predicate at a time. It is up to the previous call in the level above to test the coverage of each of them together with its own definition. Moreover, several tentative new predicates are brought forth by each level and passed onto the next recursive activation. There is, thus, an *many-to-many* communication between two levels that is not shown in the figure. A maximum depth is also defined in order to prevent an infinite chain of recursive calls to Shrinp—see Section 5.6.1.1.

There are situations when the system fails. This happens when:

- the set of positive examples is empty or contains non-ground instances;
- the sets of positive and negative examples overlap;
- Shrinp finds no regularities on the positive examples, which is possible when none of the constraints given in Section 3.5.2.1 are satisfied;
- a definition is induced whose coverage test leads to a loop in case the filtering process is deactivated—see Section 5.3.2;
- no induced definition passes the filters;
- all possibilities of completing a clause with the invention of a new predicate fail. This situation can occur either because no definition could be found for the new relation (that is, the recursive call to Shrinp was unsuccessful for these very reasons) or because no specialization was possible for the current clause; or
- the target definition is indeed not learnable by Shrinp, that is, it lies outside the language bias defined in Section 6.2.1.

In case of failure, no definition is returned. So, for example, if the activation of Shrinp in Level 3 of Fig. 5.3 fails, the second activation of Shrinp will have to specialize the clause in step (ii) with another new predicate and call Shrinp recursively again, this time with the examples for this newly created relation. This process is done for all possibilities of completing the definition of *newp*. If none succeeds, activation 2 returns an empty list of definitions to level 1. When the failure occurs in the first level, the user is informed that Shrinp could not find a definition for the

target predicate and the execution stops.

In the next subsections the view of each module is expanded.

5.3.1 The Inducer Module

This is Shrinp's main module. It serves as the interface between the user and the other parts of the system. The inducer's principal function is to induce Horn clauses from the user-given positive and negative instances. It implements the process presented in Section 3.5.2 by first computing the subterms (STs) and embedding terms (ETs) of the positive examples, as well as the matching generating terms (GTs), as shown in see Fig. 5.4. These three sets of terms are then passed to a sub-module that induces n potential base clauses (BCs). The Base Clause Inducer runs a coverage test on the given negative instances for each base clause. Any failing base clauses are simply deleted. The process is helped by the Basic Negative Examples Generator (BNEG), which uses the other n - 1 base clauses as negative examples in this test in order to discard a potentially wrong base clause (see Section 5.4.3). The base clauses that pass the test are handed, together with their generating terms, to the Recursive Clause Generator. The last sub-module creates a recursive clause for each BC, finally forming pairs of base/recursive clauses, i.e. unfiltered purely recursive definitions, which are then passed to the Filter Module. The pairs are used individually not in conjunction with one another.

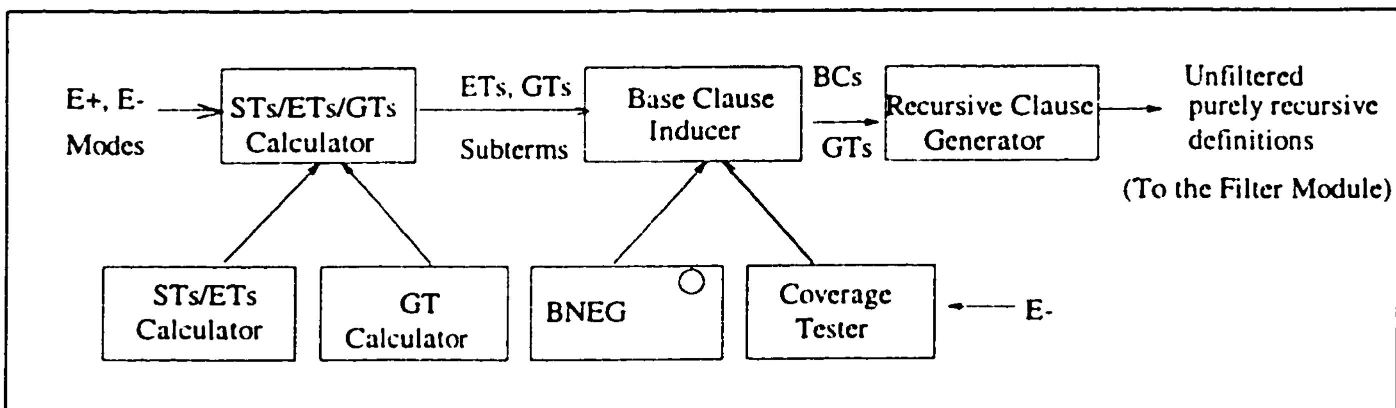


Fig 5.4. The Inducer Module.

NOTE

The execution of the modules marked with a circle on this and the next two figures may be skipped at the user's wish. There is an option, for instance, to commit Shrinp to not using BNEG, in which case the coverage test of the induced base clauses is run only on the given negative instances. The options are explained in Section 5.6.2.

5.3.2 The Filter Module

The unfiltered definitions produced by the inducer are not guaranteed to be correct. Besides possibly having useless arguments or leading to an endless loop, they may cover some negative examples or not cover all positive examples. The purpose of this module is twofold: at the same time as it screens the definitions, it also calls the predicate inventor for each definition that has passed the basic filters only (cf. Fig. 5.5). The values returned by this module are either:

- PRDs from the inducer that passed all tests; or
- complete left- or right-recursive definitions together with the definition of new predicate(s).

If none of the above occurs, the filter returns an empty list of definitions.

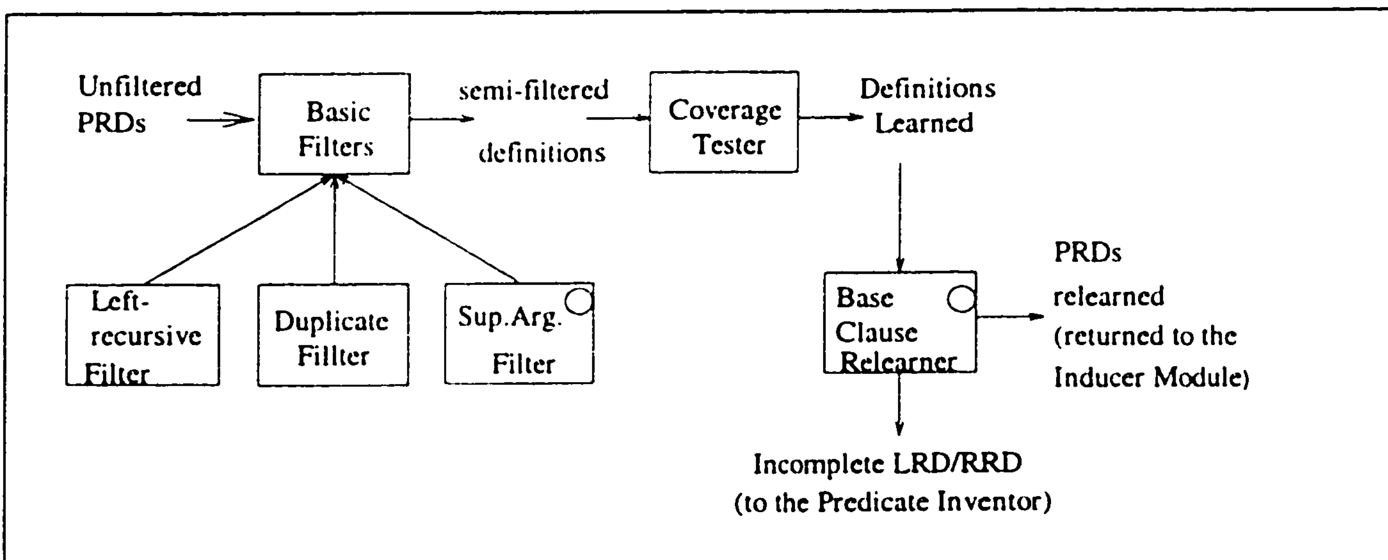


Fig 5.5. The Filter Module.

After a definition passes all tests, it is not directly reported as correct until its base clause is relearned (see Section 5.5). The relearning process concerns only LRDs and RRDs and can be prevented by an option (see Section 5.6.2.6). Individual filters are discussed next.

5.3.2.1 The Left-recursive Filter

The Left-recursive Filter performs a simple, quick check on a definition, searching for infinite recursion. All it does is to compare the input arguments of the head and the body of a PRD for equality. This simple test helps Shrinp get rid of such wrong clauses as:

p(<u>[A|B]</u>,s(C)) :- p(<u>[A|B]</u>,C).

and

q(<u>s(A),B</u>,[s(A)|D]) :- q(<u>s(A),B</u>,D).

for predicates *p* and *q* with modes *[i,o]* and *[i,i,o]*, respectively[12]. The anomalous arguments are underlined. This filter relies on the fact that if *all* input arguments are the same in a recursive call, there will be an infinite loop as the output variables are not yet instantiated.

5.3.2.2 The Duplicate Filter

Two definitions are said to be the same if they are identical after their variables are properly renamed. What this filter does is to cast off repeated definitions such as:

p([A|B],[A|D]) :- p(B,D).

and

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

which are exactly the same after the substitution {A/X,B/Y,D/Z} is applied.

5.3.2.3 Superfluous-argument Filter

A superfluous argument is an argument in the head of a clause that is repeated as is in the same position in the body and nowhere else in the clause. For an argument to be considered superfluous, it must also show as a once-occurring variable in the base clause. A definition with such characteristics can be discarded on the grounds that a similar definition *without* the faulty

[12]The convention adopted in Section 2.1 for describing the modes of a relation uses an "*i*" for an input argument and an "*o*" for an output argument.

argument has also been induced. This assertion is true by construction with respect to LRDs and RRDs (see Chapter 6 on predicate invention). As for PRDs, a useless argument means an infinite recursion that slipped though the Left-recursive Filter, hence the elimination of the definition is mandatory. Here are some examples of clauses with superfluous arguments (indicated by an underline):

p(0,s(A),0).
p(s(A),s(B),C) :- p(A,s(B),C).

q(A,0,A).
q(s(s(A)),0,s(B)) :- q(A,0,B).

r(V,[]).
r(V,A) :- r(V,C), newp(C,A).

5.3.2.4 The Coverage Tester

The coverage tester is the most time-consuming of the filters. That is why it is the last one to be executed. All it does is to run a coverage test on a definition for all given positive and negative examples (see Section 2.3). The Coverage Tester is also called by the Inducer Module as shown in Fig. 5.4.

5.3.3 The Predicate Inventor Module

In some cases, no PRD makes it through the Filter Module. But if some of the PRDs have passed all but the coverage test, they are handed to the Predicate Inventor (see Fig. 5.6). Here they are treated as incomplete definitions that must be completed with a newly invented predicate. It is the task of the Right and Left Inventors to come up with the tentative calls to the new predicate by inserting a new literal to the right or to the left of the recursive call, respectively. Each new definition will originate a search for the examples and the definition of the new predicate through a recursive call to Shrinp as mentioned before.

The examples of the new predicate are computed in two steps. The positive examples are found first. It suffices to run the definition on the given positive instances of the target predicate and record the instantiations of each call to the new predicate (see Section 6.1). This step, carried

out by the Positive Examples Builder, may produce non-ground instances, which hamper the process of finding the base clause discussed in Section 3.5.2.1. For this reason, they are filtered out by the Positive Examples Filter (see Section 6.5.1).

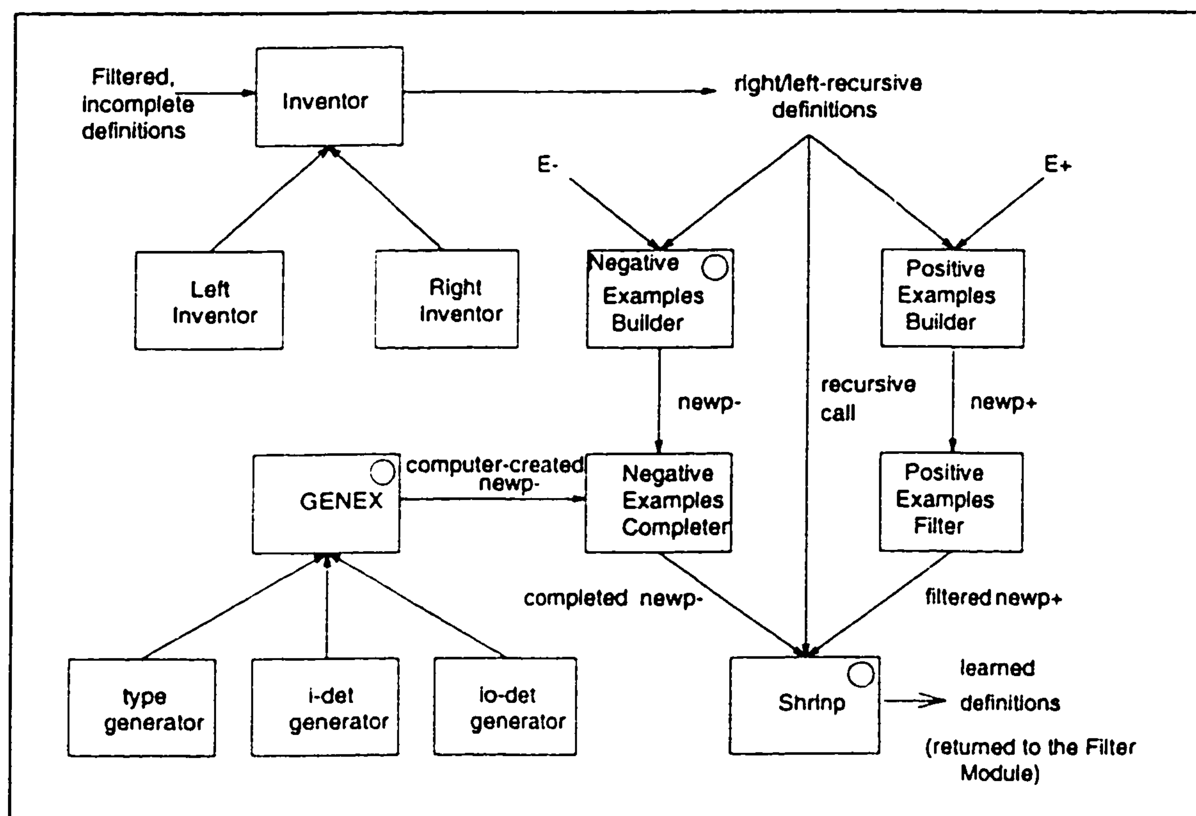


Fig 5.6. The Predicate Inventor Module.

In the second step, the negative examples are computed by the Negative Examples Builder. This task needs more work as the process of finding negative examples for the new predicate described in Section 6.1 may also generate non-ground instances. But this time they are not to be discarded. Non-ground negative examples actually mean an infinite *family* of negative instances: replacing each variable in an example by a different constant yields a different example. What the Negative Example Completer does is to substitute two constants, zero and the empty list, for each variable in the non-ground examples (see Section 6.5 for an explanation of why this is done). The resulting (possibly empty) set of ground negative instances is augmented with computer-generated examples through a call to the GENEX algorithm (explained in Section 5.4.1). Once the examples of *newp* have been gathered, a recursive call to Shrinp is carried out in order to find a definition for the new predicate. The definitions induced for *newp* together with the completed definitions of the target predicate are returned to the Filter Module.

The completion of negative examples and the GENEX algorithm can both derive incorrect examples (see Sections 5.4.2 and 6.5). For this reason, these two modules can be deactivated by options discussed in Section 5.6. Constructive learning in Shrinp is thoroughly covered later in Chapter 6.

5.4 The automatic generation of negative examples

It is known that negative examples work as a means to fine tune the final solution of a learner. As they are only used to filter out incorrect definitions, negatives are harmless to the learning process. Thus, the greater their number, the greater the chance of finding the correct definition faster as the time spent on attempting to specialize incorrect definitions will be saved. But if no examples of this kind are available, it will be impossible for the system to discard all wrong definitions that have passed the filters. No matter how sophisticated the filters may be—take for instance the one used by the system SIERES (Section 4.7)—, they are all syntactic analyzers that cannot take advantage of the actual meaning of a definition. Only negative examples provide a true semantic means to validate a definition.

A small number of positive examples is not enough to define a unique logic program. Should the training set contain BRSes of two or more programs, only the negative examples will be able to tell which one is to be learned. See, for instance, the set of examples below of a predicate *p* (modes *[i,o]*), which can at the same time represent the predicate that returns the length of a list or the sum of the elements of a list (or something else?).

$p([s^1,s^1],s^2)$
$p([0,s^3,0,s^1],s^4)$

The fact that a BRS for the predicate *sum* would certainly contain examples with the relation *plus* does not hold here because Shrinp is restricted to examples of a single predicate only—in this case, *p*. In the absence of other positive instances that might make the distinction, Shrinp can only rely on the set of negative examples. Indeed, the negatives play an important role in this respect. Once the following negative instances are presented to the system, the distinction is made: *p* is in fact the *length* predicate.

$-p([s^2,s^3],s^1)$

-p([0],0)

A popular alternative to providing a learner with negative examples, as shown in Section 4.10, is the use of the closed world assumption (CWA). In this case, those tuples that are not in E^+ are taken as negative examples. This approach works only when one has the complete set of examples known to belong to E^+, an alternative that is not always feasible or is generally expensive.

One problem is that for some real-life applications it may be sometimes hard to obtain positive examples, let alone negative ones. The latter may be either hard to compute or non-existent at all. Several scenarios are possible:

- The target predicate is new to the language so nothing is known about it. This situation occurs, for instance, when Shrinp is called recursively to learn a definition for a new predicate. In this new activation, there are no human-provided examples available of the new target predicate. Where to obtain the negative examples of the new relation?
- There are no negative examples of the target predicate or they are frequently unavailable, a problem that has already been investigated within ILP (see [DeRaedt, Bruynooghe 93], [Quinlan 94], [Mooney & Califf 95], which discuss techniques for dealing with positive-only data sources);
- It is hard to compute or concoct them;
- The computation of an adequate set of negative instances is done through a CWA and, hence, the set is infinite or at least intractably large;

Therefore, it is imperative that a system have its own means to correctly generate negative examples of the target predicate, whenever possible. The way Shrinp approaches the problem is by making use of the GENEX algorithm, explained in the next section. The aim is to use the given positive examples as a basis for the generation of new examples. By carefully corrupting the elements of E^+, the system can come up with adequate negative examples.

The use of computer-generated negative examples does not only fine tune a solution. It also speeds up Shrinp's execution as long as wrong tentative clauses are rejected as early as

possible. Section 5.4.3 explains how negative examples created by the system can be used to discard incorrect base clauses way before they give rise to a complete clause.

To summarize, the availability of negative instances helps a system discard wrong alternatives as early as possible, thereby increasing the accuracy of the learned program. The more negative examples are present, the lower the number of possibilities that have to be considered by a learner, so the faster the execution. Now that the importance of the negative examples has been made clear, the GENEX algorithm can be introduced.

5.4.1 The GENEX algorithm

The GENEX algorithm produces a set of computer-generated negative examples for a given predicate. It works by corrupting the arguments of the positive examples according to a user-selected operating mode. The mode can be either *type*, *output determinacy* or *input-output determinacy*. A combination of the first mode and one of the other two modes can be set at the same time. The algorithm is depicted later in Fig. 5.7. The modes are explained next.

This section also discusses problems with the use of this algorithm. Options are provided that inhibit GENEX's use or at least set the *io-det* mode off. See Section 5.6 for details on these and other options. Also refer to the experiments in Chapter 7 on the results of using or not using the algorithm.

5.4.1.1 The type mode

When in *type* mode, GENEX replaces the arguments of a positive example by an object of a different type. This does not mean, however, that the language L has to be extended with the definition of types for the arguments. It simply means that GENEX substitutes a new constant $c \in$ L for a given argument. For instance, the positive example of a predicate that returns the nth element of a list, *extractNth(*s^1*,[a,b],a)*, will successively give rise to the following new negative examples:

-extractNth(-c1,[a,b],a).
-extractNth(s^1*,-c2,a).*
-extractNth(s^1*,[a,b],-c3).*
-extractNth(-c1,-c1,-c1).

where *-c1*, *-c2* and *-c3* are constants that have not been used in the problem yet (guaranteed by the use of the dash, as all constants, variables and functions allowed by the language defined in Section 2.1 start with a letter or a digit). The last example, for instance, is necessary for a base clause like *extractNth(X,X,X)* to be discarded. The use of this mode prevents such incorrect definitions that have a variable that occurs only once in the base clause from being learned. The *type* mode is the most restrictive of the three types and, thus, the least biased.

5.4.1.2 The *input determinacy* mode

The *i-det* mode is based on the assumption that the target predicate is determinate. This means that if any of the output arguments of a positive example are changed, the resulting literal is no longer positive. Said differently, the *i-det* mode assumes the target predicate to be a functional relation with possibly multiple outputs. Because no information is given about the domain of the arguments as is the case of the system PAL ([Morales 92 and 97]), a new value has to be chosen by the system. The outcome of substituting the outputs, by such new constants as *0* and *[]* (see Fig. 5.7), is a powerful set of negative examples. The following are the literals created by GENEX for $doubles([s^1],[s^2])$, a predicate that returns a list with the double of the elements of the input list.

$-doubles([s^1],[])$.
$-doubles([s^1],0)$.

Among the incorrect definitions that these negative examples help discard are *doubles([X|Y],Y)* and *doubles(X,[])*, which is not guaranteed by the *type* mode alone.

5.4.1.3 The *input-output determinacy* mode

The *io-det* mode is the least restrictive of GENEX's operating modes. It is strong but potentially unreliable. The unreliability comes from the fact that this mode assumes that a literal is determined once the input *or* the output arguments are instantiated. This allows for the replacement of the input arguments as well. The relation is, thus, expected to be an *injective function*.

To illustrate, the positive example of a predicate that returns the sum of the two first arguments, *plus*(s^1,s^2,s^3), spawns the six new literals below, which are very close to human-generated examples:

-*plus*(0,s^2,s^3).
-*plus*([],s^2,s^3).
-*plus*(s^1,0,s^3).
-*plus*(s^1,[],s^3).
-*plus*(s^1,s^2,0).
-*plus*(s^1,s^2,[]).

The algorithm is shown in Fig. 5.7. The question whether the algorithm ends is straightforward as the main loop and the inner loops depend, respectively, on the size of the set of positive examples and the number of arguments of the predicate that the system is trying to learn. As there are finitely many positives and the arity of the target predicate is a constant, the algorithm will eventually stop.

5.4.2 When not to use GENEX

Shrinp benefits from using the GENEX algorithm in several ways. In the absence of negatives from both the user and the algorithm, it would be virtually impossible to learn such predicates as *plus*, *twice* and *append* alone, which are all relations invented by the system in order to find a definition for *sum*, *doubles* and *reverse*, respectively (see Section 7.2.2). There are situations, though, when GENEX's use may be harmful. One involves the use of the *io-det* mode and the other the use of the algorithm altogether.

5.4.2.1 Problems with the io-det *mode*

The use of this mode can be harmful, especially if the target predicate is not injective. It can lead to such wrong negative example of the *member* predicate as *member*(0,[s^1,0]), which is obtained from *member*(s^1,[s^1,0]) when the first argument is replaced by the constant zero. So there is a risk of GENEX creating positive examples that are wrongly taken as negatives.

```
algorithm GENEX
Generate negative examples from user-given positive examples
In: Pos, a set of positive examples
    Mod, a set of modes (type, input or inputOutput)
Out: Neg, a set of negative examples
Functions:
   simpleExample: return a literal with same arguments
   allArgumentsOf: return all arguments of an example
   outArgumentsOf: same for the output arguments
   newConstant: return a new constant
   replace: substitute a new constant for an argument

begin
 Neg = { };
 if type ∈ Mod then Neg = simpleExample(p);
 foreach p ∈ Pos do
   if type ∈ Mod then
      foreach a ∈ allArgumentsOf(p) do
         Neg = Neg ∪ replace(p,a,newConstant);
   if input ∈ Mod then
      foreach a ∈ outArgumentsOf(p) do
         Neg = Neg ∪ replace(p,a,0) ∪ replace(p,a,[]);
 if inputOutput ∈ Mod then
    foreach a ∈ allArgumentsOf(p) do
       Neg = Neg ∪ replace(p,a,0) ∪ replace(p,a,[]);
 endfor
end;
```

Fig 5.7. GENEX algorithm.

Other mistaken examples:

- *substring([],[s^1,s^2,0])* created from *substring([s^2],[s^1,s^2,0])* and
- *noneIsZero([])* created from *noneIsZero([s^1,s^2])*.

To try to avoid this situation, Shrinp always checks if a new negative example is already a member of the set of positive examples. If so, it is discarded. Although the idea is not always feasible because not all positives are available, at least the consistency of the training set may sometimes be guaranteed. Another alternative is to employ the more reliable *type* and *i-det* modes together instead of the *io-det* mode. Otherwise, it constitutes the most powerful tool in the

automatic generation of negative instances by GENEX.

The *io-det* mode can be turned off—see Section 5.6.2.4.

5.4.2.2 Problems with the algorithm itself

The importance of system-generated negative examples has already been shown (see also experiments in Chapter 7). However, GENEX may make assumptions that are wrong. Thus, there exist predicates that can only be learned when the algorithm is inactive. Consider, for example, the examples below of a predicate that checks if a list of natural numbers is in non-decreasing order, with mode *[i]*:

Given positive examples:

isSorted([0,s^1,s^3]).
isSorted([0]).
isSorted([s^1,s^2,s^3]).
isSorted([s^1,s^3]).
isSorted([s^2,s^3]).

and negative examples:

-*isSorted*([]).
-*isSorted*([s^1,0]).
-*isSorted*([s^2,0]).
-*isSorted*([s^2,0,s^3]).
-*isSorted*([s^3,s^2,s^1]).

