KATHOLIEKE UNIVERSITEIT LEUVEN
FACULTEIT WETENSCHAPPEN
FACULTEIT TOEGEPASTE WETENSCHAPPEN
DEPARTEMENT COMPUTERWETENSCHAPPEN
Celestijnenlaan 200A — B-3001 Leuven

INTERACTIVE CONCEPT-LEARNING

Examencommissie :
Voorzitter Prof. Dr. ir. Y.D. Willems
Prof. Dr. ir. M. Bruynooghe, promotor
Dr. D. De Schreye
Prof. ir. M. Gobin, promotor
Prof. Dr. Y. Kodratoff (LRI, Orsay)
Prof. Dr. K. Morik (GMD, Sankt-Augustin)

Proefschrift voorgedragen tot
het behalen van het doctoraat
in de Informatica

door

Luc DE RAEDT

Februari 1991

Interactive Concept-Learning.

Luc De Raedt

Department of Computer Science,
Katholieke Universiteit Leuven

Abstract

Concept-learning is the best understood area in machine-learning, a subfield of artificial intelligence. Concept-learning is the process of deriving generalizations from examples and counter-examples of a given concept. Today, many systems that attempt to automate this process have been developed. Yet few of these systems are interactive or able to use background-knowledge.

A first contribution is the introduction of a synthesis of concept-learning research by means of a detailed generic concept-learning algorithm called Gencol. Evidence is provided that clearly demonstrates the power and generality of Gencol. The evidence takes the form of proving that 5 seemingly disparate concept-learning algorithms are instantiations of Gencol.

As a second contribution, we develop a well-founded framework for interactive concept-learning in knowledge based systems. The framework is incorporated in the system Clint. The most prominent features of Clint are its user-friendly and opportunistic nature; the system is easy to teach and eager to learn. These characteristics are due to the integration of many interesting techniques into a working whole : Clint generates examples, shifts its bias, uses knowledge and meta-knowledge, acquires facts, reasons about the unknown and performs constructive induction. Furthermore the system is sound in the sense that it converges to a solution when it exists.

To Lieve, for believing in me.

Acknowledgements.

First, I would like to thank Maurice Bruynooghe for numerous discussions, suggestions and readings of this text and many others. Most of all, I am grateful for his interest in the area of machine learning, which was new to him, for his gentle guidance, his insistence on rigour and his invaluable advice on all aspects of scientific research. Working with him has always been a pleasure because he keeps up with very high scientific and moral standards. I think he should be considered as my "scientific father".

To Yves Willems, I am grateful for creating the excellent environment for scientific research at the department, for his "carte blanche" to do research in the novel area of machine learning, and for his interest, encouragements and support throughout the years. He also taught me how to pose hard but relevant questions about the significance of artificial intelligence and machine learning, which lead to puting my own research in perspective.

I am grateful to Marc Gobin for accepting to be one of my promoters, his interest and suggestions on the thesis text.

Daniël De Schreye has – besides being an excellent office mate – always been a positive example of a researcher. When I fought against theory and formalizations, I often also fought against Danny who insisted many times on a mathematical treatment of my investigations. In the end, I am glad that he won most of those fights because they significantly improved the presentation and contents of this thesis, in particular the "trivial" version of $\mathcal{DVS}$ and chapter 7.

During the past years I have also had the opportunity to visit three outstanding research centers in Europe : Orsay, GMD and Aberdeen. I am very grateful to Yves Kodratoff, Katharina Morik and Derek Sleeman for their hospitality and invitations to join their groups for some time.

In many respects, my research on machine learning started in Paris and Les Arcs. My visit to Yves Kodratoff's group was not only very inspiring but also very agreeable : many ideas contained in this thesis were inspired by work of Yves and his group. To name a few : example generation, shift of bias and object knowledge acquisition. Furthermore, Yves made me (and I still do) feel part of his French family of researchers.