The definition that is being looked for is the following, where *lessEq* is a relation that succeeds if and only if the first argument is less than or equal to the second argument:

```
isSorted([A]).
isSorted([A,s(B)|C]) :-
      lessEq(A,B),
      isSorted([s(B)|C]).

lessEq(0,A).
lessEq(s(A),s(B)) :-
     lessEq(A,B).
```

What happens is that the *type* mode implies the creation of the following incorrect negative example for the new predicate: *lessEq(0,-c13)*. This example comes from the (correct) positive example *lessEq(0,0)*, which is found when the head of *isSorted*'s recursive clause is instantiated with the first positive example, *isSorted([0,s^1,s^3])*. When the *type* mode is employed, *lessEq(0,0)* is corrupted to *lessEq(0,-c13)*, which matches with the base clause of the correct definition of *lessEq*, thereby invalidating such a definition. This example shows the vulnerability of the *type* mode when the base clause contains variables. One solution to this situation is to use the GENEX algorithm as a default option and to give the user a chance to turn it off (see Section 5.6).

5.4.3 Using base clauses as negative examples

The process of computing the base clause from a small number of instances was presented in Section 3.5.2.1. Recall that Shrinp first matches the generating terms of the examples, producing groups of subterms with the same GT. Let these groups be G_1, G_2, ..., G_n. A potential base clause is induced from the subterms of each group by computing the lgg of their corresponding arguments. There will be, as a result, n candidates: bc_1, bc_2, ..., bc_n, of which only one is *the* correct base clause (Fig. 5.8), according to the language bias. The problem is that some of them are overly general or overly specific. BCNEG is the algorithm that helps the system get rid of the wrong ones.

The way the algorithm works is by running a negative coverage test. Inconsistent candidates are removed if they cover any negative instance.

When a recursive activation of Shrinp is called from the Predicate Inventor, there is a chance that there will be not many negative instances available (see discussion in Section 6.5). Recall from Section 3.5.2.1 that, for a candidate base clause to be considered correct, it must be consistent with the set of negatives. Considering that the language bias imposes the existence of one single base clause, a potential base clause bc_i can be tested against the members of the groups other than G_i. If the subterms in G_i are the true base members that will produce the correct clause, no other element of the groups G_1, G_2, ..., G_{i-1}, G_{i+1}, G_{i+2}, ..., G_n, excluding those that also belong to G_i, can be considered correct as they represent different operations at different depths

to get to the base clause from the examples. This fact suggests a way to get rid of incorrect base clauses.

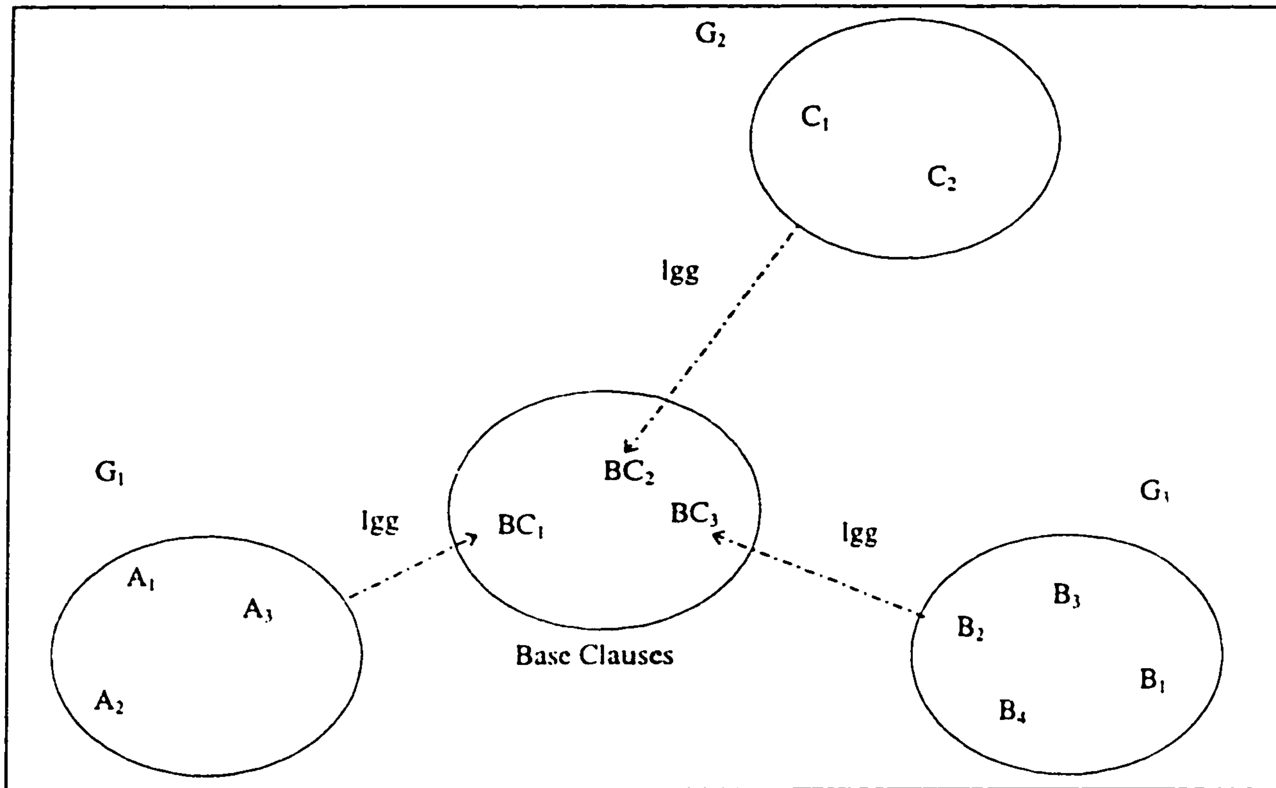


Fig 5.8. The BCs are obtained by lgging the subterms of each group.

To know if a base clause bc_i is false, just check if it covers any of the negative examples given by the expression:

$$\{\forall x \mid x \in E^{-} \text{ or } (x \in G_j \text{ and } x \notin G_i,\ 1 \leq j \leq n,\ i \neq j)\}.$$

If it does, discard it.

The method is an easy, yet time-consuming, way of increasing the number of negative instances. In general, the time saved on learning a definition from an incorrect base clause pays for its use. It has to be said, though, that the method presented here may also wrongly eliminate correct base clauses—see discussion in Section 5.4.3.2.

5.4.3.1 An example: learning a base clause for minus

The predicate *minus* (modes *[i,i,o]*), that computes the subtraction of the first from the second arguments, will be used as an example. A subtraction on natural numbers is assumed: 2 minus 5

will give zero as an answer while 5 minus 2 equals 3. Consider the training instances below. The examples on the left are true while those on the right are false. Note that E-2 is a negative example since s^1 minus s^2 should be zero.

(E+1) $minus(s^1,s^1,0)$ (E-1) $-minus(s^3,0,s^2)$

(E+2) $minus(s^2,s^3,s^1)$ (E-2) $-minus(s^2,s^1,s^1)$

Consider the subterms and generating terms of the two positive instances above. Table 5.3 presents the eight groups of matched subterms extracted from E+1 and E+2, given the restrictions discussed in Section 3.5.2.1. The first subterm comes from E+1 while the second subterm comes from E+2. The *lgg under implication* of each group is also shown as the potential base clauses.

When Shrinp uses only the above negative examples, E-1 and E-2, for discarding the incorrect base clause, a total of six potential base clauses are cast off: bg_1, bg_2, bg_3, bg_4, bg_5 and bg_8. The two remaining base clauses will, therefore, generate two different recursive clauses. Of these, only one definition is correct, that one with bg_7 as the base clause as any number minus zero equals itself[13]. But because bg_6 is still in the game a considerable amount of time will be spent before the incorrect definition it gives rise to is rejected, probably after the invention of several predicates. The sooner a clause is recognized as being wrong, the more efficient the learning process will be. The negative examples alone sometimes are not enough.

Now consider what happens when the subterms of a group are used as negative examples for the base clauses derived from the other groups. Before checking the contenders against the user-given negative examples, Shrinp verifies that clauses bg_2, bg_3 and bg_4 cover members of group G_1 and are automatically discarded. Clause bg_6 covers one subterm of G_2 and is also ignored. Clauses bg_1, bg_5 and bg_8 do not cover elements of other groups but are discarded as they cover the given negative examples.

The only base clause left, bg_7, does not cover any of the members of the other groups that are different from subterms in G_7, nor does it unify with the given examples. Being consistent

[13]Note that the base clause *minus(0,A,A)* would never be discarded by any of GENEX's modes.

with all examples, bg_7 is correctly taken as *the* target base clause.

Table 5.3. Subterms and potential base clauses for the predicate *minus*.

Groups	Subterms	Potential base clauses
G_1	$minus(s^1,s^1,0)$ $minus(s^2,s^1,s^1)$	$bg_1 = minus(s(A),s^1,A)$
G_2	$minus(s^1,s^1,0)$ $minus(0,s^3,s^1)$	$bg_2 = minus(A,s(B),C)$
G_3	$minus(s^1,s^1,0)$ $minus(0,s^1,s^1)$	$bg_3 = minus(A,s^1,B)$
G_4	$minus(s^1,s^1,0)$ $minus(s^2,0,s^1)$	$bg_4 = minus(s(A),B,A)$
G_5	$minus(s^1,0,0)$ $minus(s^2,s^1,s^1)$	$bg_5 = minus(s(A),A,A)$
G_6	$minus(0,s^1,0)$ $minus(0,s^3,s^1)$	$bg_6 = minus(0,s(A),B)$
G_7	$minus(0,0,0)$ $minus(0,s^1,s^1)$	$bg_7 = minus(0,A,A)$
G_8	$minus(s^1,0,0)$ $minus(s^2,0,s^1)$	$bg_8 = minus(s(A),0,A)$

As a consequence, only one recursive clause is generated and the final definition is found:

```
minus(0,A,A).                                   (D1)
minus(s(A),s(B),C) :-
    minus(A,B,C).
```

This way, considerable time was spent on inducing definitions for other base clauses and possibly searching for new predicates to specialize them.

5.4.3.2 Pitfalls

Table 5.3 showed how to use the subterms that generate other candidates to being base clause in

order to discard an incorrect candidate. They work perfectly as system-generated negative examples in this specific situation, so why not extend their use to help discard incorrect definitions as a whole? The problem is that not all of the subterms in incorrect groups are true negative examples. Take, for instance, the subterm *minus*(s^1,s^1,0) from G_1. If definition D1 above were to be run against that subterm, it would be incorrectly rejected because *minus*(s^1,s^1,0) is actually a positive example being wrongly used as a negative instance! Given the uniqueness of the target base clause, the groups of subterms guarantee only that false base clauses may be discarded. Nothing can be said about their use with respect to complete definitions. Even so, there is nonetheless no absolute guarantee that good base clauses will not be discarded either.

Consider, for example, the predicate *prefix* (modes *[i,i]*). It succeeds if the first list is a prefix of the second one. In the process of finding its definition from the following set of examples (positives on the left; negatives on the right):

prefix([a],[a,b,c])	*-prefix([a],[b,c])*
prefix([e,b],[e,b,c])	*-prefix([e,b,c],[e,b])*
prefix([b],[b])	*-prefix([a,c],[a,b,c])*

Shrinp generates 32 potential base clauses. When BCNEG is disabled, only four of these candidates pass the coverage test, giving rise to four different definitions. The Filter Module later reduces them to just one, the correct definition:

```
prefix([],A).                                   (D2)
prefix([A|B],[A|C]) :-
    prefix(B,C).
```

When Shrinp is run with the subterms being used as negative examples, only two of the 32 base clauses pass the coverage test, but this time D2 is not induced. Why? Because the correct base clause bg_{24} in Table 5.4 is summarily dropped in the process as it covers the second subterm of the 16th group. Note that that subterm does not belong to the group where the correct base clause comes from.

The subterm of G_1 in Table 5.3 that is in fact a positive example fortunately did not prevent the target definition from being found as it does not unify with the correct base clause. But there exists anyway a potential for error. The use of subterms from other base clauses to

eliminate wrong candidates can definitely be harmful. An option is provided that inhibits BCNEG's use (see Section 5.6).

Table 5.4. Some subterms and potential base clauses for the predicate *prefix*.

Groups	Subterms	Potential base clauses
G_{16}	*prefix([a],[a,b,c])*	bg_{16} = *prefix(A,[B|C])*
	prefix([],[e,b,c])	
	prefix([b],[b])	
G_{24}	*prefix([],[b,c])*	bg_{24} = *prefix([],A)*
	prefix([],[c])	
	prefix([],[])	

Although this example suggests the need for user-supplied negative examples, Shrinp can still successfully learn the base clause in many cases even when only system-generated negative examples are used. See experiments in Chapter 7.

5.5 Relearning the Base Clause

We have seen in Section 3.5.2.1 that the recursive call is generated by applying the generating term (GT) of each argument only once to the arguments in the head of the clause. The GTs originate from the operations applied to the positive examples that yielded the "winning" base clause. So, there is a great correlation between the base clause and the recursive call of a purely recursive definition. Both share the same "winning" GT. However, when a left-recursive definition is learned, the following situation arises: the induced output values of the base clause may no longer be correct as the addition of a new predicate to the right of the recursive call may intervene in the result of the recursive clause by possibly changing the value returned by the clause before the introduction of the new literal. For RRDs, a similar situation is possible: the input arguments of the base clause may need fixing because of the addition of a new literal to the left of the recursive call. In such cases, the base clause has to be relearned in order to reflect the new circumstances.

Shrinp relearns the base clause in a pretty efficient manner. To illustrate the process,

consider the next LRD, where all predicates have modes *[i,o]*.

p(a,b). (BC)
p([A|B],Y) :- p(B,C), newPred1(C,Y). (RC)

The addition of *newPred1* to RC intervenes in the result of that clause, as the recursive call to *p* returns a value, *C*, that is transformed by *newPred1* before it becomes *p*'s output, *Y*. One efficient way to check if BC is correct after this addition is by running RC on the positive examples of *p*. But before that, Shrinp temporarily shifts RC by making the recursive call the last literal[14]

p([A|B],Y) :- newPred1(C,Y), p(B,C). (RC')

Picking one example *e* at a time, the system executes RC'. With *Y* now instantiated, *C*'s value can be known merely by executing *newPred1*. The ground instance found in the recursive call, *p(value of B, value of C)*, can now be used to trace RC one more time, yielding a new ground instance *p'*. This process is repeated until the input argument of the ground instance p^n found matches the input argument of BC, *a*. At this point, the last value found for *C* must agree with *b*. If it does, it is safe to assume that the base clause is correct. Otherwise, a new, correct base clause is derived by substituting the last value found for *C* for the wrong output argument *b*. In other words, to relearn a base clause, it suffices to execute the pair BC/RC on every positive example. If the base clauses found at the end are the same as BC, do not change the base clause. Should all base clauses found be equal but different from BC, modify BC.

But what happens if the base clauses found are different from one another? As there can be only one base clause, BC is changed only if the values obtained from each positive example are identical. In case they are not, BC is in fact wrong. So the definition that contains BC can be safely rejected. The process, therefore, makes it easy to discard wrong tentative definitions as they give rise to different base clauses. The language bias discussed in Chapter 6 guarantees that all the variables are instantiated when the recursive call is reached.

To relearn an RRD, a similar process is involved, but this time no shifting needs to take

[14]Even though RC' can be considered a legitimate LRD, it could never be treated as correct given that it disagrees with the modes defined for *p* and *newPred1*. Only the variables *A* and *B* are supposed to be known by the time *newPred1* is to be evaluated.

place as the variables in the new predicate do not depend on the recursive call. If at the end of the above process the values found for the input arguments disagree with the input arguments of the base clause, the base clause is changed. Again, wrong base clauses are discarded if the values found are not identical.

The first system to relearn the base clause was CLAM (Section 5.1), but it is limited to inducing LRDs. Shrinp extends CLAM's relearning process by allowing RRDs to be fixed as well. The correctness of the base clause of a definition D can only be verified after the definition of the new predicate that is invoked by D has been induced.

5.5.1 An example: learning a definition for the relation *Permut*

To illustrate, consider the predicate *permut* with modes *[i,o]*. This predicate returns a permutation of the elements of a list. Shrinp is given the following positive examples:

permut([1,w],[w,1]). *permut([p,b,d],[b,d,p]).* *permut([2],[2]).*
permut([1,2],[1,2]). *permut([w],[w]).* *permut([b,d],[b,d]).*

and the negative examples below:

-permut([],[e,g]). *-permut([z],[a]).*

The definition D3 induced from these examples as wells as the definition of the invented predicate *newPred14* are shown in Table 5.5. Note how the induced base clause, *permut([],A)*, erroneously tells that anything could be the permutation of the null list. To check that base clause, Shrinp runs each positive example of *permut* on D3. It is easy to see that the output argument found from each run is consistently the same: *[]*. That is an indication that the base clause has to be fixed. To correct it, it is enough to substitute *[]* for the variable *A*. After the correction is made, Shrinp finally comes up with the target definition D4 for the relation *permut*. The relearning of the base clause is, therefore, necessary for this predicate to be learned by Shrinp.

5.6 Parameters and Options

For reasons of efficiency and correctness, various options and parameters are provided by the

system Shrinp. They are indicated in the next two sections.

Table 5.5. Definition induced for *permut* before and after its base clause is relearned.

Induced Definition D3	Invented Predicate	Corrected Definition D4
permut([],A).	*newPred14(A,B,[A|B]).*	*permut([],[]).*
permut([A|B],C) :-	*newPred14(A,[B|C],[B,D|E]) :-*	*permut([A|B],C) :-*
permut(B,D),	*newPred14(A,C,[D|E]).*	*permut(B,D),*
newPred14(A,D,C).		*newPred14(A,D,C).*

5.6.1 Parameters

Parameters are numeric values that guide the execution of Shrinp. The judicious choice of these values may be the difference between a fast and a slow session. Two parameters are defined: maximum depth and number of positive examples.

5.6.1.1 Maximum depth (maxdep)

To learn a definition of a relation may involve the learning of the definition of a new predicate, which by its turn may need the attempt to learn another new predicate and so on and on (see Fig 5.3). Each attempt to learn a definition of a new necessary predicate is a recursive call to the system. Some predicates can be learned in just one call to Shrinp such as *append* and *member*. These are depth-one relations. Depth-two relations, such as *sum* and *subset*[15], need a definition of a new purely recursive predicate. Depth-three relations need a depth-two relation and so on. In the search of the definition of a relation, Shrinp may generate deep incorrect relations until the correct definition is found, an extremely time-consuming process. If the user believes that the target predicate can be defined by an n-depth relation, he or she might prefer to set this parameter to n. Hence, for $maxdep = n$, m-depth relations with $m < n$ are learnable.

5.6.1.2 The number of positive examples (npos)

It is possible that a Basic Representative Set of a relation be present in just a portion of the

[15] See Chapter 7 for the definition of these predicates.

positive examples, especially when there is a large number of them. The more the number of positive examples, the greater the number of subterms and, as a consequence, of generating terms. More generating terms mean that more base clauses will be generated and tested against the negative examples, more computer-generated negative examples will be created and, of course, more housekeeping (garbage collection etc.) will be necessary. So, the fewer the positive examples there are, the more efficient the learning will be. Recall that a BRS defines one and only one relation. So there is no point in dealing with the other examples. Shrinp picks a subset of the positive examples at random to work with, in hopes that a BRS will be contained inside the chosen subset. The size of the subset is given by the parameter *npos*. If zero, it means that the whole set of examples is to be considered.

5.6.2 Options

The options tell Shrinp how the execution must be carried out. They indicate whether several functions have to be performed or not. The correct choice of the option may be the difference between a successful and an unsuccessful learning, as shown in Chapter 7. Several options are defined:

5.6.2.1 BADARG

If this option is enabled (the default), Shrinp checks clauses for useless arguments. That means that every time a definition is found, Shrinp does a quick look at each argument to see if it repeats itself in the body and the base clause without being changed. If so, the clause is rejected on the grounds that a clause similar to that, except that it does not have that argument, has also been learned. See Section 5.3.2.3 for details.

5.6.2.2 BCNEG

This option, initially on, tells Shrinp to use the wrong BCs as negative examples while finding the correct base clause. It may take longer for the base clause to be induced, but the number of potential definitions will be as low as possible—see Section 5.4.3.

5.6.2.3 GENEX

This option activates/deactivates the GENEX algorithm. As seen earlier in Section 5.4.2, it may sometimes be wise to turn some of GENEX's operating modes off. Instead of having one option for each of GENEX's modes, which could be confusing, we opted for having only two: one to simply turn GENEX on or off, the default being on, and another option to choose from the *i-det* or the *io-det* modes, GMODE. See next.

5.6.2.4 GMODE

GMODE makes GENEX mode be either input or input-output determination. For some predicates an *io-det* mode can be harmful (see Section 5.4.2.1), although other predicates can only be learned when this mode is on. The default mode is the more reliable *i-det*.

5.6.2.5 NEGUSER

If active, this option tells Shrinp to use the given target predicate's negative examples to learn the examples of a new predicate. Although effective in some cases, this alternative is error-prone as explained in Section 6.5.2, so it is initially set to off.

5.6.2.6 RELEARNBC

This option tells Shrinp to relearn the base clause, if need be, after a definition is found, according to the discussion in Section 5.5. Due to the extra time it takes, this option is off by default.

5.6.2.7 SHRINP

This option toggles the Shrinp algorithm on/off. If inactive, the system will run very fast but only depth-one (purely recursive) definitions will be learned. That is why this option is initially turned on, so both purely and right/left-recursive definitions can be learned.

5.6.3 Defaults

Below are the default values for the parameters. Table 5.6 also indicate what value contributes to

a more efficient execution. The reader must read this column cautiously as the gain in efficiency for not performing a task may, on the other hand, imply more work later.

Table 5.6. Default values for the parameters.

Parameters	Default	More efficient if...
maxdep	2	a depth of two is chosen. However "shallow", it allows for several important predicates to be learned.
npos	0	set to two. This is the minimum number of positive examples that can be picked at a time. Since it is difficult for the BRS to be in such a small set of examples, chances are that the results will not be very promising.

The default value for the options are shown in Table 5.7. As before, we preferred to sacrifice efficiency for correctness. This means that the default value does not always imply an efficient execution as can be seen in the third column of that table. For instance, Shrinp may run faster if the BCNEG option is turned off, but when this option is enabled, Shrinp will be able to eliminate incorrect base clauses as they are generated, which contributes to a better performance of the system.

The default values can be easily changed by the user after Shrinp is loaded—see example in Appendix B.

5.7 Complexity Analysis

This section analyzes the computational complexity of our technique. We want to find an expression that tells the time taken for Shrinp to learn a definition based on the size of the training set and other parameters. A close look at Shrinp reveals that our technique can be divided into two main parts: the search for the base clause and the search for the recursive clause. Each part will now be treated separately.

Table 5.7. Default values for the options.

Options	Default	More efficient if...
BADARG	on	left off. If so, clauses with useless arguments may be learned. A high potential for infinite recursion is possible.
BCNEG	on	disabled. As a tradeoff, the number of potential base clauses will be higher and more incorrect clauses will be generated.
GENEX	on	disabled. The tradeoff is that by turning this option off, one runs the risk of having more incorrect recursive clauses pass the coverage test.
GMODE	i-det	the i-det mode is used for the sole reason that the other mode, io-det, encompasses this one.
NEGUSER	off	left off. The use of the target predicate's given negative examples to learn the examples of a new predicate not only is error-prone. It also is time-consuming. Some invented predicates are learned faster when this option is on, though.
RELEARNBC	off	disabled. Most of the time the base clause needs not be relearned. Relearning a base clause takes time and is done after each definition is found.
SHRINP	on	disabled. When Shrinp is not running, only depth-one definitions are learned. No new predicates are attempted and, therefore, no recursive call to Shrinp is carried out.

We start by defining the parameters. Let:

- p be the number of positive examples, $p > 0$;
- n be the number of negative examples, $n \geq 0$;
- t be the cost of running one test against one example;
- a be the arity of the target predicate, $a > 0$;
- c be the maximum number of arguments of a function (e.g., *pair* has two arguments), $c > 0$;

- d be the maximum argument depth of any example (e.g., $d = 3$ for the positive example $p(s(s(s(0))))$), $d \geq 0$;
- t be the cost of running a test against one example; and
- m be the maximum depth of a definition (see Section 5.6.1.1), $m > 0$.

5.7.1 The complexity of the search for the base clause

As illustrated in Fig. 5.4, the search for the base clause involves finding several candidate clauses from the positive examples. To pass this phase, a candidate must pass a negative coverage test. The algorithm for finding the potential base clauses starts by finding the subterms of the arguments of the positive examples (see Section 3.5.2.1). There are at most cd subterms for each argument. Hence, there will be $(cd)^a$ generating terms for each example. The cross-product of the valid matches found for each positive example results in $p(cd)^{2a}$ possible base clauses. Of course this number is much smaller due to the constraints shown in Section 3.5.2.1. These constraints, among other things, discard those combinations with different generating terms and different depths. In our empirical tests, the $p(cd)^{2a}$ base clauses are reduced by a large factor that varied from four to as much as eight after such constraints are applied.

In the next step, the potential base clauses are tested for coverage of the n negative examples. It follows that the search for the base clause is bounded by the expression $O(npt\,(cd)^{2a})$. As a consequence, the work done by the Base Clause Inducer is exponential in a. Note that if the BCNEG option is enabled (see Section 5.6.2.2), the number of potential base clauses is even smaller.

5.7.2 The complexity of the search for the recursive clause

The search for the winning recursive clause is directly affected by the previous process. Each base clause that passes all constraints and the coverage test gives rise to a different recursive clause, thereby forming pairs of definitions. For a purely recursive definition to be learned, a coverage test is run for each definition as is. Hence, the complexity of learning a PRD is proportional to $(n+p)npt^2\,(cd)^{2a}$.

Learning RRDs and LRDs is a more complex task because they require the invention of a new predicate *newp*. The call to *newp* in the body of the recursive clause can be written in at most $(2acd)^{2a}$ ways[16]. Hence, the number of different recursive clauses comes to at most $npt^3 (n+p)^2 (2acd)^{2a} (cd)^{2a}$ as a coverage test is also required for each new definition. Finally, Shrinp has to find a definition for *newp*. The examples of *newp* are determined by running each definition on the given examples (see Section 6.5). The search for the definition of the new predicate is actually a recursive call to Shrinp that can, in its turn, result in the generation of that many definitions of *newp*. Because *newp* may require another new predicate, which may require another one and so on, Shrinp will be recursively called at most as many times as the value of m. It follows that the time taken by Shrinp to learn a left- or a right-recursive definition of maximum depth m is bounded by $O([npt^3 (n+p)^2 (2acd)^{2a} (cd)^{2a}]^m)$. So, after simplification, we come to the time taken by Shrinp to learn a definition:

$$O\left(\left[(npt)^3 (2a)^{2a} (cd)^{4a}\right]^m\right)$$

However complex the above expression may look, it has to be noted that in practice Shrinp's execution time is low due to a number of reasons. For instance, the value of m is low because the usual recursive definitions are relatively shallow, that is, they are usually two or three levels deep. And besides that, the *maxdep* parameter is set by default to two (see Section 5.6.1.1), so typical executions of Shrinp will not generate too deep definitions. Also, the number of arguments of functions that manipulate lists and numbers is usually small and, in addition, many incorrect definitions may be discarded early in the process. That explains why the runtimes reported in Chapter 7 are mostly small.

5.8 Discussion

This chapter presented the main characteristics of the system Shrinp. The discussion focused on the modules that comprise the system, the automatic generation of examples, the relearning of the base clause and several other details on the implementation. An analysis of the complexity of the algorithm and another extension of Crustacean, CLAM, were also shown.

---

[16] We are assuming that the maximum arity of the new predicate is twice as much as the arity of the target predicate.

Already touched upon earlier, what mainly distinguishes Shrinp from its predecessors, LOPSTER, Crustacean and CLAM, is the invention of new predicates. The use of a background set of supporting definitions not only implies that the user somehow knows the target definition but it also restricts the language considerably to the combination of those relations. If the background knowledge is small, few definitions are learned; if it is large, an exponential explosion may occur. Constructive learning extends a language with new terms and a hypothesis space with new definitions.

In spite of depending on background knowledge, CLAM has a slightly less restrictive language bias than Shrinp. CLAM allows recursive clauses with more than two literals in the body as long as the recursive call is the first one. The only limit is the number of the relations in the background knowledge, which can be used only once in a clause.

The only similarity between Shrinp and CLAM, besides being extensions of Crustacean that specialize PRDs that are not consistent with the training set, is the process of relearning the base clause. But while CLAM applies this process to correct LRDs only, Shrinp goes a little further by relearning the base clause of both LRDs and RRDs.

The ability to learn in the absence of negative evidence should be a characteristic of robust learners. Although the approach utilized here is conceptually quite simple, it is quite powerful given the premise that no help from the user is possible. Experiments in Chapter 7, where learning is successful with positive examples only (plus GENEX's own) show how effective the GENEX algorithm can be. Current learning systems could also benefit if endowed with such mechanisms in order to improve their performance. Indeed, the result of a learning process can be enhanced when both user- and computer-generated negative examples are applied. Usually, the greater the number of negative examples, the better the solution at the expense of a slight increase in runtime.

[Yardeni, Shapiro 87 and 91] propose a *type specification* for logic programs. The approach requires the use of a regular unary language to describe the types of each data structure used. While the description of types suggests a somewhat easier way to syntactically tell negative from positive examples, it makes using the system more complex in that the user is required to provide much more than just the modes and examples of the target relation. A complete type

system for natural numbers, atoms, lists of numbers, lists of lists etc. is necessary. This is true even if the system already had built-in type declarations, as the user would be required to spend time trying to understand them and they might not necessarily match his or her needs. A type system conflicts with the principle of learning from the data only and was not pursued. Moreover, algorithm discovery would be troublesome if the students were to provide types each time an attempt to learn a definition is made. The method we designed is simple, efficient, easy to implement and, perhaps more importantly, transparent to the user.

Another artificial intelligence paradigm that has received considerable attention lately—called Genetic Programming ([Koza 92 and 94])—also aims at program induction. Just like ILP, GP searches a space of possible computer programs that solve a given problem, i.e. a program that produces the desired output for a particular input. But there are many more similarities (and differences) between the two approaches.

GP works by first generating a set of random programs, called the initial population, that are evaluated for fitness to the problem at hand. The programs that best fit the problem are then modified in a way that resembles nature's evolution. This way, a new population of computer programs is "genetically" bred that is also tested for fitness. New generations are continually created until a program that solves (or approximately solves) the problem is found. Evolution is an on-going process that has no well-defined terminal point. Therefore, one criterion for termination of the learning session may either be the time, i.e. the breeding of a certain number of generations, or the induction of a (probably inexact) solution.

One similarity between the two paradigms lies in the background information that is given to the system. Many ILP systems make use of background knowledge to induce a program that covers the training set. This background set is composed of several relations and literals that can be used in the definitions of the induced program. Likewise, GP uses a set of primitive functions, which range from arithmetic functions to boolean and conditional operations, that can also be used in the learned programs. The first generation is mainly composed of a random selection of these primitive functions. In both paradigms, this supporting set is composed of various functions/predicates that are peculiar to the given problem domain. In this respect, Shrinp

differs from GP in that it requires no background knowledge.

Both approaches can learn constructively. Several systems were described in this work that are capable of predicate invention. The GP counterpart to predicate invention is the dynamic discovery of subroutines, or functions that solve a partial problem. Shrinp, along with other few ILP systems, invents relations that are necessary to the learning process. Shrinp stops the invention of new predicates when a certain depth is reached, while GP stops creating new offsprings when a certain number of generations has passed. Finally, in both GP and ILP one is not concerned at first with the size and shape of the solution.

We can also identify some differences between the two paradigms:

- The search mechanism in both approaches is totally different. Not only are the operations that change one representation to another different, but the test if a final solution was induced is also different.
- ILP systems stop when a program is learned that is consistent with the training set. GP systems stop when a solution that highly fits the problem is found. This means that approximate solutions are also accepted, especially in optimization problems, as long as a criterion for fitness measure is achieved.
- GP has been applied to several fields including protein classification, game playing, pattern recognition etc. Although ILP systems have also been applied to a variety of fields (see Section 1.1), Shrinp is more limited as it was designed to be used as part of a general-purpose learner and is specialized in learning simple recursive definitions.
- GP applications usually consume a large amount of data. Take, for instance, weather and biological sequence data ([Koza 96a]). Shrinp, on the other hand, assumes that the available data is both sparse and small.
- A final, minor difference regards the target language. Shrinp relies on logic as the learned definitions are expressed as Horn clauses. GP deals basically with functional languages such as Lisp and Scheme and makes extensive use of Lisp's S-expressions.

A discussion of the conceptual and practical difficulties of a cross-paradigm comparison between ILP in general and GP can be seen in [Koza 96b]).

The development of Shrinp was guided by two main principles: autonomy and the scarcity of examples. The system was designed to operate even in the hardest conditions when no information is available other than the positive examples and the modes of the target predicate, a very likely situation in real-world applications. Predicate invention in such conditions is not an easy task because the system does not have access to all examples or to an oracle that could somehow help. Shrinp is the first learner that inverts implication to invent its own predicates. Details on this important aspect are explained in the next chapter.

chapter six

6. Predicate Invention

This chapter is dedicated to *predicate invention* (PI). It shows how the invention of recursive predicates can be achieved by adding a new literal Q to an incomplete clause and inducing a recursive definition for the newly created predicate given the assumption that Q is not included in the background knowledge. Such issues as determining Q's arity and argument structure as well as finding examples for the new predicate are also discussed. This chapter contains as well an explanation of how constructive learning is implemented in Shrinp. The system's language bias for learning both LRDs and RRDs is also defined. The criteria for stopping the PI process are presented at the end of the chapter. We decided to make this chapter rather lengthy to overcome the lack of uniformity in the few papers on this topic ([Pazzani 96]).

6.1 The Predicate Invention Task

The weakness of many concept learning systems lies in the fact that all the necessary background knowledge K has to be supplied. If K is too small, the system fails to learn many relations; if too large, the system's performance may be degraded. Optimal background knowledge for a certain target relation to be learned can only be provided when the user knows in advance what literals participate in the definition of the relation. It is crucial for successful learning that an appropriate set of supporting relations be available.

The invention of new terms can remedy the situation. A learner that is capable of *constructive learning* is a system that extends the language with the definition of new predicates created by itself. According to [Ling 91b], new terms and their postulates may serve as higher-level building blocks for finding the definition of a target program. As the new terms do not appear in K, a burden is taken from the user's shoulders, who will not have to provide them anymore. PI is considered necessary whenever there is no finite logic program that uses only the fixed vocabulary of predicates from the evidence and the background knowledge that can solve the ILP task at hand ([Flener 95]). Indeed, the invention of a new predicate is not only useful for syntactically shifting the language bias. It also extends the language with new, interesting predicates. And given the finite nature of the hypothesis language, PI appears as the most promising approach to implement bias shift in ILP systems ([Stahl 95b]).

For a new term to be *necessary*, it must be recursively defined. If not, it will, at most, make the learned program more compact and K will still be needed. Unnecessary although *useful terms* do not affect the learnability of a program (see Section 2.4). Because the W operators described in Section 3.3.2 are not capable of producing recursive clauses, they are definitely ruled out as a mechanism that invents necessary predicates.

The *predicate invention task* is similar to the ILP problem stated in Section 2.3. Given an incomplete clause $C \in L$, $C = (C_0 \text{ :- } C_1, C_2, \ldots, C_n)$ and a set of positive and negative examples of C_0, C_0^+ and C_0^-, the idea is to find a new literal Q and provide its definition D so that:

- $C' = C \cup Q$
- $C', D, K \vdash c^+$ for all $c^+ \in C_0^+$
- $C', D, K \nvdash c^-$ for all $c^- \in C_0^-$

In Shrinp's case, the set of supporting predicates is not necessary, so K is empty.

Note that the first condition imposes no restrictions on the placement of the new predicate in the incomplete clause C. This freedom *per se* may imply a language bias if a fixed position is chosen, say in the end of C, as done by the system Champ (see Section 4.6).

Four main steps are involved in PI, namely:

1. Choosing the incomplete clause C to be specialized through the addition of the new literal Q;

2. Determining an argument structure for the call to Q in the new clause C';

3. Finding examples for the new predicate; and

4. Building its definition D.

Now let us analyze these steps one by one. Due to its importance, the determination of the arguments of the new literal is treated in a separate section, 6.1.1.

The selection of the right clause C is important for efficiency reasons. Exhaustive search can be employed on the set of incomplete clauses but is too time-consuming. One alternative is to restrict PI to those incomplete clauses that cover negative examples ([Bain, Muggleton 91]). Another alternative, employed by the system Champ, uses special heuristics to order the clauses according to a scoring system based on *likelihood* ([Kijsirikul *et al.* 91]) and FOIL's *gain* heuristics ([Quinlan 90]). In any case, only over-general incomplete clauses shall be considered in this step. Over-specialized clauses can be corrected without new predicates. Decision criteria for introducing new predicates are discussed in [Stahl 96].

To find the sets of positive and negative examples for the new predicate, Q^+ and Q^-, information about covered examples can be used. By instantiating the clause C' with the values provided by the examples of C the two sets can be found by calculating:

- $Q^+ = \{ Q\theta \mid C_0\theta \in C^+ \}$
- $Q^- = \{ Q\theta \mid C_0\theta \in C^- \}$

Care must be taken while determining the elements of Q^- (see Section 6.5 below). What makes PI particularly difficult is that in general the evidence of the new predicate is quite insufficient. The examples of Q tend do be sparse with respect to the Basic Representative Set (see Section 2.3 and [Lapointe, Ling, Matwin 93]). Once examples of the new predicate are found, the induction of the definition D can be pursued. Formally speaking,

Given:

- Q^+, a set of positive examples of the new predicate

- Q^-, a (possibly empty) set of negative examples of the new predicate such that $Q^+ \cap Q^- = \varnothing$
- L, the hypothesis language and
- a background knowledge such that K $\not\vdash e^+$ and K $\not\vdash e^-$, for all $e^+ \in Q^+$ and $e^- \in Q^-$,

  Find:

- a definition $D \in L \cup \{Q\}$ that covers all examples in Q^+ but none in Q^-.

That is, D,K $\vdash e^+$ but D,K $\not\vdash e^-$ for all $e^+ \in Q^+$ and $e^- \in Q^-$, and C',D,K $\vdash c^+$ but C',D,K $\not\vdash c^-$ for all $c^- \in C_0^-$ and $c^+ \in C_0^+$. Due to a similarity with the ILP problem, PI's last step can be made simpler by merely calling the learner itself recursively.

6.1.1 Finding the arguments of the new predicate

Once an incomplete clause C has been selected, the system has to determine the arguments of the new literal Q. This choice is crucial as the efficiency and the success of the learning process depend on the examples of Q. System designers must take into account that the complexity of inducing a definition D for Q grows exponentially with the number of arguments ([Stahl 95a]). A decision has to be made regarding:

- The *arity n* of the new predicate
- An adequate argument structure for the call to Q in C that may take into account the terms that already appear in C or use new ones.

The number of combinations may be infinite for any small *n* as several terms can take part and a function may be applied to itself an infinite number of times. Any number of the following potential terms is acceptable:

- A variable that already appears in C
- A new variable
- A constant that already appears in C
- A new constant

- Complex terms that involve functions applied to any of the items above
- Repeated terms

A few restrictions help reduce the size of the tuples to be considered. Whatever the choice, it must be enough for the specialized clause $C' = C \cup Q$ to *discriminate completely* the examples of C_0. That is, there should be no $e^+ \in Q^+$ such that $C' \not\vdash e^+$ and at the same time no $e^- \in Q^-$ such that $C' \vdash e^-$.

Also, the arguments of Q must be chosen in such a way that the sets of positive and negative examples found comply with the principle that they may not overlap (Section 2.3). An important result is provided in [Stahl, Weber 94] that shows that it suffices to consider as arguments 1-level deep (simple) terms only like *newp(a, f(0,b,b))* but not such complex terms as *newp(g(g(c)), f(f(a,g(b),c),g(a),b))*. Even so, while these restrictions eliminate potential argument tuples, a large number of tuples are still left over for consideration. The number of possible arguments can still grow large even though the number of functions, variables and constants is finite. An ordering on the potential calls to a new predicate is formulated in [Muggleton 93].

A minimal set of arguments is pursued by some systems. The DBC algorithm (Section 4.6) is used by the system Champ to find a minimal set of variables with no repetition. The algorithm employs a greedy search on a tuple containing all the variables in the clause. It removes them one by one, testing if discrimination is still possible with the remaining tuple. The minimal set is not guaranteed but at least the DBC succeeds in getting rid of some variables. Other approach ([Stahl, Weber 94]) combines the DBC method with concepts from *rough set theory* ([Pawlak *et al.* 88]). A set of *core variables* is first determined before further variables are added. Only then is discrimination checked. The core arguments help reduce the number of heuristic choices of the greedy algorithm. A more severe restriction is imposed by the KRT/CLT methods ([Wrobel 94]) which invent only unary and binary new predicates.

6.2 Predicate Invention in Shrinp

The system Shrinp has a module specially designed to deal with constructive learning. The Predicate Invention Module (PIM), presented in Section 5.3.3, is called when all induced PRDs

have failed the coverage test. It specializes the incomplete clause C by adding a new literal to its left or to its right, thereby forming an RRD or an LRD, respectively. The problems addressed in the previous sections are treated in the following way:

Choosing an incomplete clause. PIM tries all clauses in the order that they are received from the Filter Module.

Determining an argument structure. Following the discussion in Section 4.6, the small number of examples given to Shrinp does not allow for a minimal set of variables to be found that enables complete and consistent discrimination. The selection of variables to form the arguments of the call to the new predicate is carried out through the use of *argument-binding chains* (ABCs), which also permit the creation of new variables. See Sections 6.3 and 6.4 below.

Finding examples. Two sub-modules are implemented: the Positive and the Negative Examples Builders introduced in Section 5.3.3. They follow the guidelines discussed in the previous section. Special algorithms remove non-ground positive instances and complete non-ground negative instances that result from running the extended clause C' on the examples of the target predicate. System-generated examples are also implemented (see GENEX in Section 5.4). Several issues are discussed in Section 6.5.

Finding a definition. Shrinp is called recursively with Q^+ and Q^- as examples. A sequence of recursive calls may occur when the definition of a new predicate requires the invention of another new predicate and so on (see Fig. 5.3). The criteria for stopping the chain of recursive calls to Shrinp are explained in Section 6.6.

6.2.1 Language bias

Now we can state Shrinp's language bias. The definitions learned by the system are the class of linear PRDs, LRDs and RRDs with only one base clause as long as they comply with the argument structure defined by the argument-binding chains (see Sections 6.3 and 6.4). The recursive clauses are restricted to three literals at most: the head, the recursive call and, if necessary, a call to a new predicate. However strong the bias may seem, it accounts for several useful and classical definitions to be learned (and invented). Some of them are *append*, *reverse*, *subset*, *sum*, *last_of* and *permut*. W. Cohen named such definitions *constant-depth determinate*

clauses ([Cohen 95a]).

The language bias is illustrated below where *newp* is a predicate invented by Shrinp and the *Args* are lists of terms. Details on the bias for choosing terms for the call to the new predicate are shown next.

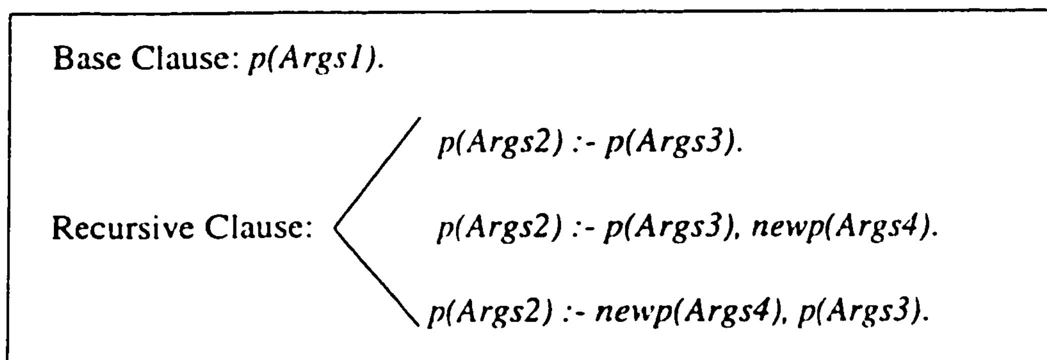


Fig 6.1. Shrinp's language bias.

6.2.2 Argument structure

Special care has to be taken while finding an adequate argument structure for the new literal Q. The system Champ, for instance, limits itself to the variables that occur in the clause. It has at its disposal all positive and negative examples of the target predicate (up to a limit on the size of the structures used). Therefore, it is a straightforward task to check if the addition of a new literal to the clause C leads to a discrimination of the examples. Shrinp, on the other hand, has limited access to the examples, as only a few of them are available. In such circumstances, not only is it impossible to find a minimal set of variables for the call to the new literal, the computation of negative examples for Q becomes seriously compromised. The choice of arguments for Q has to be based solely on the terms that already appear in C. The approach followed by Shrinp mixes these terms and new variables depending on the type of recursive definition being induced, LRD or RRD. The creation of new terms, complex or not, from those that are already in the clause would result in a combinatorial explosion. Our approach avoids this problem and allows for the use of more kinds of arguments than Champ's. The choice of arguments, represented by constants, repeated terms and new variables, is presented in the next few sections.

6.2.2.1 Constants

Constants might as well be included as arguments of the new literal. But let us show why their inclusion is pointless. Recall that they have a generating term of depth zero. If a constant is placed as the argument in position *i* of Q, it will appear in the same position in all elements of Q^+. That means that any definition induced for the new predicate will also have the same constant in the *i*th position both in the base clause and in the head and body of the recursive clause. What use is an argument that is always the same in all steps of a recursive chain? The use of constants, thus, has no effect and is not allowed in Shrinp.

Let us see an example. Consider the incomplete clause C1 in Table 6.1 and the two possible completions of C1, C1' and C1". Both *newp1* and *newp2* are new predicates with modes *[i]* and *[i,i]*, respectively. Note that *newp2* has one more argument than the other new predicate: the constant zero. *LEven* is a relation that succeeds if its argument is a list of even numbers.

Table 6.1. Possible completions for a definition of *lEven*.

C1	C1'	C1"
lEven([A|B]) :-	*lEven([A|B]) :-*	*lEven([A|B]) :-*
lEven(B).	*newp1(A),*	*newp2(A,0),*
	lEven(B).	*lEven(B).*

If the positive examples in Table 6.2 are used, the instances found for both new predicates are as indicated in the same table. For simplicity we do not care about the set Q^- at this time.

Table 6.2. Positive examples found for *newp1* and *newp2*.

$LEven^+$	$newp1^+$	$newp2^+$
lEven$([s^2,0])$.	*newp1*(s^2).	*newp2*$(s^2,0)$.
lEven$([s^4])$.	*newp1*(0).	*newp2*$(0,0)$.
	newp1(s^4).	*newp2*$(s^4,0)$.

When Shrinp is separately run on $newp1^+$ and $newp2^+$, definitions D1 and D2 are produced, respectively. Note how the constant *0* is repeated as the second argument of all occurrences of *newp2* in D2, although this definition is correct. The presence of the constant zero

did not contribute to the success or failure of the learning, but rather consumed precious milli-seconds of the processor time.

```
newp1(0).                                                    (D1)
newp1(s(s(X)) :- newp1(X).
```

```
newp2(0,0).                                                  (D2)
newp2(s(s(X),0) :- newp2(X,0).
```

6.2.2.2 Repeated arguments

Just like constants, the repetition of arguments is useless. They will simply duplicate the value of an argument, thereby yielding the same terms on the base and recursive clauses of the induced definition. Repeated arguments are, for this reason, disallowed in Shrinp. As an example, let C2 below be another extension of C1 above where the modes for *newp3* are *[i,i]*.

```
lEven([A|B]) :-                                              (C2)
      newp3(A,A),
      lEven(B).
```

For the same positive examples of *lEven* in Table 6.2, Shrinp would find the following set of examples of the new predicate: $\{newp3(s^2,s^2),\ newp3(0,0),\ newp3(s^4,s^4)\}$. These examples would give rise to the definition D3 below where the first argument is duplicated in all literals. It is easy to see that the second argument has no effect.

```
newp3(0,0).                                                  (D3)
newp3(s(s(X),s(s(X))) :- newp3(X,X).
```

6.2.2.3 New variables

Shrinp resorts to new variables only when it is learning RRDs (see Section 6.3). In an LRD, a new variable *V* can only be included if it is repeated as an argument of Q as there is no other literal to the right of Q where *V* can be referenced. As an example, consider an incorrect left-recursive extension of the clause C1 shown earlier. The new predicate *newp4* in clause C3 below has *V* as a new variable.

```
lEven([A|B]) :-                                              (C3)
      lEven(B),
      newp4(A,V).
```

The examples in Table 6.2 will produce the following set of positive instances for *newp4*: {*newp4*(s^2,_), *newp4*(0,_), *newp4*(s^4,_)}. As usual, the underscore stands for unnamed variables. As none of these examples is ground, they will all be eliminated by the Positive Examples Filter and nothing will be learned from the remaining empty set. New variables in LRDs, therefore, are useless.

6.3 Argument-Binding Chains for Learning RRDs

Constants, complex terms and repeated terms have been seen to be disallowed as arguments of the new literal. The choice of arguments for RRDs makes use of the variables and terms already in the clause and may also introduce new variables. Two types of program structures are defined, depending on whether a new variable is introduced: RV and RNV, respectively (Fig. 6.2). The two groups comprise five argument-binding chains, which are explained next. Examples of relations that can be learned through their use are shown later.