GMD proved to be an excellent choice for continuing my research. The work of Katharina Morik and her group on Blip was another source of inspiration for Clint. This is revealed by the use of constraints, the multi-valued logic, the acquisition of object knowledge and the constructive induction by analogy. Katharina has always been very kind to me, which I appreciate very much. She has also spent much time explaining her views on the role of the user and on practical aspects of knowledge acquisition to me. In this way Katharina's conception of machine learning research formed an adequate counterbalance of Leuven's theoretical attitude towards research.

I also appreciate very much the continuing interest, support and encouragement of Yves Kodratoff and Katharina Morik after my visits to their institute. This is best

demonstrated by their willingness to be in my jury and to attend Benelearn.

Several persons have made specific contributions to this thesis :

- Bern Martens has taught me many important things about databases and has improved many issues concerning databases and semantics in this thesis, in particular the multi-valued semantics and the use of constraints.
- Joachim Hertzberg (GMD) inspired the work on negation and multi-valued logic by asking why Clint did not use negation and the discussion that followed.
- Gunther Sablon has read and reread many versions of this thesis and earlier papers. Above all, he was always prepared to discuss things over with me.
- Johan Feyaerts with whom I developed the techniques described in chapter 6.
- Many other master students contributed directly or indirectly to the techniques in Clint. To name the most important ones : Hilde Ade, Erwin Declercq, Dominique Ectors, Patrick Lambrix and Frank Van Belle.

On many occasions I have been influenced by discussing my work with other researchers. This includes : Rudiger Oehlman, Jean-Francois Puget, Celine Rouveirol, Gheorghe Tecuci, Maarten Van Someren and Stefan Wrobel.

I also like to thank the members of the logic programming and artificial intelligence group for their interest, especially my office mates.

Outside science, I want to express my gratitude to my parents for stimulating scientific thinking during my education, to Frank Dobbelaere, for reading and correcting certain parts of this thesis (the introduction, the conclusions and the acknowledgements) but also for climbing "other heights" with me when I was not in my ivory tower. Last but certainly not least (to use the cliche she hates), I am grateful to Lieve for believing in me and for her feministic attitude which made her stimulate me to travel a lot. It is also because of this attitude that I have taken the liberty to turn all users of Clint into women.

I also wish to acknowledge the grants that supported my stays abroad and my research: the KULeuven, the Belgian incentive for Artificial Intelligence (RFO/AI/3), the Belgian National Fund for Scientific Research, the German National Center for Computer Science, the departments of Computer Science in Orsay and Aberdeen, IBM and IJCAII.

Sint-Amandsberg, 12 January, 1991.

Contents

1 Introduction. 1
- 1.1 Machine Learning. 1
- 1.2 Learning apprentices and concept-learning. 2
- 1.3 A brief reader's guide 3
- 1.4 Overview 4

2 Principles of Concept-Learning. 7
- 2.1 Introduction. 7
- 2.2 The problem of concept-learning. 8
- 2.3 Concept-learning as search. 12
  - 2.3.1 Structuring the search-space. 12
  - 2.3.2 The GENeric COncept-Learning algorithm 14
  - 2.3.3 Possible choices for Gencol's parameters. 16
- 2.4 Well-known instantiations of Gencol 18
  - 2.4.1 Typical concept-learning systems. 18
  - 2.4.2 The Top-Down Induction of Decision Trees family. 19
  - 2.4.3 The candidate elimination approach. 21
  - 2.4.4 The AQ methodology using Induce. 24
  - 2.4.5 The Model Inference System. 27
- 2.5 Applying Gencol in practice. 29
- 2.6 Related work. 30
- 2.7 Conclusions. 31

3 Interactive Concept-Learning. 33
- 3.1 Introduction. 33
- 3.2 A logical framework for concept-learning. 35
- 3.3 Abstract concept-learners. 40
  - 3.3.1 The conjunctive versionspace. 42
  - 3.3.2 The disjunctive versionspace. 42
  - 3.3.3 The meaning of $\mathcal{VS}$ and $\mathcal{DVS}$. 44
- 3.4 Bias in concept-learning. 45
  - 3.4.1 Language bias and search bias. 45
  - 3.4.2 Language bias: some assumptions. 47
  - 3.4.3 Generalization. 48
  - 3.4.4 Justifications. 50
- 3.5 Clint's basic algorithms. 52
  - 3.5.1 Approaching the problem. 52

3.5.2 Generalizing examples. . . . 53
3.5.3 Specialization. . . . 57
3.5.4 Clint's basic algorithm. . . . 60
3.5.5 Some sessions with Clint. . . . 60
3.6 Theoretical aspects of Clint's basic algorithm. . . . 65
3.6.1 Inductive inference: some concepts. . . . 65
3.6.2 Convergence of Clint. . . . 66
3.7 (Im)perfect query answering procedures. . . . 71
3.7.1 Problems with perfect query answering mechanisms. . . . 72
3.7.2 The need for a perfect query answering mechanism 72
3.7.3 Allowing imperfect query answering procedures. . . . 73
3.8 Complexity of Clint. . . . 75
3.9 Clint as an instantiation of Gencol 76
3.9.1 Clint's disjunctive part. . . . 76
3.9.2 Clint's conjunctive part 77
3.10 Related work for Clint. . . . 79
3.11 Summary of the achievements. . . . 81

4 Indirect Relevance and Bias. 83
4.1 Introduction. . . . 83
4.2 Language bias revisited. . . . 84
4.3 A solution for indirect relevance. . . . 86
4.3.1 Abstract languages. . . . 86
4.3.2 When and how to shift the bias ? 92
4.3.3 Using knowledge to reduce the search-space. . . . 94
4.3.4 Dynamically computing the next language bias. . . . 97
4.4 Applying the method in practice. . . . 98
4.4.1 Versionspaces. . . . 98
4.4.2 Integrating the method in Clint. . . . 98
4.5 Related Work. . . . 104
4.6 Summary. . . . 105

5 Meta-Knowledge and Constraints. 107
5.1 Introduction. . . . 107
5.2 Constraints in interactive concept-learning. . . . 108
5.2.1 Integrity constraints 108
5.2.2 The relation between integrity constraints and meta-knowledge . . . 110
5.3 Principles for learning from constraints. . . . 110
5.4 The lazy method. . . . 111
5.4.1 The lazy method: algorithm. . . . 112
5.4.2 The lazy method: properties. . . . 115
5.5 The eager method. . . . 116
5.5.1 Eager method: algorithm 116
5.5.2 Eager method: properties 122
5.6 Conclusions and Related Work. . . . 123

6 Knowledge Acquisition. 127
6.1 Introduction. . . . 127
6.2 Problem-specification. . . . 128
6.2.1 Why Clint might run into difficulties. . . . 128
6.2.2 Problem-specification. . . . 129
6.3 Updating the knowledge base. . . . 130
6.3.1 Updating rule predicates. . . . 132
6.3.2 Updating Fact Predicates. . . . 134
6.3.3 A heuristic to select the most interesting candidates first. . . . 141
6.3.4 Properties of the algorithms. . . . 141
6.4 An example session with the system. . . . 142
6.5 Integrating the methods in Clint. . . . 145
6.6 Conclusions and Related Work. . . . 147

7 Negation and Multi-valued Logic. 149
7.1 Introduction. . . . 149
7.2 The problems with negation. . . . 150
7.3 An abstract approach. . . . 151
7.3.1 Mitchell's versionspace approach. . . . 151
7.3.2 A general approach: explicit representation of the negated concept. 152
7.4 A multi-valued logic for concept-learners. . . . 153
7.5 Special cases of the multi-valued logic. . . . 155
7.5.1 A three-valued logic : true, false and unknown. . . . 155
7.5.2 A three-valued logic : true, false and inconsistent. . . . 157
7.5.3 A classical two-valued logic : true and false. . . . 158
7.6 Adapting Clint. . . . 158
7.6.1 The modified problem statement of Clint. . . . 158
7.6.2 Modifying Clint. . . . 159
7.7 Properties of the three-valued Clint. . . . 162
7.8 Conclusions and Related Work. . . . 164