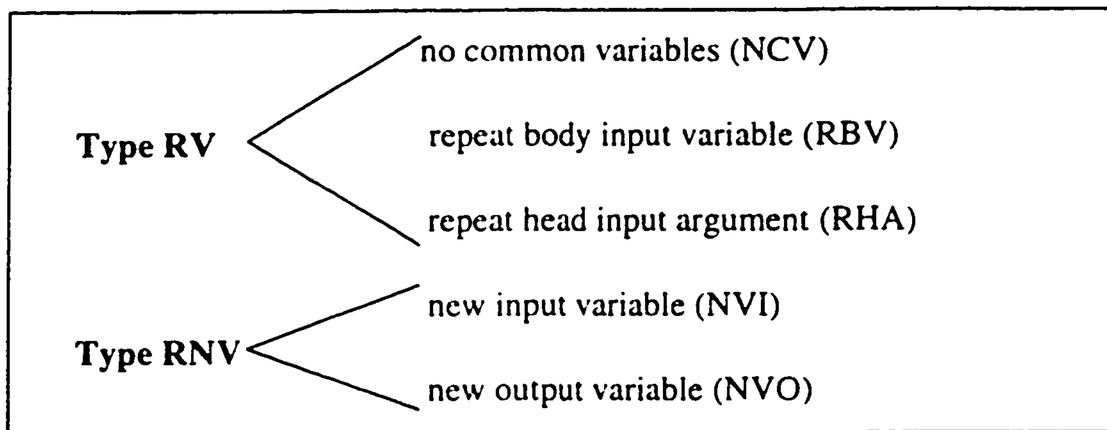


Fig 6.2. Types of argument-binding chains to learn right-recursive definitions.

a) NCV

This is the simplest of the argument-binding chains. It is also a starting point for all the others. All NCV does is to come up with a list of arguments that do not intersect with those of the recursive call. Because of this disjointness, the new literal could be placed anywhere in the body. The first place is chosen in order to derive an RRD, which in general executes somewhat faster than LRDs when *tail recursion* is detected by the compiler ([Aho, Sethi, Ullman 86]). All variables must be used at least twice in the clause. Some important relations are learned as a

result of NCV. In this and the other argument chains, the system first tests if the argument to be inserted into the new predicate is a constant. If so, the argument is not included.

b) RBV

This argument chain allows one input variable of the recursive call at a time to be repeated as an input argument of the new literal. For each variable that is repeated, a new clause is produced. Only body variables that are not yet present in the new literal are considered. The variables that appear only once in the clause are first added to the new literal.

c) RHA

Here head input arguments whose terms do not appear in the new literal are added as input arguments to the latter. As before, each inclusion gives rise to a different clause and the variables that appear only once in the clause are first added to the new literal.

d) NVI

A variable that has not been used in the clause is successively substituted for each body input argument in this argument chain. The new variable is then added to the output arguments of the new literal. Arguments in the head are also included into the new literal as long as the terms that appear in the head do not intersect with any current variable of the new literal, in which case they replace the variable. The output arguments of the body are left as is. The main idea is to make the new literal be the link between the input arguments of the head and the recursive call. All variables have to be used twice.

e) NVO

The arguments of the recursive call are successively substituted by a new variable that is included as an output argument of the call to the new predicate. All input arguments of the head are also inserted into the new literal.

The effect of these simple argument chains can be observed in Table 6.3. The table shows right-recursive definitions learned by Shrinp as provided by the argument chains. To make the description more clear, we substituted more meaningful names for the actual names of the invented predicates. The predicate *isSorted*, for instance, is learned through the use of RBV by

repeating as an input argument of the new predicate the variable *B*, which appears in the input argument of the recursive call to *isSorted*. The induced definition found for the new predicate equals that of the *lessEq* predicate. This ABC differs from RHA in that RHA repeats arguments from the head, not variables from the body, as can be seen in the call to *member*, where *[C|D]* came from the head of the clause. Note how in the last ABC two input terms from the head are used by the new predicate *myDelete* to return a variable that is used by the call to *eqList*, thereby providing a link between the head and the recursive call.

Table 6.3. Predicates invented through the use of the RRD argument chains.

<table>
<tr><th>C.</th><th>Incomplete clause</th><th>Induced Clauses</th><th>Invented predicate</th></tr>
<tr><td>N
C
C
V</td><td>lEven([]).
lEven([A|B]) :-
lEven(B).</td><td>lEven([]).
lEven([A|B]) :-
isEven(A),
lEven(B).</td><td>isEven(0).
isEven(s(s(A)) :-
isEven(A).</td></tr>
<tr><td></td><td>doubles([],[]).
doubles([A|B],[C|D]) :-
doubles(B,D).</td><td>doubles([],[]).
doubles([A|B],[C|D]) :-
twice(A,C),
doubles(B,D).</td><td>twice(0,0).
twice(s(A),s(s(B)) :-
twice(A,B).</td></tr>
<tr><td>R
B
V</td><td>isSorted([A]).
isSorted([A,s(B)|C]) :-
isSorted([s(B)|C]).</td><td>isSorted([A]).
isSorted([A,s(B)|C]) :-
lessEq(A,B),
isSorted([s(B)|C]).</td><td>lessEq(0,A).
lessEq(s(A),s(B)) :-
lessEq(A,B).</td></tr>
<tr><td>R
H
A</td><td>subset([],[A|B]).
subset([A|B],[C|D]) :-
subset(B, [C|D]).</td><td>subset([],[A|B]).
subset([A|B],[C|D]) :-
member(A,[C|D]),
subset(B,[C|D])</td><td>member(A,[A|B]).
member(A,[B,C|D]) :-
member(A,[C|D]).</td></tr>
<tr><td>N
V
I</td><td>eqList([],[]).
eqList([A|B],[C|D]) :-
eqlist(B,D).</td><td>eqList([],[]).
eqList([A|B],[C|D]) :-
myDelete(A,[C|D],E),
eqList(B,E).</td><td>myDelete(A,[A|B],B).
myDelete(A,[B,C|D],[B|E]) :-
myDelete(A,[C|D],E).</td></tr>
<tr><td>N
V
O</td><td>repeat([A|B],[A|B]).
repeat([A|B],[C,D|E]) :-
repeat([A|B],[D|E]).</td><td>repeat([A|B],[A|B]).
repeat([A|B],[C,D|E]) :-
remFront([A|B],[C,D|E],F),
repeat([A|B],F).</td><td>remFront([],[A|B],[A|B]).
remFront([A|B],[A,C|D],[E|F]) :-
remFront(B,[C|D],[E|F]).</td></tr>
</table>

6.4 Argument-Binding Chains for Learning LRDs

A slight adjustment is necessary before LRDs can be learned. The incomplete clause received by the Left Inventor (see Section 5.3.3) usually contains complex terms as output arguments of both the head and body literals. First, Shrinp replaces each of these arguments with a different new variable. Once the modification is performed, the problem is then reduced to inventing a predicate that takes as input the body output variables (and possibly input arguments from the head) and produces as output the output variables in the head. The idea is to let the new literal provide the connection between what is returned by the recursive call and what is to be returned by the clause.

Two argument-binding chains are defined in Type LV for learning LRDs (see Fig. 6.3):

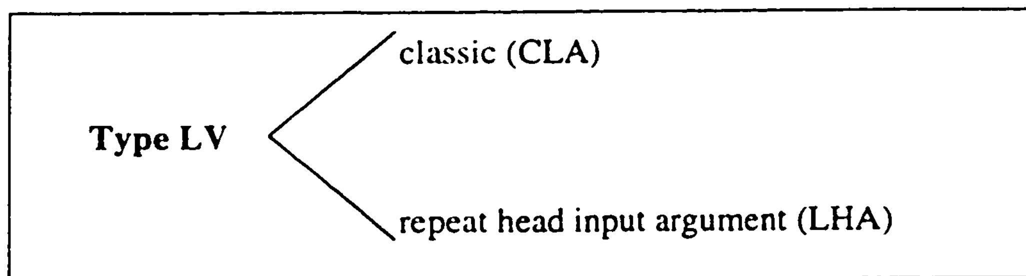


Fig 6.3. Types of argument-binding chains to learn left-recursive definitions.

a) CLA

In this argument chain, the new predicate will contain only the variables that are used exactly once in the incomplete clause. Of course it will contain the new variables introduced to the head and body literals in the first step. This is the simplest and most common case of LRD.

b) LHA

A little like the RHA argument chain, the LHA extends the literal produced by the CLA argument chain with one or more of the input arguments in the head. Arguments that share variables with the new literal are ignored.

The Type LV argument chains are enough to make Shrinp learn the predicates shown in Table 6.4. The predicate *select*, for instance, is learned when Shrinp tries to induce a left-recursive definition for the predicate *permut*. Note how the output argument of the head, *[C|D]*, was changed to a new variable, *E*, that is later used in the call to *select*, making this invocation of

the new predicate the link between what is returned by the recursive call and what the clause outputs. The predicate *plus* was learned both for *sum* and for *sumToN*.

Table 6.4. Predicates invented through the use of the LRD argument chains.

<table>
<tr><th>C.</th><th>Incomplete clause</th><th>Induced Clauses</th><th>Invented predicate</th></tr>
<tr><td></td><td>permut([],[]).</td><td>permut([],[]).</td><td>select(A,B,[A|B]).</td></tr>
<tr><td>C
L</td><td>permut([A|B],[C|D]) :-
permut(B,D).</td><td>permut([A|B],E) :-
permut(B,D),
select(A,D,E).</td><td>select(A,[B|C],[B,D|E]) :-
select(A,C,[D|E]).</td></tr>
<tr><td>A</td><td>sum([],0).</td><td>sum([],0).</td><td>plus(0,A,A).</td></tr>
<tr><td></td><td>sum([A|B],s(s(C))) :-
sum(B,C).</td><td>sum([A|B],E) :-
sum(B,D),
plus(A,D,E).</td><td>plus(s(A),B,s(C)) :-
plus(A,B,C).</td></tr>
<tr><td>L</td><td>sumToN(0,0).</td><td>sumToN(0,0).</td><td>plus(0,A,A).</td></tr>
<tr><td>H
A</td><td>sumToN(s(A),s(s(s(C)))) :-
sumTo(A,C).</td><td>sumToN(s(A),E) :-
sumToN(A,D),
plus(s(A),D,E).</td><td>plus(s(A),B,s(C)) :-
plus(A,B,C).</td></tr>
</table>

6.5 Finding Examples for the New Predicate

There are a few important issues concerning the way examples of the new predicate Q are computed. The pair of equations below means that to find examples for Q, Shrinp has to run the definition C' on the given positive and negative examples of the target predicate and record the instantiations of the call to Q (see Section 6.1).

$$Q^+ = \{ Q\theta \mid C_0\theta \in C^+ \}$$

$$Q^- = \{ Q\theta \mid C_0\theta \in C^- \}$$

Besides checking whether the sets Q^+ and Q^- may overlap, other measures must be taken. First, while executing C', the recursive call must be executed again and again until the base clause is reached. Each instantiation of Q's arguments gives rise to an example of the new predicate. The execution of the recursive calls generates as many examples as possible. The more examples, the greater the chance of finding a correct definition for Q, even if the examples lie in

the same resolution chain. For instance, consider clause C1' as in Table 6.1 shown earlier. The instantiation of the variable *A* in *newp1* for the positive example *lEven([s^2,0,s^4])* produces *newp1(s^2)*. When the subsequent recursive calls to *lEven* are followed, two more examples are generated: *newp1(0)* and *newp1(s^4)*. So, for the sake of increasing the number of examples, all recursive calls to the target predicate must be pursued. A few problems may occur, as discussed next.

6.5.1 Problems with unnamed variables

At times, the induced examples contain *unnamed variables*. This situation occurs exactly when the new literal in C' has variables that do not become instantiated by the recursive call, a very likely scenario since only a fraction of the possible examples of the target predicate are available. In these cases, the examples of Q are non-ground. Non-ground positive examples have to be discarded as they are too general. Consider, for instance, the task described in Table 6.5:

Table 6.5. Examples and incomplete clause for the predicate *sum*.

sum^+	sum^-	BC	C'
sum([0,s^2]),s^2)	*sum([s^2],0)*	*sum([],0)*	*sum([A|B],E) :-*
sum([0,s^1],s^1)	*sum([],s^2)*		*sum(B,D),*
sum([s^1],s^1)	*sum([0,s^2],s^1)*		*newp5(A,D,E).*

The predicate *newp5* is meant to be the relation *plus*. From the set of positive examples sum^+, the following members of Q^+ are found: {*newp5(0,_,s^2)*, *newp5(s^2,0,_)*, *newp5(0,s^1,s^1)*, *newp5(s^1,0,s^1)*}, where the underscore stand for unnamed variables. Clearly the two first elements of Q^+ are incorrect. *newp5(s^2,0,_)*, for instance, tells that *2* + *0* equals any number! Both examples are removed from the set.

On the other hand, the first negative example produces a valid member of Q^-, *newp5(s(s(0)),0,0)*. The second one does not match the head of any clause and is left aside, while the third one produces *newp5(0,_,s^1)* and *newp5(s^2,0,_)*. Shrinp ignores them and corrects the non-ground elements of Q^-. If the unnamed variables are replaced by any value, a valid negative example is created. The Negative Examples Completer substitutes *0* and the empty list for each unnamed

variable (see Section 5.3.3). The final sets are composed of ground examples only:

$Q^+ = \{newp5(0,s^1,s^1), newp5(s^1,0,s^1)\}$

$Q^- = \{newp5(s^2,0,0), newp5(0,0,s^1), newp5(0,[\,],s^1), newp5(s^2,0,0), newp5(s^2,0,[\,])\}$

It is easy to see that the examples in Q^+ and Q^- are all correct.

6.5.2 Problems with using negative examples of the target predicate

Another problem that may occur is very serious. Because complete discrimination is impossible, the computation of negative examples may result in valid positive examples taken as negative examples! See, for instance, what happens when the (correct) definition of *doubles* below is run on the negative example *doubles([s², s¹],[0,s²])*.

```
doubles([],[]).
doubles([A|B],[C|D]) :-
    twice(A,C),
    doubles(B,D).
```

The set of negative examples found for *twice* is $\{twice(s^2,0), twice(s^1,s^2)\}$. But that is wrong because $twice(s^1,s^2)$ should actually be in the set of positive examples of *twice*! To fix that, the best Shrinp can do is to generate Q^+ first and then remove from Q- the elements that are in the intersection of the two sets. Needless to say, if $twice(s^1,s^2)$ is not a member of Q^+, the above definition will fail and Shrinp will not be able to find the target definition for *doubles*. Not much can be done since the training set for *doubles* is too small.

This example illustrates that it is not always possible to safely produce a series of negative examples for Q by recursively applying the induced definition on the user-given negatives. For this reason, Shrinp gives the user the chance to completely prevent the system from generating negative examples this way. The reader is referred to Section 5.6.2.5 for more details on this.

6.5.3 Problems with the argument structure in the head

Finally, one more problem concerns the argument structure in the head of the recursive clause. It

may be that the negative examples of the target predicate do not unify with the head because of a repeated variable. To illustrate, note that the negative example *-p([a,b],[c,d])* does not match the head of the clause C4 below:

```
p([A|B],[A|C]) :-                                        (C4)
    newp6(A),
    p(B,C).
```

If it could unify with C4, the precious negative example *-newp6(a)* would be generated[17]. In order to do that, Shrinp temporarily substitutes new terms for the output variables that are also input variables in the head of the clause before computing Q'. The result of inserting the new variable *V* in C4 is the more general clause C5 below. C5 matches *p([a,b],[c,d])* and allows the production of the expected negative example of *newp6*. After that, the clause C5 is rewritten back as C4.

```
p([A|B],[V|C]) :-                                        (C5)
    newp6(A),
    p(B,C).
```

The error-prone nature of the computation of examples for the new predicate once again attests to the need for system-generated negative examples as discussed in Section 5.4.

6.6 Criteria for Stopping

As with any other algorithm, there is always the question of stopping the recursive calls. Does Shrinp ever stop creating new predicates? The invention of a "parent" predicate *newp7* may lead to the invention of another new predicate, say *newp8*, which may need another one and so on (see Fig. 5.3). Of course the algorithm stops as soon as a PRD is induced that passes the coverage test. But what if no such definition is found? Shrinp has a few rules for putting an end to the recursive process of inventing predicates. The sequence of recursive executions stops when at least one of the following conditions holds:

---

[17]Only if the option to do so is active (see Section 5.6.2.5), but still nothing guarantees that *newp6(a)* is in fact negative.

- A PRD definition is induced. Success.
- The examples generated for the current new predicate are identical to those of the "parent" predicate. In this case, the "parent" predicate refers to the new predicate of the previous call in the chain of recursive executions of Shrinp.
- No more examples can be generated for the current new predicate. This situation occurs when the examples of the current definition are not enough for examples of the next new predicate one level deeper to be found.
- The maximum depth of recursive calls to Shrinp has been reached. This value is set by the *maxdep* parameter (see Section 5.6.1.1). This means that at most *maxdep* new predicates can be invented during a given learning session.

If a purely recursive definition is not learned, the current activation of Shrinp fails. In case of failure, the normal execution resumes, i.e., the next potential definition created by the "parent" level is tried. The current definition is discarded.

6.7 Discussion

This chapter showed the importance of inventing the predicates a system needs towards a successful learning. Shrinp completely ignores the need for background knowledge and relies totally on the predicates it invents. The main issues concern which clause to complete and how to complete it. An infinite number of options exists that include the use of the terms already in the clause, new terms or repeated terms. Shrinp restrains itself to simple terms that already appear in the clause but does not allow constants nor duplicated terms as determined by the argument-binding chains. New variables are only allowed when it is learning RRDs. This form of language bias is enough for several well-known relations to be learned together with the invention of their required supporting predicates. The fact that the definition of the new predicates is recursive makes the method implemented in Shrinp a true predicate invention mechanism. Experiments that show the effectiveness of the method are shown in the next chapter.

One conclusion of this chapter is that no method of generating negative examples by machine is completely guaranteed to be safe. There is always a chance of taking positives as

negatives or discarding correct definitions. But in the absence of other alternatives, i.e. given the scarcity or absence of negative evidence, these methods still account for several relations to be learned alone as results in the next chapter indicate.

Other researchers have also aimed at constructive learning. [Stahl 95a] maintains that it is desirable to use a minimal set of variables in the new predicate as the complexity of inducing the new definition grows exponentially with the number of arguments. The system Champ ([Kijsirikul, Numao, Shimura 92]), described in Section 4.6, uses a discriminant-based constructive induction algorithm to obtain a minimal set of variables for each new predicate that it creates. Miro ([Drastal, Raatz 89]) is an attribute-value system that constructs high-level attributes based on knowledge about how low-level attributes interact (a knowledge-guided constructive induction). A similar approach, that introduces new high-level attributes in decision trees, is described in [Pagallo, Hausler 90]. [Matheus, Rendell 89] also deals with the problems of inventing new terms on decision trees. The system Index ([Flach 93]) invents new predicates based on an interaction with the user who is supposed to help the system choose one of several dependencies. Other approaches to constructive learning can be found in [Schlimmer 87] and [Utgoff 86].

As explained before, Shrinp differs in a number of ways from other approaches with respect to PI. First of all, systems like LFP2 and CIGOL, which implement the V and W operators, are not able to invent recursive predicates. The system Champ requires that all positive and negative examples up to a certain complexity be given. SIERES deals with the absence of negative examples by employing the CWA and syntactically discarding over-general clauses that have unbound output variables. The problem is that there is an infinite number of clauses whose output variables are all bound but that still are over-general. Only Shrinp learns from a few examples and has means to deal with the likely absence of evidence, by generating its own negative examples for discarding over-general or otherwise incorrect base clauses and complete definitions. It is also capable of relearning the base clause when need be and discarding incorrect definitions through the use of both negative examples and a filtering process. Most importantly, both Champ and SIERES are based on inverting resolution, while Shrinp inverts implication, making it the first system of this kind to invent recursive relations.

The use of argument-binding chains is not new in ILP. Several other similar techniques have been employed in other systems: *clause schemes* ([Jorge, Brazdil 95] and [Chao 94]), *skeletons* and *techniques* ([Kirschenbaum, Sterling 91]), *program schemas* ([Lakhotia 89]), *rule models* ([Kietz, Wrobel 91]) and *clause templates* ([Bergadano, Giordana, Ponsero 89] and [Tausend 94]). Our technique is based on reusing the variables and arguments that are already in the clause. It also has means to create new variables by substituting them for arguments in the body while the other techniques are based on leaving blanks in the body of a clause that are later filled in with calls to literals in the background knowledge. The argument binding in this case follows a different path from Shrinp's.

chapter seven

7. Experimental Results

This chapter describes the experiments carried out with the learning systems Shrinp and CLAM, which were described in Chapter 5. The goal is to evaluate the ideas described earlier in this work. The experiment presented in Section 7.1 tests Shrinp's ability to learn several predicates from few hand-tailored examples. Section 7.2 contains the results of the experiment of running Shrinp, Golem and Progol with and without machine-generated examples, while Section 7.3 evaluates the effect of the options described in Section 5.6. Experiments with CLAM can be found in Section 7.4. A relation that cannot be learned is shown in Section 7.1.1.

Throughout all the experiments, an induced definition is considered correct only if it is complete and consistent with respect to a training set. All times reported are expressed in milliseconds and refer to the execution of the systems on a Sun SPARCStation 20. Shrinp and CLAM are currently implemented in Quintus Prolog Release 3.2.

7.1 Testing Shrinp on Some Predicates

This first experiment is very simple. All we want to test is Shrinp's main characteristic, that of learning from a small number of sparse examples several recursive definitions that comply with the language bias defined in Section 6.2.1. We also want to see if it can correctly invent the predicates it needs and find a definition for them. The hypothesis is that Shrinp can learn various relations if given the appropriate examples and modes. So no extra help from the user and no background knowledge are allowed. Recall from Section 2.1 that the modes of a relation p define

the input/output nature of each argument in p.

For the experiment, we chose 31 relations that are usually found in Prolog textbooks (see Table 7.1). Four of the relations are left-recursive (LRDs), eight are right-recursive (RRDs) and the rest is purely recursive (PRDs). The table also shows the purpose, the modes and the type of each recursive definition. We ran Shrinp only once for each relation, leaving the default options, discussed in Section 5.6, unchanged.

Table 7.1. A list of predicates and their modes.

Relation	Purpose	Modes	Type
addLast	add an element to the last position of a list	*[i,i,o]*	PRD
app	join two lists	*[i,i,o]*	PRD
checkNth	check if the nth element of two lists is the same	*[i,i,i]*	PRD
doubles	double the elements of a list	*[i,o]*	RRD
eqList	test if two lists have the same contents	*[i,i]*	RRD
eqNList	test if two lists have elements of same depth	*[i,i]*	RRD
extractNth	return the nth element of a list	*[i,o]*	PRD
fact	compute the factorial of a natural number	*[i,o]*	PRD
isEven	check if argument is an even natural number	*[i]*	RRD
isInteger	test if argument is a natural number	*[i]*	PRD
isSorted	check if a list is in ascending order	*[i]*	RRD
last_of	check the last element of a list	*[i,i]*	PRD
lessEq	check if number is not greater than another	*[i,i]*	PRD
lEven	check if elements of a list are even numbers	*[i]*	RRD
lOdd	check if elements of a list are odd numbers	*[i]*	RRD
member	test for membership	*[i,i]*	PRD
minus	subtract two natural numbers	*[i,i,o]*	PRD
myDelete	delete first occurrence of an element	*[i,i,o]*	PRD
nextTo	check if two elements are adjacent	*[i,i]*	PRD
permut	find a permutation of the elements	*[i,o]*	LRD
plus	find the sum of two natural numbers	*[i,i,o]*	PRD

remFront	remove prefix of a list	*[i,i,o]*	PRD
repeat	check if list's contents repeat several times at other list	*[i,i]*	RRD
reverse	reverse a list	*[i,o]*	LRD
sameDepth	test if two arguments have the same depth	*[i,i]*	PRD
select	select an element from a list	*[i,i,o]*	PRD
subset	check if a list is a subset of another	*[i,i]*	RRD
sum	find the sum of the elements of a list	*[i,o]*	LRD
sumToN	find the sum of the first *n* natural numbers	*[i,o]*	LRD
twice	return the double of a natural number	*[i,o]*	PRD
twiceAsLong	check if a list is twice as long as another	*[i,i]*	PRD

Table 7.2 contains the training set built for the experiment. Because the goal is mainly to test the possibility of learning in such harsh conditions, we hand-tailored the examples. Their number is kept to a minimum with an average of 2.7 for the positive instances and 2.6 for the negative instances. As explained in Section 2.1, the function "*s*" is used to represent natural numbers so that *s(0)* and *s(s(s(0)))* stand for *1* and *3*, respectively. As a syntactic sugar, we will also be using s^1, s^2, s^3 etc. to represent *1*, *2*, *3* etc. Note that the examples of *lEven* are the exact inverse of *lOdd*'s.

Table 7.2. Positive and negative examples used in this experiment.