8 Constructive Induction by Analogy 165
8.1 Introduction. . . . 165
8.2 Some notions . . . 166
8.3 Applying analogy to concept-learning. . . . 167
8.4 Constructive Induction by Analogy. . . . 169
8.5 Experiments with Clint-Cia . . . 178
8.5.1 The family domain. . . . 178
8.5.2 Learning simple chess rules . . . 180
8.6 Evaluation of the experiments and Cia . . . 186
8.6.1 A fundamental difficulty . . . 186
8.6.2 Asking less questions . . . 187
8.6.3 Analogy . . . 188
8.6.4 Towards an intelligent predicate editor . . . 189
8.6.5 Learning how to learn. . . . 191
8.7 Related work. . . . 191

9 Applications **193**
9.1 Introduction. 193
9.2 Program synthesis 193
9.3 Learning apprentices. 194
9.4 Autonomous agents 195
9.5 Knowledge acquisition and sloppy modeling 196
9.6 Testing, verification and validation 197
9.6.1 Using Clint for generating experiments. 197
9.6.2 Modifying the knowledge base. 198
9.6.3 Using integrity constraints. 199
9.6.4 Summary. 199
9.7 Conclusion 199

10 Conclusions **201**
10.1 Summary of the Achievements. 201
10.2 Future research. 202

List of Figures

1.1 Dependency graph of chapters 4

2.1 The covers relation. 9
2.2 The taxonomies of the concept-learning problem 11
2.3 C_1 is more specific than C_2 12
2.4 The structure of a decision tree. 19

3.1 A dependency graph. 39
3.2 Illustrating language bias 46

6.1 The typegraph of the drives knowledge base. 137
6.2 The typegraph of the planes knowledge base. 145

7.1 Two-valued and three-valued classification. 151

8.1 Analogy 168
8.2 A chess board 181

List of Algorithms

2.1 Gencol 15
3.1 A naive generalization operator for a finite language. 49
3.2 Computing $\mathcal{JUS}(KB, L, p)$ specific to general. 52
3.3 Generalization in Clint. 55
3.4 Computing a clause for an uncovered positive example. 58
3.5 Specialization in Clint. 59
3.6 Recovering from a covered negative example. 60
3.7 Basic algorithm of Clint. 61
4.1 Two-layered concept-learning. 93
4.2 Choosing the next bias dynamically 98
4.3 Clint and shifting the bias. 100
5.1 The lazy constraint algorithm 113
5.2 The eager constraint algorithm 119
5.3 The eager constraint algorithm (continued) 120
6.1 Solving a goal. 133
6.2 Realizing rule updates. 135
6.3 Realizing Fact Updates. 138
6.4 Procedures used by handle_fact. 139
6.5 Integration in Clint 146
6.6 Integration in Clint (continued) 147
8.1 Cia version of handle_uncovered. 171
8.2 Constructive Induction by Analogy 172

Chapter 1

Introduction.

1.1 Machine Learning.

Machine learning is a subfield of artificial intelligence, which is a young science combining principles of computer science and cognitive science. Whereas artificial intelligence attempts to build *intelligent* machines; its subfield, machine learning, attempts to build learning machines. A learning machine can be defined as a machine that is able to make changes to itself, which enable it to perform better at a given task. Therefore a machine or system can be said to learn if it works more precise, can handle a broader class of problems or works more efficiently over time.

There are at least two good reasons to explain the current interest in machine learning. First, although there is no consensus on the nature of intelligence, people agree that something is not intelligent unless it is capable of learning. People like John Haugeland state for instance that

> Certainly the ability to learn is essential to full intelligence; artificial intelligence cannot succeed without it.

John Haugeland's view is commonly accepted and partly explains the prominence of machine learning in artificial intelligence. This justifies machine learning from the scientific point of view.

Second, machine learning can contribute to the automation of the knowledge acquisition phase, which forms a justification from the engineering point of view. Indeed, the success of artificial intelligence in industry is to a large degree due to the performance of knowledge based systems. It is generally accepted (and by now a cliche) that the main problem to be faced when building knowledge based systems is the acquisition of knowledge. This acquisition is very expensive because it is necessary to interview and observe domain experts. The experts are often unable to communicate their expertise in a suitable form for use by the machine. Today, there is a high demand and interest for tools that assist during the knowledge acquisition process [Gaines and Boose]. Machine learning techniques can be successfully applied in such tools.

The history of machine learning starts around 1960, when Arthur Samuel developed his checker player system [Samuel 67]. Samuel's program did not only perform very well at playing checkers — it could defeat several champions — but it also improved its performance. To improve the program's behaviour, a relatively simple indexing mechanism was used together with an evaluation function whose weights were automatically adapted.

Arthur Samuel's contribution was certainly one of the most impressive results of early machine learning research.

In the 1970's, with the advent of expert systems, researchers became very much interested in empirical learning. This kind of learning considers the problem of deriving general rules from examples. The assumption underlying empirical learning states that general rules which perform well at classifying the known examples will also perform well at predicting the classification of unseen examples. In the late 1970's, the first successes of this engineering branch of machine learning were realized : the learning of diagnostic rules for soy-bean diseases [Michalski and Chilausky 80], the use of empirical learning in the Meta-Dendral project [Buchanan and Mitchell 78] and the approach of [Quinlan 79] to derive decision trees from large collection of examples.

The success of these techniques and the impressive results of the automated discovery system AM [Lenat 83] attracted a rapidly growing number of researchers in the 1980's. The main novelties of this period include certainly explanation-based learning [Mitchell, Keller and Kedar-Cabelli 86, De Jong and Mooney 86], which aims at improving the efficiency of knowledge based systems by reformulating knowledge, genetic learning [De Jong 88], which incorporates biological evolution and darwinism in computer programs that improve their behaviour, and learning apprentice systems [Mitchell, Mahadevan and Steinberg 85, Smith, Mitchell, Winston and Buchanan 85], which are interactive aids that improve their performance over time during normal operation of the system.

The last class of programs motivates our work on interactive concept-learning. Let us therefore investigate it in more detail.

1.2 Learning apprentices and concept-learning.

[Smith, Mitchell, Winston and Buchanan 85] define a learning apprentice as follows :

> A *learning apprentice system* is an interactive aid for building and refining a knowledge base. Its aims are twofold : (1) to partially automate initial construction of a knowledge base by generating shallow rules automatically from an approximate domain theory; and (2) to interact with users to help refine the knowledge through experience gained during normal problem solving.

Today, there are several learning apprentices : e.g. Leap [Mitchell, Mahadevan and Steinberg 85], the one of [Smith, Mitchell, Winston and Buchanan 85], Disciple [Tecuci and Kodratoff 90], Blip [Morik 89] and Odyseus [Wilkins 89]. Most of the mentioned approaches are large and rather complex knowledge based systems that integrate many features and subsystems. The system Disciple has (cf. also section 9.3), for instance, three integrated learning modules, a knowledge base consisting of action models and object knowledge and an expert system shell. There is a wide variety of learning techniques that is employed by these apprentices. It ranges from explanation based learning (Leap, Disciple) over analogy (Disciple) to discovery (Blip) and empirical learning (Odyseus, Disciple, Blip and the approach of [Smith, Mitchell, Winston and Buchanan 85]).