Relation	Positive Examples	Negative Examples
addLast	*addLast(a,[b],[b,a])* *addLast(t,[e,c],[e,c,t])*	*addLast(a,[b,c],[b,a])* *addLast(t,[ ],[e,c,t])*
app	*app([d],[ ],[d])* *app([a,b],[c],[a,b,c])*	*app([a],[ ],[ ])* *app([d],[e],[d,e,f])*
checkNth	checkNth(s^1,[t,e],[t,g]) checkNth(s^3,[o,k,f],[y,n,f])	checkNth(s^1,[a,b],[ ]) checkNth(s^1,[ ],[a,b]) checkNth(s^2,[a],[a]) checkNth(s^1,[e],[a]) checkNth(s^1,[a,b],[d,b])
doubles	*doubles([0],[0])* doubles([s^1,s^2],[s^2,s^4])	doubles([s^1,0],[0,s^1]) doubles([0],[s^1])
eqList	*eqList([c,b,d,a],[a,b,c,d])* *eqList([b],[b])* *eqList([b,d,a],[a,b,d])* *eqList([a,b],[b,a])*	eqList([a],[a,c$\vert$d]) *eqList([a],[c])*

eqNList	*eqNList([q(q(0)),q(0)],[p(p(0)),p(0)])*		*eqNList([q(a)],[p(a)])*	
	eqNList([0],[0])		*eqNList([],[p(p(p(0)))])*	
			eqNList([q(q(q(0)))],[])	
extractNth	*extractNth*(s^2,[y,z],z)		*extractNth*(s^2,[x,y],x)	
	extractNth(s^3,[a,b,c,d],c)			
fact	*fact*(s^3,s^3*(s^2*s^1))	*fact*(s^2,s^2*s^1)	*fact*(s^4,s^4*s^3)	
isEven	*isEven*(s^2)	*isEven*(s^4)	*isEven*(s^5)	*isEven*(s^3)
isInteger	*isInteger*(s^1)		*isInteger([a])*	*isInteger(a)*
	isInteger(s^2)		*isInteger(s(a))*	
isSorted	*isSorted*([s^2,s^3])	*isSorted*([s^1,s^2,s^3])	*isSorted*([s^3,s^2,s^1])	*isSorted*([s^2,0])
	isSorted([s^1,s^3])	*isSorted([0])*	*isSorted*([s^1,0])	*isSorted([])*
	isSorted([0,s^1,s^3])		*isSorted*([s^2,0,s^3])	
last_of	*last_of(a,[c,a])*	*last_of(b,[x,y,b])*	*last_of([x,y],[x])*	
lessEq	*lessEq*(s^1,s^2)	*lessEq*(s^1,s^1)	*lessEq(s(b),s(s(b)))*	*lessEq*(s^1,0)
	lessEq(s^2,s^2)		*lessEq*(s^5,s^4)	
lEven	*lEven([0])*	*lEven*([s^2,s^4])	*lEven*([s^1])	*lEven*([s^3,s^5])
lOdd	*lOdd*([s^1])	*lOdd*([s^3,s^5])	*lOdd([0])*	*lOdd*([s^2,s^4])
member	*member(j,[k,j])*	*member(c,[a,b,c])*	*member(a,[c,d])*	*member(c,[a,b,e])*
minus	*minus*(s^2,s^3,s^1)		*minus*(s^1,0,0)	*minus*(0,0,s^1)
	minus(s^1,s^1,0)		*minus*(s^2,s^1,s^1)	*minus*(0,0,s^2)
myDelete	*myDelete(e,[f,e],[f])*		*myDelete(d,[d,r],[d])*	
	myDelete(c,[a,b,c,d],[a,b,d])			
nextTo	*nextTo(c,d,[a,h,c,d])*	*nextTo(e,[e,f,h,i])*	*nextTo(a,b,[b,a])*	*nextTo(a,b,[b])*
	nextTo(a,b,[c,a,b])		*nextTo(a,b,[a,c,b])*	
permut	*permut([p,b,d],[b,d,p])*	*permut([l,w],[w,l])*	*permut([z],[a])*	
	permut([l,t],[l,t])	*permut([t],[t])*	*permut([],[e,g])*	
	permut([b,d],[b,d])	*permut([w],[w])*		
plus	*plus*(s^2,0,s^2)	*plus*(s^1,s^1,s^2)	*plus*(0,0,s^1)	*plus*(s^1,0,0)
remFront	*remFront([a],[a],[])*		*remFront([e],[a],[])*	*remFront(a,a,[])*
	remFront([b,c],[b,c,b,c],[b,c])		*remFront([a],[a],[d])*	
			remFront([b,c],[b,c,b,c],[])	
			remFront([],[b,c,b,c],[b,c])	

repeat	*repeat([d],[d,d])*		*repeat([a],[b])*	*repeat([a],[])*
	repeat([a],[a])		*repeat([],[a])*	*repeat (a,b)*
	repeat([b,c],[b,c,b,c])		*repeat([b,c],[b,b,c])*	*repeat([],b)*
reverse	*reverse([f,g,h],[h,g,f])*	*reverse([g,h],[h,g])*	*reverse([a],[a,c])*	*reverse([x],[y])*
	reverse([c,b,a],[a,b,c])	*reverse([x],[x])*	*reverse([x,y],[x,y])*	
	reverse([b,a],[a,b])		*reverse([a,b,c],[b,c])*	
sameDepth	*sameDepth(s(s(0)),p(p(0)))*		*sameDepth(s(a),p(a))*	
	sameDepth(s(0),p(0))		*sameDepth(s(s(0)),p(0))*	
select	*select(l,[w],[w,l])*	*select(d,[],[d])*	*select(b,[],[b,a])*	*select(c,[e],[y,f])*
	select(p,[b,d],[b,d,p])	*select(w,[],[w])*	*select(c,[y,h],[y])*	*select(a,[a],[a])*
	select(a,[b],[a,b])			
subset	*subset([j,k],[k,j])*	*subset([t],[t])*	*subset([g],[k,l,i])*	*subset([a],[])*
	subset([c,b],[a,b,c])		*subset([at,g],[at])*	
sum	$sum([s^2,0],s^2)$	$sum([s^1,s^1],s^2)$	$sum([s^1],s^2)$	$sum([0],s^2)$
sumToN	$sumToN(s^1,s^1)$		$sumToN(s^2,s^1)$	$sumToN(0, s^1)$
	$sumToN(s^2,s^3)$		$sumToN(s^1,0)$	
twice	$twice(s^1,s^2)$	$twice(s^2,s^4)$	$twice(0,s^2)$	$twice(s^1,s^1)$
twice-AsLong	*twiceAsLong([a],[o,p])*		*twiceAsLong([a,r],[])*	
	twiceAsLong([a,b],[d,f,g,h])		*twiceAsLong([a,b],[f,g,i,t,a])*	

The result of the experiment is depicted below. Table 7.3 shows the learned definitions and the runtime for each relation. As expected, Shrinp was capable of learning a correct definition alone for all relations, with an average runtime of 2.4 seconds. Although the accuracy on testing data was not computed, it is clear that all induced definitions are 100% correct. Whenever the invention of a predicate was necessary, a new one was created with the name *newPred*n where *n* is a number generated by the program. For the relations *doubles*, *eqList*, *eqNList*, *isSorted*, *lEven*, *lOdd*, *permut*, *repeat*, *subset* and *sum*, Shrinp invented new predicates that correspond to the definitions of *twice*, *myDelete*, *sameDepth*, *lessEq*, *isEven*, *isEven*, *select*, *remFront*, *member* and *plus*, respectively.

Observe how the definition induced for *newPred12*, which would correspond to the relation *app*, perfectly fits the definition found for *reverse*. The base clause of *newPred12* builds

a list out of the first argument so that *reverse* can correctly return a list. Also, both *lEven* and *lOdd* invented the same predicate, *isEven*, but note that their definitions are different.

Although we do not care too much about the runtimes at this point, they are indicated in Table 7.3 only for purposes of comparing the time taken to learn different predicates. Note how the runtime varies for each relation, with the RRDs and LRDs not necessarily having the highest figures. *Select*, a PRD for one, took one second to be learned while *double*'s runtime was only 0.44 seconds. This proves that the invention of predicates, despite being a costly process, still can be carried out in an efficient manner. The argument-binding chains (ABCs) used to find the argument structure of the call to each new predicate are also shown (see Sections 6.3 and 6.4).

Table 7.3. Definitions of the previous relations learned by Shrinp from given examples.

Relation (ABC)	**Learned Definitions**	**Time**	**Invented Predicate**
addLast	*addlast(A,[],[A]).* *addlast(A,[B|C],[B,D|E]) :- addlast(A,C,[D|E]).*	140	
app	*app([],A,A).* *app([A|B],C,[A|D]) :- app(B,C,D).*	110	
checkNth	*checkNth(s(0),[A|B],[A|C]).* *checkNth(s(s(A)),[B,C|D],[E,F|D]):- checkNth(s(A),[C|D],[F|D]).*	400	
doubles (NCV)	*doubles([],[]).* *doubles([A|B],[C|D]) :- newPred6(A,C), doubles(B,D).*	440	*newPred6(0,0).* *newPred6(s(A),s(s(B))) :- newPred6(A,B).*
eqList (NVI)	*eqList([],[]).* *eqList([A|B],[C|D]) :- newPred10(A,[C|D],E), eqList(B,E).*	10117	*newPred10(A,[A|B],B).* *newPred10(A,[B,C|D],[B|E]) :- newPred10(A,[C|D],E).*
eqNList (NCV)	*eqNList([],[]).* *eqNList([A|B],[C|D]) :- newPred6(A,C), eqNList(B,D).*	410	*newPred6(0,0).* *newPred6(q(A),p(B)) :- newPred6(A,B).*
extractNth	*extractNth(s^1,[A|B],A).* *extractNth(s(s(A)),[B,C|D],E) :- extractNth(s(A),[C|D],E).*	310	

fact	*fact(s^1,s^1).* *fact(s(s(A)),s(s(A))*B) :-* *fact(s(A),B).*	180	
isEven	*isEven(0).* *isEven(s(s(A))) :-* *isEven(A).*	190	
isInteger	*isInteger(0).* *isInteger(s(A)) :-* *isInteger(A).*	50	

isSorted
 	
(RBV)
	isSorted([A]).
	
	isSorted([A,s(B)|C]) :-
	    newPred1(A,B),
	    isSorted([s(B)|C]).
		1817	*newPred1(0,A).*
			
			newPred1(s(A),s(B)) :-
			    newPred1(A,B).

last_of	*last_of(A,[A]).* *last_of(A,[B,C|D]) :-* *last_of(A,[C|D]).*	120	
lessEq	*lessEq(0,A).* *lessEq(s(A),s(B)) :-* *lessEq(A,B).*	1400	

lEven
 	
(NCV)
	lEven([]).
	
	lEven([A|B]) :-
	    newPred4(A),
	    lEven(B).
		360	*newPred4(0).*
			
			newPred4(s(s(A))) :-
			    newPred4(A).

lOdd
 	
(NCV)
	lOdd([]).
	
	lOdd([s(A)|B]) :-
	    newPred4(A),
	    lOdd(B).
		350	*newPred4(0).*
			
			newPred4(s(s(A))) :-
			    newPred4(A).

member	*member(A,[A|B]).* *member(A,[B,C|D]) :-* *member(A,[C|D]).*	140	
minus	*minus(0,A,A).* *minus(s(A),s(B),C) :-* *minus(A,B,C).*	90	
myDelete	*myDelete(A,[A|B],B).* *myDelete(A,[B,C|D],[B|E]) :-* *myDelete(A,[C|D],E).*	280	
nextTo	*nextTo(A,B,[A,B|C]).* *nextTo(A,B,[C,D,E|F]) :-* *nextTo(A,B,[D,E|F]).*	630	

permut (CLA)	*permut([],[]).* *permut([A|B],C) :-* *permut(B,D),* *newPred14(A,D,C).*	17900	*newPred14(A,B,[A|B]).* *newPred14(A,[B|C],[B,D|E]) :-* *newPred14(A,C,[D|E]).*
plus	*plus(0,A,A).* *plus(s(A),B,s(C)) :-* *plus(A,B,C).*	110	
remFront	*remFront([],A,A).* *remFront([A|B],[A|C],D) :-* *remFront(B,C,D).*	170	
repeat (NVO)	*repeat([A|B],[A|B]).* *repeat([A|B],[C,D|E]) :-* *newPred8([A|B],[C,D|E],F),* *repeat([A|B],F).*	1610	*newPred8([],[A|B],[A|B]).* *newPred8([A|B],[A,C|D],[E|F]) :-* *newPred8(B,[C|D],[E|F]).*
reverse (CLA)	*reverse([],[]).* *reverse([A|B],C) :-* *reverse(B,D),* *newPred12(A,D,C).*	11790	*newPred12(A,[],[A]).* *newPred12(A,[B|C],[B,D|E]) :-* *newPred12(A,C,[D|E]).*
sameDepth	*sameDepth(0,0).* *sameDepth(q(A),p(B)) :-* *sameDepth(A,B).*	180	
select	*select(A,B,[A|B]).* *select(A,[B|C],[B,D|E]) :-* *select(A,C,[D|E]).*	1000	
subset (RHA)	*subset([],[A|B]).* *subset([A|B],[C|D]) :-* *newPred8(A,[C|D]),* *subset(B,[C|D]).*	3550	*newPred8(A,[A|B]).* *newPred8(A,[B,C|D]) :-* *newPred8(A,[C|D]).*
sum (CLA)	*sum([],0).* *sum([A|B],C) :-* *sum(B,D),* *newPred8(A,D,C).*	1610	*newPred8(0,A,A).* *newPred8(s(A),B,s(C)) :-* *newPred8(A,B,C).*
sumToN (LHA)	*sumToN(0,0).* *sumToN(s(A),E) :-* *sumToN(A,D),* *newPred14(s(A),D,E).*	8510	*newPred14(0,A,A).* *newPred14(s(A),B,s(C)) :-* *newPred14(A,B,C).*
twice	*twice(0,0).* *twice(s(A),s(s(B))) :-* *twice(A,B).*	150	

twice-AsLong	*twiceAsLong([],[]).* *twiceAsLong([A|B],[C,D|E]) :-* *twiceAsLong(B,E).*	180	

7.1.1 A definition that is not learned

Shrinp is not capable of learning all recursive definitions that comply with its language bias. The process of searching for a base clause (borrowed from Crustacean) that agrees with the constraints specified in Section 3.5.2.1 makes it impossible for some definitions to be induced. Take, for instance, the predicate *times*. A possible target definition is shown below. It implements the multiplication of two integers, by returning the result in the third argument.

```
times(0, A, 0).                                  (D1)
times(s(A), B, C) :-
      times(A,B,D),
      plus(D,B,C).
```

One would expect the relation *plus* to be invented through the use of the LHA argument-binding chain (Section 6.4). However, the depth of the matching subterms of any set of positive instances is so varied that the regularities across the examples that Shrinp finds do not correspond to a valid base clause. Therefore, a correct definition can never be found.

To illustrate, consider the following positive examples:

$times(s^2, s^3, s^6)$ $\quad$ $times(s^3, s^2, s^6)$ $\quad$ $times(s^1, s^6, s^6)$ $\quad$ $times(s^6, s^1, s^6)$

To have 0 as the first argument of the base clause, we have to consider the following GTs and depths for the first argument of the first example:

s(1),s(1),1
s(1),2

The constraints specify that only the GTs with same depth are valid matches across the arguments of an example. For the third argument, only two GTs have the same depths, 1 and 2, as the ones above:

s(1), s(1), s(1), s(1), s(1), s(1),1
s(1), s(1), s(1), 2

The constraints also require that the same GT be applied to the same argument across the examples. For the second example we have the following GTs and depths for the first arguments:

s(1),s(1),s(1),1
s(1),3

and for the third arguments:

s(1), s(1), s(1), s(1), s(1), s(1),1
s(1),s(1),3

The next step of the algorithm shown in Section 3.5.2.1 finds the cross-product of the GTs across the examples. Only combinations that have the same sequence of operations for the same argument are considered valid matches. Clearly no combination of GTs of the first example that have the same depth match GTs of the second example that have the same depth. For instance, the combination:

s(1),2 and *s(1), s(1), s(1), 2* for the first argument have equal depth, 2
s(1),3 and *s(1),s(1),3* for the third argument have equal depth, 3

but although they share the same GT with respect to the first argument, *s(1)*, their GTs for the third argument are not the same. So, from these two examples, a base clause that has zero in both of these places is never generated. The same process can be applied to the other examples, $times(s^{1},s^{6},s^{6})$ and $times(s^{6},s^{1},s^{6})$, and the result will be the same. No set of positive instances of *times* will produce the target base clause because the induction of such a clause would not pass the constraints. Therefore, *times* is an example of a relation that cannot be learned by Shrinp although it satisfies the language bias.

7.2 Experiments with GENEX

A few questions regarding the execution of a learner in the absence of negative evidence are:

- How well can learning be done in the absence of negative examples?
- How would some of today's systems behave if they are given no negative examples,

considering that most of them were designed to operate in their presence?

- How could a system automatically generate its own negative examples?
- How good and reliable are computer-generated examples?

We have seen in Section 5.4 that the GENEX algorithm is able to automatically generate negative examples by corrupting the given positive examples. To answer some of these questions, we ran a number of experiments on three learners providing them with and without the negative examples produced by GENEX to see how they would perform without human-generated negative examples. The experiments as well as the results are described next.

7.2.1 Running GENEX on two learners

The purpose of this experiment is to see if two well-known systems, Golem ([Muggleton, Feng 90]) and Progol ([Muggleton 95]), can do any learning without any human-prepared negative examples. And if so, we want to see how effective machine-generated negative examples can be in terms of the accuracy of the learned definitions on unseen cases. The hypothesis is that the negative examples from GENEX should be enough for these two learners to work reasonably well once correct positive examples are also provided.

To run the experiment, we picked three of the predicates shown in Fig. 7.1. The training was done with seven randomly generated positive examples and with the negative examples provided by GENEX. We kept the maximum length of a list to three and the highest natural number to four. GENEX's mode was set to the reliable combination of *type* and *i-det*. The accuracy, computed as the percentage of positives covered and negatives not covered by a learned definition, was obtained from a randomly selected testing set of 100 examples, of which approximately 50% are positive (the exact percentage is also random). In the testing set, the limits for the length of a list and the highest natural numbers were fixed at three and five, respectively. In order to compare the performance of both systems, we also executed them in the presence of "correct" negative examples (i.e. hand-tailored examples) and in the complete absence of negative evidence, be it from the user or from the algorithm. The results, averaged over ten runs, are presented in Table 7.4.

Table 7.4. Golem's and Progol's accuracy and runtime on different sets of negative examples.

	Golem			Progol		
Predicates	**no examples**	**GENEX**	**"correct"**	**no examples**	**GENEX**	**"correct"**
doubles	50.2%	58.1%	65.3%	50.0%	71.0%	64.7%
	633	3903	4200	415	94	293
reverse	57.0%	56.3%	57.1%	57.9%	85.3%	85.3%
	448	1613	1439	487	263	270
sum	50.6%	66.0%	66.0%	49.9%	57.7%	57.7%
	8902	136213	135426	185	1153	1021

The values shown above are quite remarkable. For Golem, the use of GENEX as opposed to the execution without negative examples slightly increases the accuracy in nearly all cases, almost as much as when the "correct" negative examples are used. The runtime also increases but is comparable to the runtime obtained from running with a "correct" set.

For Progol, the results are more impressive. The use of computer-generated examples gave Progol a far better accuracy on the training set than when no negative examples are used. The accuracy was sometimes better than with the use of "correct" negative examples. But as the next table shows, Progol actually showed a better performance when given the hand-crafted set.

Table 7.5 has the number of times a correct definition was learned in all ten runs. When given hand-tailored negative examples, Progol was able to correctly learn a target definition for the predicates at least once. The same is true for Golem, except with respect to the predicate *reverse*, which it never succeeded to learn. A few correct definitions were also found from GENEX-produced examples only. It is worth noticing that in the absence of negative examples none of the systems was able to learn a correct definition of the target relations, which also explains why the CPU times were so low. Nonetheless, the learned definitions for this case had a high accuracy (cf. Table 7.4). The reason for that is simple. Both systems output the original seven positive examples when they cannot learn a non-ground definition. Because roughly half of the testing set is composed of negative examples, the accuracy of the output in this case is at least around 50%.

Table 7.5. Number of times out of ten that a correct definition was found alone.

	Golem			Progol		
Predicates	**no examples**	**GENEX**	**"correct"**	**no examples**	**GENEX**	**"correct"**
doubles	0	2	3	0	0	2
reverse	0	0	0	0	1	1
sum	0	2	2	0	1	1

So, as can be seen, our expectation was correct. Clearly Golem and Progol *can* in fact have an acceptable performance in the absence of user-supplied negative examples, as long as GENEX's instances are used. The results suggest that if these and other systems incorporated an algorithm for creating its own negative evidence, they might in general learn more effectively.

7.2.2 Running GENEX on Shrinp

We have just seen the benefits of GENEX to Golem and Progol. We know from Section 5.4 that GENEX works on three different operating modes, namely: *type*, *i-det* and *io-det*. The aim of this next experiment is to see how Shrinp acts when these modes are set as compared to when GENEX is not running. The *type* mode was said to be the most restrictive of GENEX's modes while *io-det* was regarded as the strongest. We expect, therefore, that the use of *io-det* alone brings a better result than the use of *type* alone. When combined, these two modes should have a performance better than any other combination of modes. We also anticipate poor results when GENEX is off, following the results obtained in the previous section, and as a consequence, a high runtime as well.

For this experiment, we ran Shrinp on several of the predicates shown earlier in Fig. 7.1. Unlike the previous experiment, we employed this time the same positive examples described in Table 7.2. No user-provided negative examples whatsoever took part in the experiment, though. We tried GENEX with the three modes separately and with a combination of *type* with *i-det* and *type* with *io-det*. The test was repeated when the algorithm was disabled. A testing set of 20 positive and 20 negative examples was hand-crafted to test the correctness of all induced definitions. The accuracy was not computed as the aim is exactly to see only the number of learned definitions and how many of them are correct.

Table 7.6 contains the number of final definitions learned by Shrinp in these circumstances. For the relation *subset* a target definition was never induced. That is because the system could always come up with a PRD that was consistent with the given examples. Note that only one correct definition was found for all other predicates no matter what modes were used or if GENEX was not running. So if only one definition was induced, it was correct.

Table 7.6. Number of final definitions learned by Shrinp given different modes.

predicates	none	*type*	*i-det*	*type + i-det*	*type + io-det*	largest reduction
app	8	1	1	1	1	88%
doubles	6	3	4	3	1	83%
extractNth	31	2	2	2	2	94%
fact	12	5	8	5	3	75%
last_of	8	1	1	1	1	88%
member	13	1	1	1	1	92%
myDelete	23	2	2	2	2	91%
plus	9	2	3	2	1	89%
reverse	6	3	4	3	1	83%
subset	16	5	5	5	2	88%
sum	18	3	8	3	1	94%
twice	19	6	5	5	5	74%

When *type* and *io-det* modes are combined, Shrinp could learn a correct definition for seven predicates alone. Take, for instance, the predicate *reverse* that was only learned alone when these modes were set. When the simple *type* mode was used alone, still the number of learned definitions was kept to a minimum for the vast majority of the predicates. Now compare all columns with the second column, which contains the total number of tentative clauses generated by Shrinp. In the absence of negative examples, that is the number of definitions that Shrinp induces for each relation. The activation of GENEX brought a significant improvement to the final output, no matter what modes were set.

The results also demonstrate that the combination of *type* and *io-det* modes can lead to a

reasonably better performance than when the *type* and *i-det* modes are applied individually. Although it seems from the table that the use of *i-det* alone was not very effective, recall that it is more reliable than *io-det* and that three of the tested relations have no output argument (*last_of*, *member* and *subset*). What the results show is that still the use of *i-det* alone contributes to a good performance of the system.

The best reduction in the number of induced definitions as compared to the execution of Shrinp without any negatives (column "*none*") is found in the rightmost column. It was as high as 94% in two cases.

Once again, the negative examples from GENEX were very effective. The presence of system-generated negative examples not only allowed Shrinp to learn several relations alone. As expected, it greatly reduced Shrinp's runtime. Fig. 7.1 shows the time reduction for learning the eight PRDs above with respect to both the use of GENEX-provided instances only (with *type* and *io-det* modes) and the complete absence of negative examples. A decrease in runtime was verified for all cases. For some predicates like *last_of* and *member*, the reduction was higher than 20%.

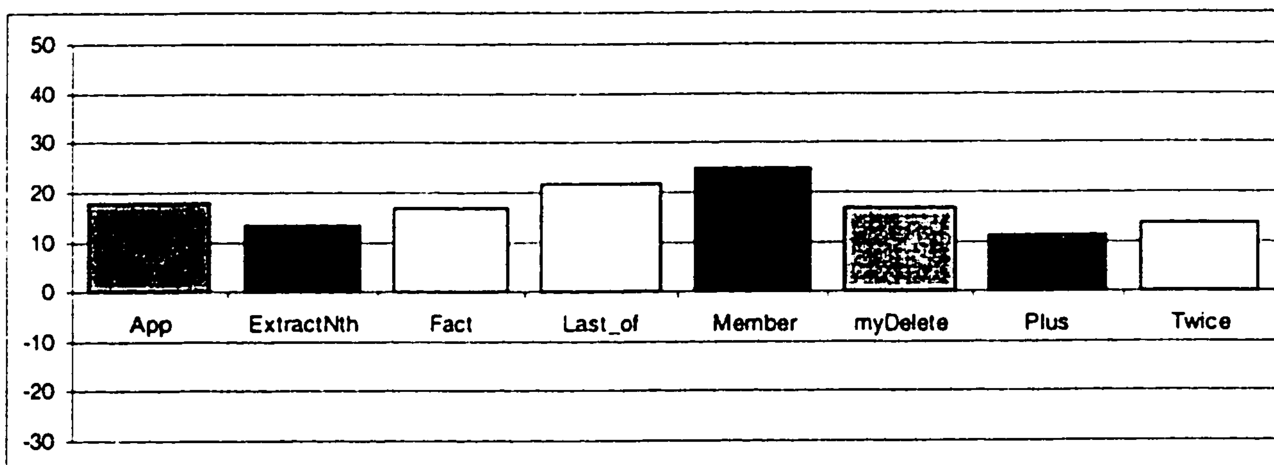


Fig 7.1. Time reduction in percentage for learning PRDs with *type* and *i-det* modes.

For the RRDs and the LRDs, the gain was not as great because of the overhead on trying new predicates recursively and finding a definition for them (see Fig. 7.2). In the case of *sum*, the runtime actually increased by almost 22%. The dramatic reduction of *subset* should not be taken into account, as a correct definition was not learned. Even so, the fact that other relations were induced alone compensates for a slightly slower execution on this type of relations.

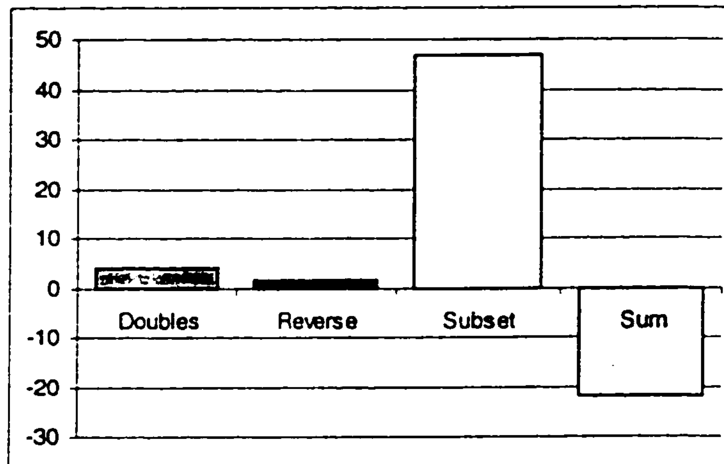


Fig 7.2. Time reduction in percentage for learning LRDs and RRDs with *type* and *i-det* modes.

7.3 Evaluating the Options

The aim of the next experiment is to evaluate the many options offered by Shrinp. Recall from Section 5.6 the existence of seven options as follows (the letters on the left comply with their nicknames in the implementation–see Appendix B):

a. GMODE
b. GENEX
c. NEGUSER
d. SHRINP
e. BCNEG
f. RELEARNBC
g. BADARG

What we want to test is their ability to efficiently help Shrinp learn a correct definition of some of the predicates shown in Table 7.1 while still not being too time-consuming. We are particularly interested in testing the performance of Shrinp when none of the options are enabled as compared with the activation of several combinations of them. For this last case, we want to see the difference between the execution of the system with single and multiple options. The assumption is that Shrinp will do very poorly if the options are disabled and better when multiple options are used. The *i-det* mode should have a comparable result to *io-det*'s although being slightly faster. The conditions for the experiment will now be explained.

Because the number of possible combinations of the options gives a reasonably large number, 127, we decided to first test each option alone and then combined with some others. The

groups formed are shown in Table 7.7.

Table 7.7. Groups of options used in the experiment. Option "d" is active in all groups. (a = GMODE, b = GENEX, c = NEGUSER, e = BCNEG, f = RELEARNBC, g = BADARG)

Single-Option Groups	Options	Multiple-Option Groups	Options
I	none	VIII	bc
II	b	IX	be
III	ab	X	cg
IV	c	XI	beg
V	e	XII	abceg
VI	f	XIII	abeg
VII	g	XIV	abg

In Group I of Table 7.7, Shrinp is tested without any active options. Groups II through VII test the effect of using each option individually. Groups II and III test GENEX with *i-det* and *io-det* modes, respectively. The other groups combine two or more options. Because of the importance and harmlessness of BADARG and GENEX, "b" and "g" are repeated in nearly all multiple-option groups. Shrinp's default configuration is also tested in this experiment (Group XIII). Although not shown, option "d" is active in all groups but Group I as it is necessary to run Shrinp in order to learn RRDs and LRDs. The value of Shrinp's parameters was kept as the default (see Section 5.6.3).

For simplicity, we chose four relations that require the invention of a new predicate to take part in this experiment so we can test all the options: *doubles*, *permut*, *reverse* and *sum*. The sets of training examples used are the same ones portrayed in Table 7.2. The fourteen groups were run ten times for each relation. The runtimes were averaged and the number of correct definitions tested against a testing set of 100 randomly generated examples, approximately 50% of which was positive.

Table 7.8 shows the results with respect to the single-option groups. Let $t(d,p)$ be the time taken by Shrinp to learn a predicate p when only the default options are enabled. To allow a better comparison, we have made all the runtimes for p relative to $t(d,p)$ by dividing each runtime

by *t(d,p)* and multiplying the result by 100. This way, those values higher than 100 mean a greater runtime than that for Shrinp with the defaults. For instance, the table shows that, by using the options in Group III, Shrinp was able to learn a definition of *doubles* alone in almost the same time that it would if the default options were enabled. In the table, "Acc" corresponds to the accuracy, "C" indicates whether a correct definition was induced or not and "A", whether it was induced alone. If a definition is found that leads to an infinite recursion, "*loop*" is shown.

Table 7.8. Accuracy, runtime and correctness of the use of single options.
(a = GMODE, b = GENEX, c = NEGUSER, e = BCNEG, f = RELEARNBC, g = BADARG)

Group	Options	*doubles*	*permut*	*reverse*	*sum*
		Acc, Time, C, A	**Acc, Time, C, A**	**Acc, Time, C, A**	**Acc, Time,C, A**
I	none	100%, 176, Y, N	loop	loop	loop
II	b	100%, 117, Y, Y	100%, 78, Y, Y	100%, 93, Y, Y	95%, 91, Y, N
III	ab	100%, 101, Y, Y	100%, 71, Y, Y	100%, 80, Y, Y	loop
IV	c	100%, 91, Y, N	76%, 55, Y, N	100%, 56, Y, Y	loop
V	e	100%, 129, Y, N	100%, 240, Y, Y	100%, 131, Y, Y	loop
VI	f	100%, 181, Y, N	loop	loop	loop
VII	g	100%, 155, Y, Y	loop	loop	91%, 110, Y, N

In the first row, which corresponds to the execution of Shrinp when all options are disabled (with natural exception of option "d", of course), a loop occurred for three of the relations. The infinite loops are explained by the fact that either a definition was induced that has a bad argument (hence, its coverage test never stopped), which is what happened to the relation *sum*, or because the given negative examples were not enough to discard incorrect definitions (*permut* and *reverse*). Without the proper means to reject such wrong definitions, Shrinp was unable to output a correct response and ended up failing.

There are incorrect definitions that can only be discarded by the use of one option while other definitions can only be filtered out by the use of another option. For all four predicates, the activation of RELEARNBC alone had no effect on the result obtained (Group VI) as it is comparable to the first row with a slight increase in the runtime. This is expected since none of

these predicates require the base clause to be relearned and no other option is set. On the other hand, the option BCNEG alone was enough for a correct definition to be learned alone for *permut* and *reverse*, but at the expense of a great increase in the CPU time (Group V). In Group VII we can notice BADARG's strength in the induction of a correct definition alone for *doubles* by discarding another correct definition that had a useless argument, which was not possible in Groups I, IV, V and VI. BADARG also contributed to the removal of the infinite recursion from *sum*, but was not enough for the other two relations to be learned. Only the use of GENEX alone had a better outcome (Groups II and III). The utility of GENEX was once again attested by the fact that the wrong definitions were cast off early in the execution. Observe, anyway, that no single option was sufficient for the relation *sum* to be learned alone. A combination of the options is, thus, required for a successful learning. That is what is conveyed by the next table, where "*error*" means that no definition was induced.

Table 7.9. Accuracy, runtime and correctness of the use of multiple options.
(a = GMODE, b = GENEX, c = NEGUSER, e = BCNEG, f = RELEARNBC, g = BADARG)

		doubles	*permut*	*reverse*	*sum*
Group	**Options**	**Acc, Time, C, A**	**Acc, Time, C, A**	**Acc, Time, C, A**	**Acc, Time,C, A**
VIII	bc	100%, 131, Y, Y	100%, 87, Y, Y	100%, 96, Y, Y	error
IX	be	100%, 119, Y, Y	100%, 100, Y, Y	100%, 115, Y, Y	100%, 100, Y,Y
X	cg	100%, 87, Y, Y	76%, 55, Y, N	100%, 56, Y, Y	error
XI	beg	100%, 120, Y, Y	100%, 102, Y, Y	100%, 113, Y, Y	100%, 101, Y,Y
XII	abceg	100%, 112, Y, Y	100%, 102, Y, Y	100%, 109, Y, Y	error
XIII	abeg	100%, 100, Y, Y	100%, 100, Y, Y	100%, 100, Y, Y	100%, 100, Y,Y
XIV	abg	100%, 97, Y, Y	100%, 70, Y, Y	100%, 79, Y, Y	95%, 82, Y, N

Table 7.9 shows a quite remarkable improvement over the previous table. Almost all combinations of options resulted in the induction of a correct definition of the target predicate alone with an accuracy of 100%. For *doubles*, *permut* and *reverse*, the lowest runtimes were obtained from the use of the options *c* and *g* together (NEGUSER and BCNEG, Group X). In this case, the check for definitions with bad arguments and the use of the given negative examples to find the examples of a new predicate were enough to discard incorrect definitions early in the

execution, thereby saving considerable time in not testing them and not searching for a great number of new predicates. But note that *permut* was not learned alone while *sum* failed.

Shrinp's default options in Group XIII are indeed very effective. A correct definition was induced alone in all cases. At the same time they offer a reasonably low runtime if compared to other groups that also induced correct definitions alone. Only Groups IV and XIV offered a better runtime for all the relations, but even so with an erroneous output for *sum*.

The setbacks were only a few. The error that is caused by the use of NEGUSER can be visualized here (Groups VIII, X and XII but see also Group IV in Table 7.8). As explained in Section 6.5.2, the activation of this option can be harmful when the learned negative examples are actually positive instances of the new predicate. This fact was the reason why all combinations that involved the use of option "c" for the relation *sum* resulted in an error. The correct definition found for the new predicate *plus* covered the wrongly computed negative examples and was, consequently, discarded.

Note how the use of GENEX's *i-det* mode as opposed to the *io-det* mode results in a better runtime for all relations: the value for Groups III and XIII are smaller than those for Groups II and XI, respectively. The reason for this is because more time is spent on generating the extra negative examples that comply with the *io-det* mode and also on testing the coverage of each definition on these examples. The experiment showed what was expected: the *i-det* mode is faster but the *io-det* mode, although less reliable, can filter out more wrong clauses.

7.4 Experiments with Random Examples

So far we have been testing Shrinp with the hand-crafted examples of Table 7.1 only. This time, we want to see the performance of the system when randomly generated examples are employed. We want to measure how many times a correct definition is induced and if it is learned alone. We also want to check the accuracy of the induced definitions. The details of the experiment will now be explained:

- The number of positive examples, which is supposed to be small, will vary in the following manner: 2, 3, 4, 5 and 6.

- The lists will be one-level deep and will have at most 3 elements taken with repetition from the set {0,1,2,3} or {a,b,c,d} depending on whether the relation needs a numeric element or not.
- The number of negatives will be fixed at a large number, 15, so that wrong definitions do not greatly influence the result of the learning process.
- The testing set will contain 100 randomly generated examples, approximately 50% of which will be positives. The lists will have a maximum length of four, the maximum natural number will also be four and the maximum letter will be *d* (from *a* to *d*).
- The accuracy will be measured as the percentage of positive testing examples covered plus the percentage of negative testing examples not covered by a definition.

We will use the same four predicates employed in the experiments reported in Section 7.3: *doubles*, *permut*, *reverse* and *sum*. The reason for that is because they all require the invention of a new predicate so we can check the number of times this is done.

For each predicate, this is what will be done. For each number of training positives (2, 3, 4, 5 and 6), Shrinp will be run 10 times. Each execution will use a different set of random positive examples. The average of the accuracies and the total number of correct definitions induced in all ten runs will be recorded. Shrinp's default options will be used. We expect that Shrinp will not find a correct definition for the given predicates in all runs, but we definitely foresee that the accuracy will increase with the increase of the size of the training set.

The results of this experiment are presented in Table 7.10. It shows the average accuracy of the induced definitions ("Acc"), the CPU time, the number of times a correct definition was induced ("C"), the number of times it was induced alone ("A") and the number of times a predicate was invented ("PI"), whether or not the learned definition was 100% accurate.

It is clear that the accuracy improves for all four predicates with an increase of the number of positive examples. The runtime also increases especially because more subterms will take part in the cross-product to find the induced base clauses.

Table 7.10. Accuracy, correctness and runtime on different numbers of positive examples.

#	*Doubles*		*Permut*		*Reverse*		*Sum*	
+	Acc	Time,C,A,PI	Acc	Time,C,A,PI	Acc	Time,C,A,PI	Acc	Time,C,A,PI
2	63%	601, 1, 1, 1	55%	873, 0, 0, 0	57%	427, 0, 0, 0	60%	426, 0, 0, 0
3	71%	7082, 1, 1, 2	67%	1763, 0, 0, 0	74%	881, 0, 0, 0	60%	367, 0, 0, 0
4	81%	853, 1, 1, 1	70%	13758, 0, 0, 1	95%	1552, 0, 0, 0	71%	262, 0, 0, 0
5	82%	9403, 0, 0, 0	75%	37663, 0, 0, 0	95%	9374, 0, 0, 0	71%	1152, 0, 0, 0
6	82%	10249, 0, 0, 0	78%	59344, 0, 0, 0	95%	9794, 0, 0, 0	71%	1416, 0, 0, 0

For small numbers of positives, some of the induced definitions are too specific and are not consistent with the training set, which forces the system to search for a new predicate. This explains why the runtime increased so much for *doubles* when three examples were used. A new predicate was invented twice in the ten runs and a correct definition was found only once.

Nevertheless, as more examples are given to Shrinp, there is a greater chance that it will find a PRD that is consistent with all the training set. For *reverse* and *sum*, the same definition was induced in all ten runs when 4, 5 and 6 examples were used. Likewise, the majority of the definitions found for the relation *permut* for 5 and 6 positives were the same. Shrinp also induced the same definition for *doubles* in all runs for 5 and 6 examples (see D2 below).

```
doubles([],[]).                                    (D2)
doubles([s(A)|B],[s(s(C)),D]) :-
    doubles(B,D).
```

Note that D2 covers *all* examples in the training set. That, of course, prevents Shrinp from inventing new predicates. Although such PRDs have a high accuracy on unseen data, yet they cannot be considered correct because they do not completely and consistently cover the testing set.

For the execution with 6 examples, the runtime was very high for *doubles*, *permut* and *reverse*, but not for *sum*. This is because the system generated less base clauses for *sum* (an average of 243) than for the others (an average of 722, 8924 and 708, respectively). Needless to say, the larger the number of base clauses, the greater the number of definitions to be tested and

possibly completed with a call to a new predicate. Hence, the runtime easily sky-rockets.

When *doubles* was executed with 3 examples, a new predicate was invented twice. The time spent on searching for the examples and a definition of the new predicates made the runtime grow to an average of over 7 seconds. Only one of these two definitions was considered correct.

As a result of this experiment, we can see that the system does not perform very well on random examples and spends more time as the number of examples increases. That is a consequence of both the constraints described in Section 3.5.2.1 and the induction of PRDs that are general enough to pass a coverage test of the training set.

7.5 Experiments with CLAM

[Rios, Matwin 96b] shows two experiments that were carried out as an evaluation of CLAM: a learning curve and a comparison with Golem and Progol. We now reproduce the results. The hypotheses are that:

- CLAM's accuracy increases with the number of positive examples. The more positive examples are presented to CLAM, the higher the chance of finding two of them with common subterms, and the larger the number of incorrect clauses discarded.
- On small training sets, CLAM should outperform Golem and Progol's accuracy. CLAM was designed to specifically learn from a small data set while the other two are general-purpose learners that require more data.

Four predicates that can be defined left-recursively were chosen, as summarized in Table 7.11. The background knowledge was rewritten extensionally so as to conform to the requirements of the other two systems used in the comparison. The accuracy was obtained from a randomly selected testing set of 100 examples, approximately 50% of which is positive. By accuracy it is meant the percentage of positive examples covered and negative examples not covered by a definition.

Table 7.11. A minimal training set for testing CLAM on four predicates.

Definition induced	Background Knowledge	Positive examples	Negative examples
allatoms([]). allatoms([A,B]) :- allatoms(B), atom(A).	none	allatoms([a]). allatoms([c,d]).	allatoms([[a]]). allatoms(a).
doubles([],[]). doubles([A,B],C) :- doubles(B,D), double(A,E), make_pair(E,D,C).	plus(0,A,A). plus(s(A),B,s(C)) :- plus(A,B,C). doubles(X,Y) :- plus(X,X,Y). make_pair(A,B,[A,B]).	doubles([0], [0]). doubles([s^1,s^1],[s^2,s^2]).	doubles([], [0]). doubles(0, 0). doubles([s^2],[s^2]).
sum([],0). sum([A,B],C):- sum(B,D), plus(D,A,C).	plus(0,A,A). plus(s(A),B,s(C)) :- plus(A,B,C).	sum([s^2,0], s^2). sum([s^1, s^1], s^2).	sum([s^1,0], 0). sum([s^2], s^1). sum([], s^2).
ureverse([],[]). ureverse([A,B],C) :- ureverse(B,D), make_pair(A,E), append(D,E,C).	append([],A,B) :- var(A), var(B), A==B. append([],A,A). append([A,B],C,[A,D]):- append(B,C,D), !. make_pair(A, [A]).	ureverse([a,b], [b,a]). ureverse([x]),[x]).	ureverse([x], [y]). ureverse([], [a]). ureverse(a,a).

7.5.1 Comparison with Golem and Progol

In this experiment, each system was given ten randomly selected training sets with five positive and ten negative examples. The final induced clauses, when not leading to an infinite recursion, were executed with the testing set described earlier and the results averaged and recorded in Fig. 7.3.

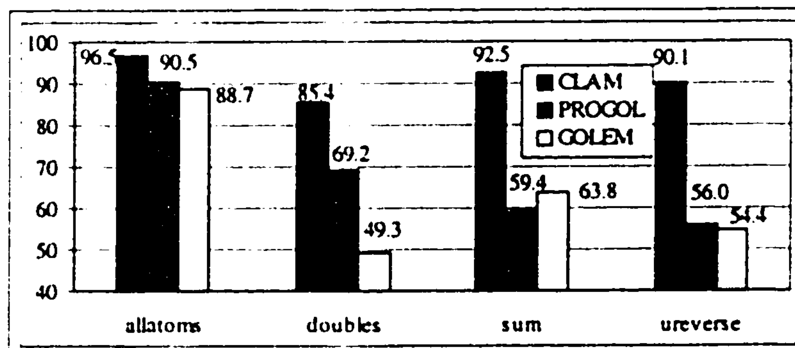


Fig 7.3. CLAM vs. Golem vs. Progol. Accuracy with five positive and ten negative examples.

Two points are worth making. The first is that it comes as no surprise that CLAM outperformed both systems as they require a higher number of positive examples to learn these

predicates. The second is that, as discussed before (Section 7.2.1), both Golem and Progol return all or some of the positive examples as the induced clauses when they can not learn an intensional definition. Of course, such ground clauses do not cover any negative example, which gives those systems a minimum accuracy of roughly 50%. That is why they got such a good accuracy on *allatoms* without ever inducing the correct clause (cf. Table 7.12). Another statistics to support this claim is the number of times a definition with at least one non-ground clause was induced by each system. While CLAM always induces a non-ground clause with high accuracy, Progol sometimes induces no non-ground clauses at all, whereas Golem induces non-ground definitions with poor predictive power.

Table 7.12. Number of times a definition was induced by each learner out of ten runs.

	Correct definitions			Non-ground definitions		
Predicates	**CLAM**	**Progol**	**Golem**	**CLAM**	**Progol**	**Golem**
allatoms	10	0	0	10	10	10
doubles	6	0	0	10	0	5
sum	9	0	0	10	10	4
ureverse	7	0	0	10	7	7

7.5.2 Learning Curve

In order to obtain a learning curve for CLAM, the number of positive examples was varied from two to nine, while the number of negative examples was set to ten. The length of a list was set to be at most three and the greatest natural number to two. The hypothesis that the accuracy should increase with the number of positive examples was confirmed. In Fig. 7.4, four positive examples were enough to obtain the maximum accuracy for *allatoms* and eight for *sum*. *Ureverse* stabilizes on 96.4% with six examples, while *doubles* goes up to 95.0% with nine.

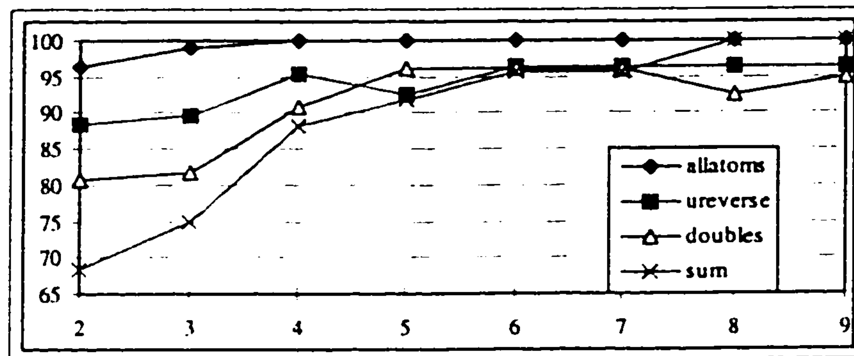


Fig 7.4. Learning curve for CLAM.

7.6 Discussion

The experiments reported in this chapter are a proof that the method being presented in this work is indeed effective. Shrinp can learn various different classes of recursive definitions from a small number of examples. No input is necessary other than the modes and the examples for the target relation. The system offers a range of options that can guarantee the success of a learning session at a reasonably low runtime as compared to the execution of Shrinp without them.

Experiments also showed how effective the use of GENEX can be. Far from being *the* definitive replacement of negative examples, the GENEX algorithm still provides Shrinp with means to learn a correct solution for most of the predicates at a low runtime. This is true even when the questionable *io-det* mode is not active. The mere fact that learning is possible in the complete absence of user-given negative instances serves an a solid evidence that the technique is valid. Two other systems reported much better results when using only the negative examples from our algorithm (and adequate positive instances) than in the complete absence of user-provided negatives. Golem and Progol were able to correctly induce a definition for several of the predicates shown in Fig. 7.1. The combination of the *type* mode with either the *i-det* or the *io-det* modes provided a strong way to discard incorrect definitions, thereby enhancing the systems' outcome. The results urge ILP systems to incorporate a similar algorithm for creating its own negative evidence in order to become stronger. Nevertheless, it has to be noted that the discussion in Chapter 5 showed that these methods may also hamper some relations from being learned.

Shrinp's predicate invention method was shown to work in this chapter. Table 7.3

showed seven relations that were learned together with the supporting predicates they need. For one of them, a rather curious output was produced. The usual definitions of *reverse* contain a call to the predicate *append* which is defined in the usual manner (see the definition of *app* in the same table). However, because the definition that Shrinp induced for *reverse* did not pass the first argument of the invented predicate as a list, the base clause for the new predicate was automatically modified by Shrinp in order to return a list. In addition, the experiments also show how the argument-binding chains discussed in Sections 6.3 and 6.4 are really effective in providing Shrinp with means to learn several kinds of recursive definitions.

The experiment with the options showed how the outcome of a learning session can vary according with the choices made. Shrinp's default options were seen to be strong with a reasonable runtime if compared to the use of other combinations. Even so, the correctness of the induced definitions compensates for the slightly slower execution time.

Shrinp does not perform very well with random data. The experiment showed that although the accuracy increases with the number of positive examples, so does the runtime. Even so, a correct definition is hardly found because the system usually comes up with a purely recursive definition that is consistent with the given examples. A human-provided set of negative examples could prevent such PRDs from being learned, thereby forcing the system to search for a new predicate. See, for instance, definition D2 in Section 7.4 which could be discarded with a carefully chosen negative example like $-doubles([s^2,s^3],[s^4,0])$

Finally, in a comparison with general-purpose learners, the system CLAM got a better accuracy on unseen data. A learning curve for CLAM shows that the accuracy improves with the increase of the small number of positive examples.

chapter **eight**

8. Conclusion

Inductive Logic Programming systems that are specially designed to learn recursive definitions of a single predicate cannot compete with large general-purpose learners. Their scope is more limited. But they can help their bigger counterparts in small tasks that involve learning a specific intermediate recursive relation because they have better means to deal with the intricacies of recursive definitions. It is almost like a simple sorting algorithm that takes over from a more complex sorting method when the original array to be sorted is broken down into a small, more manageable sub-array. In general, not many examples of the intermediate predicates are available, so it is essential that such dedicated learners be able to work with few examples. This ability is convenient for inventing a new predicate because of the likely scarcity of examples of the new relation too. As a result of using specialized learners, such small tasks can be done faster, more effectively and more reliably.

We saw in this work that the ILP task can be seen as a problem of generalization and that implication is the natural operation to be inverted in order to find a program that is more general than a set of examples, even though it lacks decidability. The development of an algorithm to compute an lgg under implication between two clauses (see Appendix A) makes this operation more attractive to research.

The main purpose of this project was to develop a new method based on inverse implication that is capable of inventing recursive predicates. The method was implemented in the system Shrinp with successful results. Shrinp was designed to learn recursive definitions of a single predicate and to operate with few examples. These characteristics make it the first inverse implication-based constructive learner. The principal features of the system are:

a) Predicate invention;

b) The ability to learn without the need for background knowledge;

c) The ability to learn from a few sparse examples (i.e. in non-intersecting resolution paths);

d) The ability to learn purely recursive, left- and right-recursive linear clauses;

e) The use of argument-binding chains to guide the search for the arguments of newly invented predicates;

f) The ability to relearn the base clause;

g) No need of any form of external help from the user, be it domain descriptions, an oracle or a type system. Only the training examples and the modes of the target relation are necessary.

Besides the development and the implementation of the method, the main contributions of this research are shown below:

a) A comparison between several ILP systems;

b) The development of an algorithm to automatically generate negative examples;

c) The implementation of a method to eliminate incorrect base clauses by using other potential base clauses;

d) The development and implementation of the system CLAM;

e) The development of a method to relearn the base clause;

f) A compilation of methods that learn from positive examples only;

g) The development of clause filters;

h) The development of argument-binding chains; and

i) Extensions to the method, as shown in the next section.

The applications foreseen for the system include working as:

a. A specialized learner; and as

b. A predicate discovery tool, in which Shrinp would be used by Prolog students as an interactive system that would give them the opportunity to play with examples and with the definitions they imply.

Shrinp is divided into three main modules: the inducer, the filter and the predicate inventor. The inducer accepts the examples from the user, induces potential definitions and passes them onto the Filter Module. This module checks their correctness and calls the Predicate Inventor Module if a literal is found to be missing for a definition to be complete. The idea is to have the predicate inventor call Shrinp recursively in order to find the definition of a new predicate. The recursive reuse of Shrinp to learn a new definition while still searching for a definition of the target relation brings serious problems of housekeeping as several definitions of varying depths are passed back and forth between different levels of recursive calls. Solutions to implementation problems such as this are discussed in Chapter 5.

This work emphasized the importance of the automatic generation of negative examples. The approach includes the use of base clauses as negative instances to discard incorrect base clauses, the corruption of positive examples, and the utilization of the negative examples of the target relation to find negative instances of the new predicate. A lengthy discussion showed that at the same time that these methods allow some definitions to be induced alone faster, they may also hamper other relations from being learned. It seems that it is not always possible to safely produce negative evidence.

8.1 Extensions to the Method and Future Work

In this section we look into improving the method and suggest solutions to some of the raised issues.

Any program that contains recursive clauses runs the risk of non-termination due to infinite recursion. Currently, a simple check is done by the Filter Module in order to see if a definition leads to an infinite recursion (cf. Section 5.3.2.1). Although not guaranteeing that all learned clauses will finish, the test is at least very efficient. Other alternatives could be explored like the method described in [Cameron-Jones, Quinlan 93], which is based on ordering the recursive literals in a clause. To implement such an idea in Shrinp, however, a type system, which brings the disadvantages discussed in Section 5.1, would have to be used.

A study published in the Journal of AI Research ([Cohen 95b]) on the learnability of recursive logic programs suggests that to learn complex recursive hypotheses, some information must be given in addition to the training set. This may be achieved either through the knowledge of some special properties of the target concepts or with the availability of the complete sets of examples. Our argument-binding chains and Shrinp's language bias represent a step in this direction especially when only a few instances are available. Further work on Shrinp could focus on developing new bindings that may account for the definition of relations not yet learnable.

Currently, the definitions that are returned by the Left and Right Inventors are taken in the given order with the RRDs first. Heuristics could be developed to order the definitions in such a way that the most likely to be the correct one comes first. An evaluation function could give preference, for instance, to definitions that have the least number of arguments in the new predicate, cover the highest number of positive examples, cover the lowest number of negative examples etc.

Shrinp's hypothesis language has a strong representational bias. Apart from purely recursive definitions, only three-literal recursive clauses and one unique base clause are allowed in left- and right-recursive definitions. Future system developers should focus on relaxing some of these limitations. Several base clauses can be generated if the positive examples are analyzed in groups. Incorrect base clauses could be discarded in the usual manner, as described in Sections 3.5.2.1 and 5.4.3.

Given the unavailability of background knowledge, we believe it is hard to overcome the small size of the recursive clauses. However, although it would require a great computational

effort for Shrinp to add two or more new predicates to an incomplete clause, there still is room for introducing other recursive calls, which would at least make divide-and-conquer-based relations such as *quicksort* learnable. This idea, which allows for the induction of non-linear definitions, still deserves more study.

One way to assess the benefit of a new predicate to the current learning task is through the use of evaluation methods. In particular, compression measures are of interest due to their theoretical foundation, be they syntactically based techniques as described in [Srinivasan *et al.* 93] or semantically based techniques as described in [Muggleton, DeRaedt 94]. Our opinion is that there is a vast room for exploration in this direction.

A clever way to induce disjunctive definitions can be achieved if one takes into consideration the result of the coverage test. It is known that learning disjunctive definitions is a harder problem than learning two-clause definitions with a base and a recursive clause ([Cohen 95b]). The induction of a base clause means that a generating term was found that produced that clause (actually, there is a GT for each argument). For each base clause, therefore, there can be only one recursive clause. A disjunctive definition can be created if induced partial solutions are joined together (Fig. 8.1).

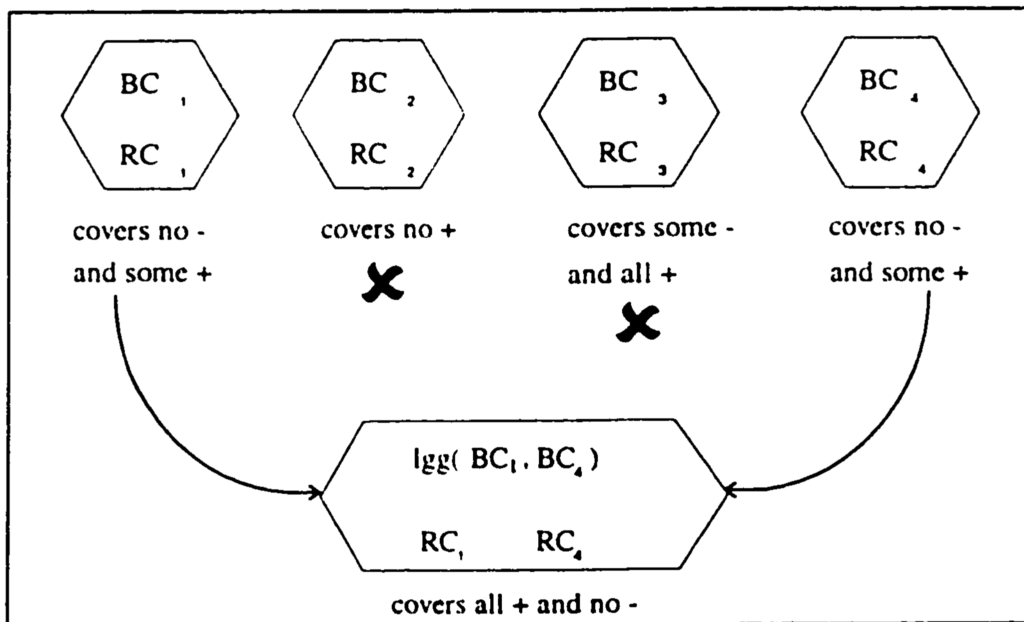


Fig 8.1. Learning a disjunctive definition.

A definition is considered a partial solution if it:

- makes it through the basic filters (see Section 5.3.2);
- is consistent with the negative examples, that is, covers none of them; and
- covers at least one but not ali positive examples.

To comply with the language bias and to be less specific, the base clauses of the partial solutions can be *lgged* so that the final definition will have only one base clause and more than one recursive clauses. By construction, the so-formed disjunctive definition will cover no negative instances, but still will have to be tested for positive coverage.

The above procedure, however efficient it may seem, is not general enough. Not only is it still limited to one base clause, it also spends considerable space and time testing and keeping track of the partial definitions. One idea that requires attention is the method proposed in [Stahl 91] for splitting the training set. She discusses how the splitting can be done in such a way that each subset has a high probability of being covered by only one clause. If Shrinp is to incorporate this technique, it still remains to be investigated how to combine the new definitions, especially with respect to their number, the coverage test and the multiplicity of base clauses.

The mechanism to select a subset of the positive examples to work with needs to be refined. Currently, the examples are picked at random. They could be grouped, say, by sub-unifiability, length, presence/absence of a constant/function or other strategy. What is important is that it be more likely to find the correct definition from the examples of a single group. Caution must be taken so that the selection process be not too time-consuming. Chao's system, introduced in Section 4.4, is a good starting point.

What would be the effect of using background knowledge together with the predicate invention facility? A hybrid system like SIERES (cf. Section 4.7) would be produced that could use the supporting relations and generate its own predicates whenever the background knowledge proved to be insufficient to learn a definition. As a consequence, the user would not have to worry too much about providing *the* necessary set of definitions for a given relation to be learned. A relatively small, structured general-purpose set would probably do. The number of programs that can be induced would be much larger in the sense that a call to non-recursive predicates could be inserted into the recursive clause. An alternative consists of letting the system

cumulatively keep the predicates it invents. This way, the restriction to obtain no further help from the user would still prevail through the use of such system-generated background knowledge. In any case, it would be a challenge to automatically simulate the human ability of dynamically organizing such knowledge according to the relevance to a problem (see [Horváth, Turán 96] for a discussion of learning with structured background knowledge).

The presence of background knowledge, on the other ledger, brings the concerns discussed earlier with respect to the *background knowledge usage bottleneck*. While a lack of the essential predicates may result in ineffective learning, an excess of (useless) predicates degrades the performance of the system. Questions regarding the choice and placement of the new predicates would result in a combinatorial explosion, not to mention the choice of arguments for the added literals from the background knowledge (see Section 4.9.3). Anyway, the system would also have to deal with incorrect or incomplete background knowledge, a problem discussed in [Mooney, Zelle 94]. Hybrid systems may solve a problem but create new ones.

We call *intensional elimination* of the search space the process of testing the correctness of an induced definition by syntactically analyzing the definition as opposed to running it on examples, which is known as an *extensional elimination*. Shrinp employs a combination of both approaches: incorrect definitions are first syntactically filtered out before a coverage test is performed. Given the scarcity of negative examples available to Shrinp, it is a good idea to bestow the system with a stronger form of intensional elimination. Integrity constraints, introduced in Section 4.10, can be used as negative evidence and represent an important step in this direction. [Sebag, Rouveirol 96] provides a thorough discussion on Constraint Logic Programming and also explains how negative instances can be turned into constraints.

On the practical side, new experiments can be devised. We envisage an experiment that compares Shrinp with other systems capable of constructive learning. The competence to learn the correct definition of necessary relations could be compared. Additionally, the accuracy of the solution on unseen examples and the runtime could also be recorded. Another experiment could test the system with respect to *noise* in the input data. We know that the input sets of examples are restricted in such a way that their intersections must be empty. How would the system react to overlapping sets of examples? Or to noise in the examples, i.e., if positive examples are taken as

negative examples and vice-versa? And how could the system be modified in order to deal with noise? These questions deserve attention.

Perhaps one of the major proposals for further work concerns the attachment of Shrinp to a general-purpose learner. As discussed earlier, the general-purpose system could be rewritten in order to call Shrinp whenever it determines that a simple recursive definition is to be induced. Experiments could be designed to check whether the execution of the new system has become any more efficient than what it was before.

Finally, we wish to study other means to extend the pruning of the hypothesis space and the generation of non-linear definitions. In addition, other criteria for stopping the recursive generation of new predicates are also to be considered. Recall that Shrinp was designed to learn from a small number of examples. When too many examples are given to Shrinp, the system slows down considerably, especially when *npos* equals zero (see Section 5.6.1.2). Means to deal with this situation still have to be devised.

chapter nine

9. References

Aha, D.W., Lapointe, S., Ling, C. & Matwin, S. (1994a). Inverting implication with small training sets. In *Proc. of the European Conference on Machine Learning* (31-48). Catania, Italy: Springer-Verlag

Aha, D.W., Lapointe, S., Ling, C. & Matwin, S. (1994b). Learning recursive relations with randomly selected small training sets. In *Proc. of the Eleventh International Conference on Machine Learning* (12-18). New Brunswick, NJ: Morgan Kaufmann

Aho, A., Sethi, R. & Ullman, J. (1986). *Compilers, Principles, Techniques and Tools*. Reading, MA: Addison-Wesley

Bain, M. & Muggleton, S. (1991). Non-monotonic learning. In (D. Michie, ed.) *Machine Intelligence* **12**. Oxford University Press

Banerji, R. (1992). Learning theoretical terms. In (S. Muggleton, ed.) *Inductive Logic Programming* (93-112). San Diego, CA: Academic Press

Bergadano, F., Giordana, A. & Ponsero, S. (1989). Deduction in top-down inductive learning. In *Proc. of the Sixth International Conference on Machine Learning* (23-25). Ithaca, NY: Morgan Kaufmann

Bergadano, F. & Gunetti, D. (1993). An interactive system to learn functional logic programs. In

Proc. of the Thirteenth International Joint Conference in Artificial Intelligence (1044-49). Chambéry, France: Morgan Kaufmann

Bergadano, F. & Gunetti, D. (1996). *Inductive Logic Programming: From Machine Learning to Software Engineering*. Cambridge, MA: The MIT Press

Cameron-Jones, R. & Quinlan, J. (1993). Avoiding pitfalls when learning recursive theories. In *Proc. of the Thirteenth International Joint Conference on Artificial Intelligence* (1050-5). Chambéry, France: Morgan Kaufmann

Casanova, M., Giorno, F. & Furtado, A. (1987). *Programação em Lógica e a Linguagem Prolog*. Rio de Janeiro, Brazil: Edgard Blücher *[In Portuguese]*

Chang, C.-L. & Lee, R. C.-T. (1973). *Symbolic Logic and Mechanical Theorem Proving*. New York, NY: Academic Press

Chao, L.J. (1994). *Using Argument Binding and Predicate Searching Schemes to Guide Partially Recursive Logic Programs Synthesis from Positive Examples*. Master's Thesis. National Tsing-Hua University, China

Cohen, W. (1993). Pac-learning a restricted class of recursive logic programs. In *Proc. of the Third International Workshop on Inductive Logic Programming, ILP 93*. Bled, Slovenia

Cohen, W. (1994). Rapid prototyping of ILP systems using explicit bias. In (T. Cohn, *ed.) Proc. of the Eleventh European Conference on Artificial Intelligence*. Amsterdam, the Netherlands

Cohen, W. (1995a). Pac-learning recursive logic programs: efficient algorithms. *Journal of Artificial Intelligence Research* **2**:501-539

Cohen, W. (1995b). Pac-learning recursive logic programs: negative results. *Journal of Artificial Intelligence Research* **2**:541-573

Colmerauer, A. & Roussel, P. (1993). The birth of Prolog. *HOPL II Preprints, SIGPLAN Notices* **28**(3):37-52

De Raedt, L. & Bruynooghe, M. (1993). A theory of clausal discovery. In *Proc. of the Thirteenth International Joint Conference on Artificial Intelligence* (1058-63). Chambéry, France:

Morgan Kaufmann

Deville, Y. & Lau K.-K. (1994). Logic program synthesis: a survey. *Journal of Logic Programming* **19**:321-350

Dolšak, N. & Muggleton, S. (1992). The application of Inductive Logic Programming to finite-element mesh design. In (S. Muggleton, ed.) *Inductive Logic Programming* (453-472). San Diego, CA: Academic Press

Dovey, M.J. (1995). Analysis of Rachmaninoff's piano performance using Inductive Logic Programming. In *Proc. of the Eighth European Conference on Machine Learning*. Heraclion, Crete, Greece

Drastal, G. & Raatz, S. (1989). Empirical results on learning an abstraction space. In *Proc. of the Eleventh International Conference on Artificial Intelligence*. Detroit, MI

Džeroski, S. & Bratko, I. (1996). Application of inductive logic programming. In (L. De Raedt, ed.) *Advances in Inductive Logic Programming* (65-81). Amsterdam, The Netherlands: IOS Press

Džeroski, S., Muggleton, S. & Russell, S. (1992). Pac-learnability of determinate logic programs. In *Proc. of the Fifth ACM Workshop on Computational Learning Theory*, Pittsburgh, PA: ACM

Feng, C. (1992). Inducing temporal fault diagnostic rules from a qualitative model. In (S. Muggleton, ed.) *Inductive Logic Programming* (473-493). San Diego, CA: Academic Press

Flach, P. (1993). Predicate invention in inductive data engineering. In (P. Brazdil, ed.) *Proc. of the European Conference on Machine Learning. Lecture Notes in Artificial Intelligence* Vol. 667 (83-94). Berlin, Germany: Springer-Verlag

Flener, P. (1995). Predicate invention in inductive program synthesis. *Technical Report BU-CEIS-9509*. Department of Computer Engineering and Information Science. Bilkent University. Ankara, Turkey

Flener, P. & Popelínský, L. (1994). On the use of inductive reasoning in program synthesis:

prejudice and prospects. In (L. Fribourg & F. Turini, eds.) *Joint Proc. of the Fourth International Workshops on Meta-Programming in Logic and Logic Program Synthesis and Transformation* (69-87). LNCS vol. 883. Pisa, Italy: Springer-Verlag

Flener, P. & Yilmaz, S. (1997). Inductive synthesis of recursive logic programs: achievements and prospects. Submitted to *Journal of Logic Programming*

Furukawa, K., Murakami, T., Ueno, K., Ozaki, T. & Shimazu, K. (1997). On a sufficient condition for the existence of most specific hypothesis in Progol. In *Proc. of the Seventh International Workshop on Inductive Logic Programming*. Prague, Czech Republic

Gallier, J. (1986). *Logic for computer science: foundations of automatic theorem proving*. New York, NY: Harper & Row

Genesereth, M. & Nilsson, N. (1987). *Logical Foundations of Artificial Intelligence*, Morgan Kaufmann

Gottlob, G. (1987). Subsumption and implication. *Information Processing Letters* **24**(2):109-111. North-Holland

Horváth, T. & Turán, G. (1996). Learning logic programs with structured background knowledge. In (L. De Raedt, ed.) *Advances in Inductive Logic Programming* (172-191). Amsterdam, The Netherlands: IOS Press

Idestam-Almquist, P. (1993a). *Generalization of Clauses*. Ph.D. Thesis. Dept. of Computer and System Sciences. Stockholm University. Stockholm, Sweden

Idestam-Almquist, P. (1993b). Generalization under implication by using or-introduction. In *Proc. of the European Conference on Machine Learning*. Vienna, Austria: Springer-Verlag

Idestam-Almquist, P. (1995). Generalization of clauses under implication. *Journal of Artificial Intelligence Research*, 3:467-489

Idestam-Almquist, P. (1996). Efficient induction of recursive definitions by structural analysis of saturation. In (L. De Raedt, ed.) *Advances in Inductive Logic Programming* (192-205). Amsterdam, The Netherlands: IOS Press

Jorge, A. & Brazdil, P. (1995). Architecture for iterative learning of recursive definitions. In *Proc. of the Fifth International Workshop on ILP*. Leuven, Belgium

Jorge, A. & Brazdil, P. (1996). Integrity constraints in ILP using a Monte Carlo approach. In (S. Muggleton, ed.) *Proc. of the Sixth International Workshop on Inductive Logic Programming*. Stockholm, Sweden: Springer-Verlag

Kietz, J.U. (1993). Some lower bounds for the computational complexity of inductive logic programming. In (P. Brazdil, ed.) *Proc. of the Sixth European Conference on Machine Learning* (115-123). Berlin, Germany: Springer-Verlag

Kietz, J.U. & Wrobel, S. (1992). Controlling the complexity of learning in logic through syntactic and task-oriented models. In (S. Muggleton, ed.) *Inductive Logic Programming* (335-359). San Diego, CA: Academic Press

Kijsirikul, B., Numao, M. & Shimura, M. (1991). Efficient learning of logic programs with non-determinate, non-discriminating literals. In *Proc. of the Eighth International Workshop on Machine Learning* (417-421). Evanston, IL: Morgan Kaufmann

Kijsirikul, B., Numao, M. & Shimura, M. (1992). Discrimination-based constructive induction of logic programs. In *Proc. of the Tenth National Conference on Artificial Intelligence* (44-49). San José, CA: AAAI Press

Kirschenbaum. M. & Sterling, S. (1991). Refinement strategies for inductive learning of simple Prolog programs. In (J. Mylopoulos, R. Reiter, eds.) *Proc. of the Twelfth International Joint Conference on Artificial Intelligence*. Sydney, Australia: Morgan Kaufmann

Kleene, S. (1952). Finite axiomatizability of theories in the predicate calculus using additional predicate symbols. In *Two Papers on the Predicate Calculus*. Memoirs of the American Mathematical Society **10**. Providence, RI: American Mathematical Society

Koza, J. (1992). *Genetic Programming: On the Programming of Computers by Means of Natural Selection*. Cambridge, MA: The MIT Press

Koza, J. (1994). *Genetic Programming II: Automatic Discovery of Reusable Programs*. Cambridge, MA: The MIT Press

Koza, J. (1996a). Future work and practical applications of genetic programming. *Handbook of Evolutionary Computation*

Koza, J. (1996b). On cross-paradigm comparisons of genetic programming with existing machine learning paradigms. Available at *http://www-cs-staff.stanford.edu/~koza/Cross-Para-8-17-95.html*

Kowalski, R. (1972). The predicate calculus as a programming language. In *Proc. of the International Symposium and Summer School on Mathematical Foundations of Computer Science*. Jablona, Poland

Kowalski, R. (1974). Predicate logic as a programming language. In *Proc. of IFIP'74 World Congress* (689-574). Stockholm, Sweden: North-Holland

Kowalski, R. (1979). *Logic for Problem Solving*. New York: Elsevier North-Holland

Laag, P. & Nienhuys-Cheng (1993). Subsumption and refinement in model inference. In (P. Brazdil, ed.) *Proc. of the Sixth European Conference on Machine Learning. Lecture Notes in Artificial Intelligence* Vol. 667 (95-114). Vienna, Austria: Springer-Verlag

Lakhotia, A. (1989). Incorporating programming techniques into Prolog programs. In (Lusk and Overbeek, eds.) *Proc. of the North-American Conference on Logic Programming* (426-440). Cambridge, MA: MIT Press

Lapointe, S. (1992). *Induction of Recursive Logic Programs*. Master's Thesis. University of Ottawa. Ottawa, Canada

Lapointe, S., Ling, C. & Matwin, S. (1993). Constructive inductive logic programming. In *Proc. of the Thirteenth International Joint Conference on Artificial Intelligence* (273-281). Chambéry, France: Morgan Kaufmann

Lapointe, S. & Matwin, S. (1992). Sub-unification: A tool for efficient induction of recursive programs. In *Proc. of the Ninth International Conference on Machine Learning* (273-281). Aberdeen, Scotland: Morgan Kaufmann

Lassez, J.-L., Maher, M.J. & Marriot, K. (1988). Unification revisited. In (J. Minker, ed.)

Foundations of Deductive Databases and Logic Programs (587-626). Los Altos, CA: Morgan Kaufmann

Lavrač, N & Džeroski, S. (1994). *Inductive Logic Programming: Techniques and Applications*, Ellis Horwood

Lavrač, N., Džeroski, S. & Grobelnik, M. (1991). Learning non-recursive definitions of relations with LINUS. In *Proc. Fifth European Working Session on Learning* (265-281). Berlin, Germany: Springer-Verlag

LeBlanc, P. (1994). BMWk revisited. Generalization and formalization of an algorithm for detecting recursive relations in term sequences. In *Proc. of the European Conference on Machine Learning* (183-198). Berlin, Germany: Springer-Verlag

Lifschitz, V. (1987). Closed-world databases and circumscription. In (M. Ginsberg, ed.). *Readings in Nonmonotonic Reasoning* (334-336). Los Altos, CA: Morgan Kaufmann

Ling, C. (1991a). Inductive learning from good examples. In (J. Mylopoulos, R. Reiter, eds.) *Proc. of the Twelfth International Joint Conference in Artificial Intelligence* (751-756). Sydney, Australia: Morgan Kaufmann

Ling, C. (1991b). Inventing necessary theoretical terms. In *Proc. of the IJCAI-91 Workshop on Evaluating and Changing Representation in Machine Learning*. Sidney, Australia: Morgan Kaufmann

Ling, C. & Narayan, M. (1991). A critical comparison of various methods based on inverse resolution. In *Proc. of the Eighth International Workshop on Machine Learning* (168-172). Evanston, IL: Morgan Kaufmann

Lloyd, J. W. (1987). *Foundations of Logic Programming*, 2nd ed. Berlin, Germany: Springer-Verlag

Loveland, D. (1970). A linear format for resolution. In *Proc. of IRIA Symposium on Automatic Demonstration* (147-162). Versailles, France: Springer-Verlag

Loveland, D. (1978). *Automated Theorem Proving: A Logical Basis*. New York: North-Holland.

Luckham, D. (1970). Refinements in resolution theory. In *Proc. of IRIA Symposium on Automatic Demonstration* (163-190). Versailles, France: Springer-Verlag

Marcinkowski, J. & Pacholski, L. (1992). Undecidability of the Horn clause implication problem. In *Proc. of the 33rd Annual IEEE Symposium on the Foundations of Computer Science* (354-362). Pittsburgh, PA: IEEE Computer Science Press

Martin. L. & Vrain, C. (1997). Systematic predicate invention in inductive logic programming. In *Proc. of the Seventh International Workshop on Inductive Logic Programming*. Prague, Czech Republic

Matheus, C. & Rendell, L. (1989). Constructive induction on decision trees. In *Proc. of the Eleventh International Joint Conference in Artificial Intelligence* (645-650). Detroit, MI: Morgan Kaufmann

Mitchell, T. (1990). The need for biases in learning generalizations. In (J. Shavlink and T. Dietterich, eds.) *Readings in Machine Learning*. San Mateo, CA: Morgan Kaufmann

Mofizur, C. & Numao, C. (1995). Top-down induction of recursive programs from small number of sparse examples. In *Proc. of the Fifth International Workshop on Inductive Logic Programming* (161-180). Leuven, Belgium

Mofizur, C. & Numao, C. (1996). Top-down induction of recursive programs from small number of sparse examples. In (L. De Raedt, ed.) *Advances in Inductive Logic Programming* (236-253). Amsterdam, The Netherlands: IOS Press

Mooney, R.J. & Califf, M.E. (1995). Induction of first-order decision lists: Results on learning the past tense of English verbs. *Journal of Artificial Intelligence Research*, **3**:1-24

Mooney, R.J. & Zelle, J. (1994). Integrating ILP and EBL. *SIGART Bulletin* **5**:12-21. ACM Press

Morales, E. (1992). *First-order Induction of Patterns in Chess*. Ph.D. Thesis. The Turing Institute. University of Strathclyde. Glasgow, Scotland

Morales, E. (1997). PAL: A pattern-based first-order inductive system. Machine Learning **26** (2), Kluwer Academic.

Muggleton, S. (1987). DUCE, an oracle-based approach to constructive induction. In *Proc. of the Tenth International Joint Conference on Artificial Intelligence* (287-292). Milan, Italy: Morgan Kaufmann

Muggleton, S. (1990). Inductive Logic Programming. In *Proceedings of the First Conference on Algorithmic Learning Theory*. Tokyo, Japan: Ohmsha Publishers

Muggleton, S. (1991). Inverting the resolution principle. In *Machine Intelligence* **12**. Oxford University Press

Muggleton, S. (1992a). Inverting implication. In *Proc. of the Second Inductive Logic Programming Workshop*. Tokyo, Japan

Muggleton, S. (1992b). *Inductive Logic Programming*. San Diego, CA: Academic Press

Muggleton, S. (1993). Predicate invention and utility. *Journal of Experimental and Theoretical Artificial Intelligence* **6**(1):127-130

Muggleton, S. (1994). Inductive logic programming: derivations, successes and shortcomings. *SIGART Bulletin* **5**(1):5-11

Muggleton, S. (1995). Inverse entailment and Progol. *New-Generation Computing Journal* **13**:245-286

Muggleton, S. (1996). Learning from positive data. In *Proc. of the Sixth Inductive Logic Programming Workshop* (225-244). Stockholm, Sweden: Springer-Verlag

Muggleton, S. & Buntine, W. (1988). Machine invention of first-order predicates by inverse resolution. In *Proc. of the Fifth International Conference on Machine Learning* (339-352). Ann Arbor, MI: Morgan Kaufmann

Muggleton, S. & De Raedt, L. (1994). Inductive Logic Programming: Theory and methods. *Journal of Logic Programming* **19/20**:629-679

Muggleton, S. & Feng, C. (1990). Efficient induction of logic programs. In *Proc. of the First International Workshop on Algorithmic Learning Theory* (368-381). Tokyo, Japan: Ohmsha Publishers

Muggleton, S., King, R. & Sternberg, M. (1992). Protein secondary structure prediction using logic. In *Proc. of the Second International Workshop on Inductive Logic Programming.* Tokyo, Japan

Muggleton, S. & Page, D. (1994). Self saturation of definite clauses. In (S. Wrobel, ed.) *Proc. of the Fourth International Workshop on Inductive Logic Programming* (161-174). Bad Honnef/Bonn, Germany: GMD

Niblett, T. (1988). A study of generalisation in logic programs. In *Proc. of the Third European Working Session on Learning* (131-138). London, UK: Pitman

Nienhuys-Cheng, S.-H. & DeWolf, R. (1996a). Least generalizations under implication. In (S. Muggleton, ed.) *Proc. of the Sixth Inductive Logic Programming Workshop.* Stockholm, Sweden: Springer-Verlag

Nienhuys-Cheng, S.-H. & DeWolf, R. (1996b). Least generalizations and greatest specializations of sets of clauses. *Journal of Artificial Intelligence Research* **4**:341-363

Pagallo, G. & Hausler, D. (1990). Boolean feature discovery in empirical learning. *Machine Learning* **5**(1):71-99. Kluwer Academic

Pawlak, Z., Wong, S. & Ziarko, W. (1988). Rough sets: probabilistic versus deterministic approach. *International Journal of Man-Machine Studies* **11**:81-95

Pazzani, M. (1996). Review of 'Inductive Logic Programming: Techniques and Applications' by Nada Lavrač, Saso Džeroski. *Machine Learning* **23**(1):103-108, Kluwer Academic

Pereira, F. & Warren, W. (1980). Definite clause grammar for language analysis — a survey of the formalism and a comparison with augmented transition networks. *Artificial Intelligence* **13**:231-278

Plotkin, G.D. (1971). *Automatic Methods of Inductive Inference.* Ph.D. Thesis. Edinburgh University, Edinburgh, Scotland

Quinlan, J.R. (1986). Induction of decision trees. *Machine Learning* **1**(1):81-106, Kluwer Academic

Quinlan, J.R. (1990). Learning logical definitions from relations. *Machine Learning* 5:239-266, Kluwer Academic

Quinlan, J.R. (1991). Determinate literals in inductive logic programming. In *Proc. of the Twelfth International Joint Conference in Artificial Intelligence* (746-750). Sydney, Australia: Morgan Kaufmann

Quinlan, J.R. (1993). *C4.5: Programs for Machine Learning*. San Mateo: Morgan Kaufmann

Quinlan, J.R. (1994). Past tenses of verbs and first-order learning. In *Proc. of AI' 94 Seventh Australian Joint Conference on Artificial Intelligence* (13-20). Singapore: World Scientific, Armidale

Quinlan, J.R. (1996). Learning first-order definitions of functions. *Journal of Artificial Intelligence Research* 5:139-161

Quinlan, J.R. & Cameron-Jones, R. (1993). FOIL: a midterm report. In *Proc. of the European Conference on Machine Learning* (3-20). Vienna, Austria: Springer-Verlag

Reiter, R. (1978). On closed-world data bases. In (H. Gallaire & J. Minker, eds.) *Logic and Data Bases*. New York, NY: Plenum Press

Rios, R. (1989). *Raciocinando por Jogos Semânticos*. Master's Thesis. Pontifical Catholic University at Rio de Janeiro. Rio de Janeiro, Brazil *[In Portuguese]*

Rios, R. & Matwin, S. (1996a). Learning recursive definitions in Horn-clause logic. *Journal of the Brazilian Computer Society* vol 1, 3:29-39. Special Issue on Artificial Intelligence, Rio de Janeiro, Brazil

Rios, R. & Matwin, S. (1996b). Efficient induction of recursive Prolog definitions. In *Proc. of the Eleventh Canadian Artificial Intelligence Conference* (240-248). Toronto, ON: Springer-Verlag

Rios, R. & Matwin, S. (1998). Predicate invention from a few examples. Accepted for presentation at the *Twelfth Canadian Artificial Intelligence Conference*. Vancouver, BC, June

Robinson, J. (1965). A machine-oriented logic based on the resolution principle. *Journal of the*

ACM **12** (1), 23-41. ACM Press

Rouveirol, C. (1990). Saturation: postponing choices when inverting resolution. In (L. Aiello, ed.) *Proc. of the Ninth European Conference on Artificial Intelligence* (557-562). Pitman

Rouveirol, C. (1991). ITOU: Induction of first order theories. In (S. Muggleton, ed.) *Proc. of the First International Workshop on Inductive Logic Programming, ILP 91* (127-158). Viana de Castelo, Portugal

Rouveirol, C. (1992). Extension of inversion of resolution to theory completion. In (S. Muggleton, ed.) *Inductive Logic Programming* (63-92). San Diego, CA: Academic Press

Rouveirol, C. & Puget, J.F. (1989). A simple solution for inverting resolution. In *Proc. of the Fourth European Working Session on Learning* (201-211). Pitman

Rouveirol, C. & Puget, J.F. (1990). Beyond inversion of resolution. In *Proc. of the International Conference on Machine Learning* (122-130)

Sammut, C. & Banerji, R. (1986). Learning concepts by asking question. *Machine Learning: An Artificial Intelligence Approach* (167-192), vol. 2 (R. Michalski, J. Carbonell & R. Mitchell, eds.). Los Altos, CA: Morgan Kaufmann

Schlimmer, J. (1987). Learning and representation change. In *Proc. of the Sixth National Conference on AI* (511-515). Seattle, WA

Schmidt-Schauß, M. (1988). Implication of clauses is undecidable. *Theoretical Computer Science* **59**:287-296

Sebag, M. & Rouveirol, C. (1996). Constraint inductive logic programming. In (L. De Raedt, ed.) *Advances in Inductive Logic Programming* (277-294). Amsterdam, The Netherlands: IOS Press

Shapiro, E.Y. (1983). *Algorithmic Program Debugging*. Cambridge, MA: The MIT Press

Shapiro, S. (1992). *Encyclopedia of Artificial Intelligence*. 2nd ed. John Wiley & Sons

Siekman, J. & Wrightson, G. (1983). *Automation of Reasoning*. New York: Springer-Verlag

Silverstein, G. & Pazzani, M. J. (1991). Relational Clichés: Constraining constructive induction during relational learning. In *Proc. of the Eighth International Workshop on Machine Learning* (203-7). Evanston, IL: Morgan Kaufmann

Srinivasan, A., Muggleton, S. & Bain, M. (1993). The justification of logical theories based on data compression. *Machine Intelligence* **13**. Oxford University Press

Stahl, I. (1991). *Induktion von disjunktiven Konzepten*. Diplomarbeit. Unversity of Stuttgart, Germany *[In German]*

Stahl, I. (1995a). The efficiency of bias shift operations in ILP. In *Proc. of the Fifth International Workshop on Inductive Logic Programming*. Leuven, Belgium

Stahl, I. (1995b). The appropriateness of predicate invention as a bias shift operation in ILP. *Machine Learning* **20**:95-119. Kluwer Academic

Stahl, I. (1996). Predicate invention in inductive logic programming. In (L. De Raedt, ed.) *Advances in Inductive Logic Programming* (34-47). Amsterdam, The Netherlands: IOS Press

Stahl, I. & Weber, I. (1994). The arguments of newly invented predicates in ILP. In *Proc. of the Fourth International Workshop on Inductive Logic Programming*. Bad Honnef/Bonn, Germany: GMD

Sterling, L. & Shapiro, E. (1986). *The Art of Prolog*. Cambridge, MA: MIT Press

Tausend. B. (1994). Representing biases for inductive logic programming. In (F. Bergadano, L. DeRaedt, eds.) *Proc. of the European Conference on Machine Learning* (427-430). Berlin, Germany: Springer-Verlag

Utgoff, P. (1986). Shift of bias for inductive concept learning. *Machine Learning* **2**:107-148. Kluwer Academic

Valiant, L. (1984). A theory of the learnable. *Communications of the ACM* **27**(11). ACM Press

Wirth, R. (1989). Completing logic programs by inverse resolution. In *Proc. of the Fourth European Working Session on Learning* (239-250). Montpellier, France: Pitman

Wirth, R. & O'Rorke, P. (1992). Constraints for predicate invention. In (S. Muggleton, ed.) *Inductive Logic Programming* (299-318). San Diego, CA: Academic Press

Wrobel, S. (1994). Concept formation during interactive theory revision. *Machine Learning* **14**:169-191. Kluwer Academic

Yamamoto, A. (1996). Improving theories for inductive logic programming systems with ground reduced programs. *Technical Report AIDA-96-19*. FG Intellektik, FB Informatik. Technische Hochschule Darmstadt, Germany

Yamamoto, A. (1997). Which hypotheses can be found with inverse entailment. In *Proc. of the Seventh International Workshop on Inductive Logic Programming*. Prague, Czech Republic

Yardeni, E. & Shapiro, E. (1987). A type system for logic programs. (E. Shapiro, ed.) *Concurrent Prolog*, vol. 2. (210-244). Cambridge, MA: The MIT Press

Yardeni, E. & Shapiro, E. (1991). A type system for logic programs. *The Journal of Logic Programming* **10**:125-153

Zelle, J. & Mooney, R. (1994). Inducing deterministic Prolog parsers from treebanks: a machine learning approach. In *Proc. of the Twelfth National Conference on Artificial Intelligence*. Seattle, WA: The MIT Press

Appendix A

A. Computing an lgg under Implication

The following is the actual Prolog code that implements the computation of the least general generalization under implication of a list of terms. This is a modified version of the original code written by David Aha. The computation of an lgg under implication is defined in Section 2.5.

```
%=================================================================
%---- Computes least general generalization (lgg) of given expressions
%---- Modes: ++--
%---- Types: <List of List of terms, List of Env,
%----          List of List of terms, List of Env>
%=================================================================
lgg_list_of_list_of_terms([List_of_terms|T1],Envs,[Lgg|T2],Final_envs) :-
  lgg_list_of_terms(List_of_terms,Envs,Lgg,Next_envs),
  lgg_list_of_list_of_terms(T1,Next_envs,T2,Final_envs).
lgg_list_of_list_of_terms([],Envs,[],Envs).


%---- Computes lgg of the given list of terms according to given env
%---- Modes: ++--
%---- Types: <List of terms, List of Env, Term, List of Env>
lgg_list_of_terms(List_of_terms,Envs,Lgg,New_envs) :-
```

```
  has_simple(List_of_terms), !,
  lgg_list_of_some_simple(List_of_terms,Envs,Lgg,New_envs).
lgg_list_of_terms(List_of_terms,Envs,Lgg,New_envs) :-
  lgg_list_of_compound(List_of_terms,Envs,Lgg,New_envs).

%---- Computes lgg of the given list, some elements of which are simple
%---- Modes: ++--
%---- Types: <List of Terms, List of Env, Term, List of Env>
lgg_list_of_some_simple([H|R],Envs,H,Envs) :-% Identical terms
  all_the_same([H|R]).
lgg_list_of_some_simple([H|R],[Env|Rest_envs],V,[Env|Rest_envs]) :-
  binding(Env,H,V),                    % All bound to the same var
  all_have_same_binding(R,Rest_envs,V).
lgg_list_of_some_simple(Terms,Envs,X,New_envs) :-
  var(X),                              % Need to introduce a new var
  insert_variable(Terms,X,Envs,New_envs).

%---- Grabs binding of Term in Environment, if any, else fails
%---- Modes: ++-
%---- Types: <List of [Ground_term,Variable], Term, Variable>
binding([[Term,V]|_],Term,V) :- !.
binding([_|Rest],    Term,V) :- binding(Rest,Term,V).
binding([],_,_) :- !, fail.

%---- Succeeds only if all the terms are bound to the same variable in
%---- their respective environments.
%---- Modes: +++
%---- Types: <List of Terms, List of list of [Ground term,Variable], Variable>
all_have_same_binding([H|R],[Env|Rest_envs],V) :-
  binding(Env,H,H_variable),
  H_variable == V,
  all_have_same_binding(R,Rest_envs,V).
all_have_same_binding([],[],_).
```

```
%---- Inserts a new term-variable binding into all environments
%---- Modes: x++- (the first argument is never bound)
%---- Types: <List of Terms, Var, List of list of [Ground,Var],
%----          List of list of [Ground,Var]>
insert_variable([H|T],X,[Env|Rest_envs],[Updated_env|New_rest_envs]) :-
  append([[H,X]],Env,Updated_env),
  insert_variable(T,X,Rest_envs,New_rest_envs).
insert_variable([H|T],X,[],[[[H,X]]|New_rest_envs]) :-
  insert_variable(T,X,[],New_rest_envs).
insert_variable([],_,_,[]).

%---- Computes lgg of the given list, all elements of which are compound
%---- Modes: ++--
%---- Types: <List of Terms, List of Env, Term, List of Env>
lgg_list_of_compound(List_of_terms,Envs,Lgg,New_envs) :-
  same_compound_structure(List_of_terms,Functor,_,List_by_argument),
  !,
  my_transpose(List_by_argument,List_by_term),
  lgg_list_of_list_of_terms(List_by_term,Envs,Lgg_list,New_envs),
  Lgg =.. [Functor|Lgg_list].
lgg_list_of_compound(Terms,Envs,X,New_envs) :-
  var(X),                                  % Need to introduce a new variable
  insrt_variable(Terms,X,Envs,New_envs).

%---- Succeeds iff all the terms are compound and have the same functor
%---- and arity.
%---- Modes: +---
%---- Types: <List of terms, Name, Integer, List of List of Terms>
same_compound_structure([Term|Rest],Functor,Arity,[Args|Rest_args]) :-
  functor(Term,Functor,Arity),
  Term =.. [Functor|Args],
  same_compound_structure(Rest,Functor,Arity,Rest_args).
same_compound_structure([],_,_,[]).
```

```
%---- Succeeds if there are any top-level terms of simple type among
%     the list of terms
%---- Modes: +
%---- Types: <List of terms>
has_simple([H|_]) :-
  simple(H),
  !.
has_simple([_|R]) :-
  has_simple(R).

%=====================================================================
%---- Given a list of lists of all the same length (?), it yields their
%     column-major format.
%---- Modes: +-
%---- Types: <List of lists,List of lists>
%=====================================================================
my_transpose(List,[]) :-
  allEmptyList(List),
  !.
my_transpose(L,[Heads|Rest]) :-
  combine_terms(L,Heads,Tails),
  my_transpose(Tails,Rest).

%---- Succeeds when all terms in the list are the empty list
%---- Modes: +
%---- Types: <List>
allEmptyList([[]|Rest]) :-
  allEmptyList(Rest).
allEmptyList([]).

%---- Strips apart a list of lists into separate lists of heads and tails
%---- Modes: +--
%---- Types: <List of pairs, List, List of lists>
combine_terms([[H|T]|Rest_pairs],[H|Rest_first],[T|Rest_tail]) :-
```

```
  combine_terms(Rest_pairs,Rest_first,Rest_tail).
combine_terms([],[],[]).

%=====================================================================
%---- Succeeds when each element is the same
%---- Modes: +
%---- Types: <List>
%=====================================================================
all_the_same([_]).
all_the_same([A,B|T]) :-
  A == B,
  all_the_same([B|T]).
all_the_same([]).
```

Appendix B

B. A Session with Shrinp

The system Shrinp is currently implemented in 4800 lines of Quintus Prolog code. This chapter shows a complete session with the system. It also explains how the current options can be displayed and modified.

Shrinp is loaded by typing *[demo]* at the Prolog prompt. The main screen appears, giving the user a chance to change the language to French, test a target predicate on the fly, view a demonstration and see/modify the options (see Section 5.6.2). The main screen looks like this:

*** Shrinp v1.1 ***

1. Learn a new definition on the fly

2. See a demonstration

3. Change options

4. Changer de langue

5. Help

0. -Exit

*** Enter a number followed by a dot:***

The next sections show in more detail each one of these actions.

B.1 Learning a definition on the fly

The first choice lets the user check if a definition can be found for a given set of examples without having to create a file. The system asks the set of positive and negative examples and the modes, which are to be entered on the fly. The results, if any, are shown on the screen. For instance, suppose the user enters the data below. Underlined text stands for user input.

Please end all your answers with a dot

Enter predicate's name: len.

Enter the list of positive examples: [len([c],s(0)), len([a,b], s(s(0)))].

Now enter the (empty?) list of negative examples: [len([], s(s(0)))].

Now enter the list of modes (i,o): [i,o].

Note that the list of examples is typed in Prolog list format while the arguments use the *s* function described in Section 2.1 to represent natural numbers. The relation *len* is supposed to return the length of a given list, hence the modes *[i,o]*. After processing, Shrinp succeeds in learning a correct definition for this relation by printing out the following definition. The CPU time is also informed.

```
----------------Clause(s) learned----------------

len([],0).

len([A|B],s(C)) :-

    len(B,C).

Runtime is 83 milliseconds
```

B.2 Demos

Demonstrations of some predicates that can be learned by Shrinp can be viewed if the second

choice, "*see a demonstration*", is selected. Currently, there are 31 predicates in the demo. They are shown in the following screen:

-== DEMOS ==-

Demo	*Relation Name*	*Demo*	*Relation Name*
1	*addLast*	*17*	*minus*
2	*app*	*18*	*myDelete*
3	*checkNth*	*19*	*nextTo*
4	*doubles*	*20*	*permut*
5	*eqList*	*21*	*plus*
6	*eqNList*	*22*	*remFront*
7	*extractNth*	*23*	*repeat*
8	*fact*	*24*	*reverse*
9	*isEven*	*25*	*sameDepth*
10	*isInteger*	*26*	*select*
11	*isSorted*	*27*	*subset*
12	*last_of*	*28*	*sum*
13	*lessEq*	*29*	*sumToN*
14	*lEven*	*30*	*twice*
15	*lOdd*	*31*	*twiceAsLong*
16	*member*	*0*	*EXIT*

**** Enter a number followed by a dot:**

When a valid number is picked, Shrinp loads the examples for that relation and runs the

algorithm on them, showing the obtained results. For instance, choosing the relation *sum* yields the following:

```
Shrinp Version 1.1 working...

Given positive examples:

sum([s(s(0)),0],s(s(0)))

sum([s(0),s(0)],s(s(0)))

Negative examples:

sum([s(0)],s(s(0)))

sum([0],s(s(0)))

----------------Clause(s) learned----------------

sum([],0).

sum([A|B],C) :-
        sum(B,D),
        newPred8(A,D,C).

newPred8(0,A,A).

newPred8(s(A),B,s(C)) :-
        newPred8(A,B,C).

Runtime is 2283 milliseconds
```

The definition of *sum* requires the invention of the necessary relation *plus*, represented above by the predicate *newPred8*. For this demo, the only inactive options were NEGUSER, GENEX and RELEARNBC.

B.3 Changing options and parameters

Shrinp permits that the user modifies the parameters and options discussed in Section 5.6. Choosing 3 on the main menu pops up a screen that indicates with an asterisk what options are currently active and with a number the current value of the parameters:

-== OPTIONS ==-

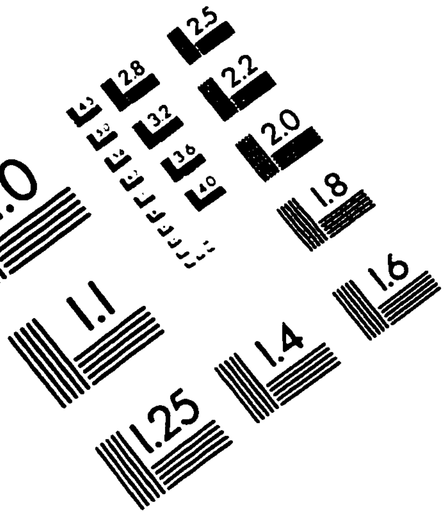
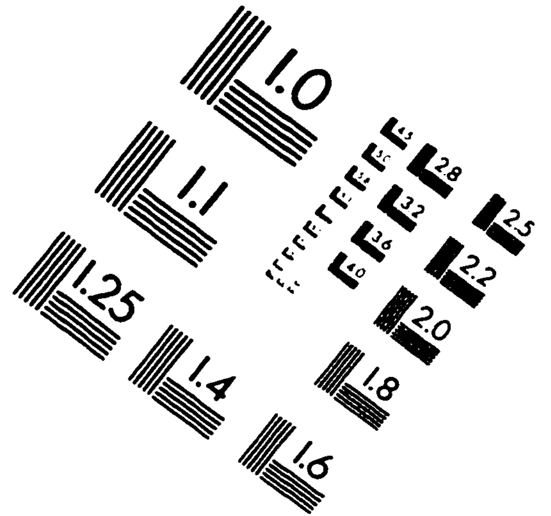
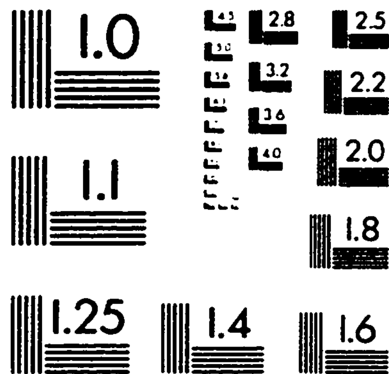
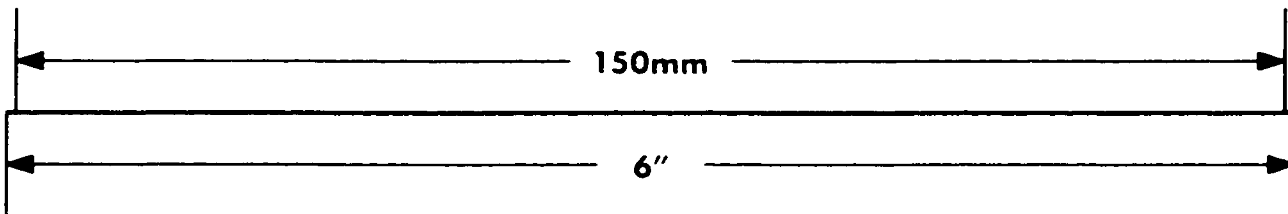
The current options are indicated by a star

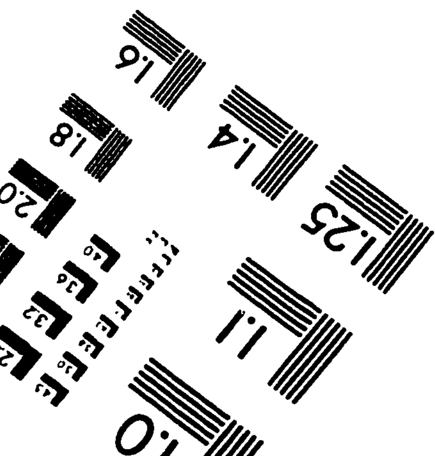
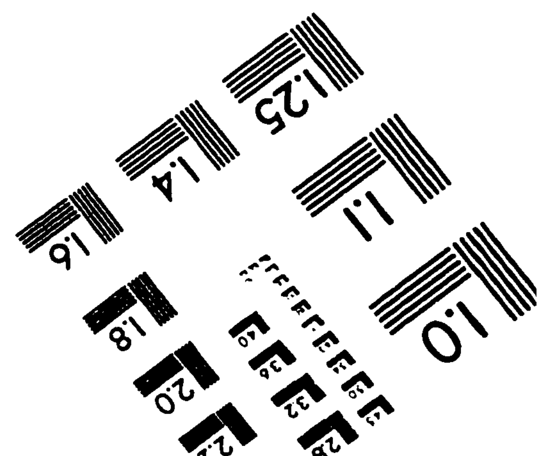
*	a. GMODE	Genex mode is input determination (off means *io-det*)
	b. GENEX	Genex algorithm is active
	c. NEGUSER	Predicate invention uses user's negative examples
*	d. SHRINP	Shrinp algorithm is active
	e. BCNEG	Wrong BCs used as ex- in finding correct base clause
	f. RELEARNBC	Base clauses are relearned after the clause is found
*	g. BADARG	Shrinp checks clauses for useless arguments
	h. MAXDEP	Maximum depth of a definition = 2
*	i. NPOS	Maximum number of positive examples (0 means all) = 3
	0. go back to main menu	

**** Enter a letter followed by a dot to toggle an option:**

To turn an option on or off it suffices to enter the letter that corresponds to that option. In the screen above, GENEX's mode is *i-det* (option *a*), although GENEX is inactive (option *b*). So are NEGUSER, BCNEG and RELEARNBC. Only BADARG and SHRINP are on. The maximum depth is set to two and the number of positive examples to be selected is set to three.

IMAGE EVALUATION TEST TARGET (QA-3)

APPLIED IMAGE, Inc
1653 East Main Street
Rochester, NY 14609 USA
Phone: 716/482-0300
Fax: 716/288-5989

© 1993, Applied Image, Inc., All Rights Reserved