The learning components of all these apprentices (except Leap) can also be seen as concept-learners, i.e. programs that derive generalizations from examples and counter-examples of a concept. This reveals that concept-learning is a major subtask in learning

apprentices. In the literature (see [Michalski, Carbonell and Mitchell 83, Michalski, Carbonell and Mitchell 86, Kodratoff and Michalski 90]), many concept-learning techniques are described. However, most of them study how to learn general rules from a given set of examples. Also, few techniques use knowledge to improve the result of the learning. As a consequence, these concept-learning techniques are not directly suited for use in the learning apprentice generation of knowledge based systems. This leaves an interesting opportunity for an investigation of these matters.

In my opinion, the two principle characteristics of learners in the learning apprentice context should be :

- user-friendliness : a system is user-friendly if it can be used in a simple manner by users that are not familiar with the underlying techniques
- opportunism : a system is opportunistic for learning if it takes the initiative to learn whenever there is an opportunity to do so

This means that these learners should be easy to teach and eager to learn. One of the most important implications of user-friendliness and opportunism, is *interactiveness*. Interactive systems generate clever questions to the user, which enable these systems to speed up the learning.

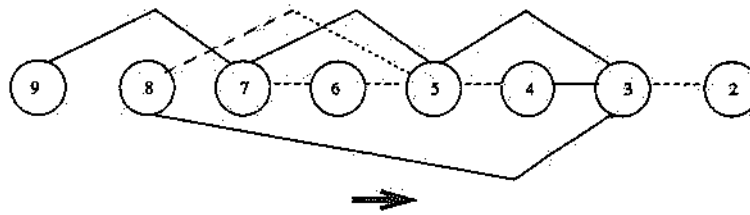
Few of the existing concept-learners meet these requirements. Probably the systems that meet these requirements best are Disciple [Tecuci and Kodratoff 90] and Marvin [Sammut and Banerji 86]. However, the approaches to concept-learning in Disciple and Marvin leave a number of open questions and seem to be extendible. For both systems, it is for instance not known which class of concepts can be learned. One aspect in which they are certainly extendable is the use of meta-knowledge in the form of constraints (see chapter 5).

The state of the art of concept-learning motivates an investigation of concept-learning that is focussed around the issues of user-friendliness and opportunism. Such an investigation is the subject of this thesis. Our investigation will study – although inspired on systems such as Disciple and Marvin – opportunistic and user-friendly concept-learning in its own right, without a particular target learning apprentice in mind. Rather, we aim at a generally applicable and firm framework for such concept-learning. A generally applicable concept-learner will alleviate the need for redesigning a novel concept-learner each time a novel learning apprentice is developed. As an incarnation of our framework, we have developed the concept-learning environment Clint (Concept-Learning in an INTeractive way [De Raedt and Bruynooghe 89e]). It can be used as a module for learning apprentice and knowledge acquisition systems. This is also the way in which our work on Clint is meant. Our approach does not pretend to be relevant from a psychological or cognitive point of view.

1.3 A brief reader's guide

The chapters in this thesis all present a different aspect of interactive concept-learning. Some of these aspects are very much related whereas others are not. The result is that some chapters can be read rather independently of each other. A graph that mirrors the interdependencies among chapters is shown in figure 1.1. A dotted arc between two

Figure 1.1: Dependency graph of chapters



numbers means that the first chapter weekly depends on the second one, i.e. the second chapter is not really a prerequisite for understanding the first one, although the second chapter refers at some place(s) to the first one. A full arc from one chapter to a second one means that the first chapter strongly depends on the notions introduced in the second one. To understand the first chapter, one has to read the second one. Note also that the arcs obey the following rules :

- a dotted arc followed by a dotted arc gives a dotted arc
- a full arc followed by a full arc gives a full arc
- a full arc followed by a dotted one gives a dotted one
- a dotted arc followed by a full one gives a full one
- if one can derive a full arc among two chapters and a dotted one, the result is a full arc

1.4 Overview

Before talking about interactive concept-learning, it is a good idea to summarize existing work on concept-learning. Doing so will clarify the principles of concept-learning. This is done in chapter 2, where we revisit Mitchell's *Generalization as search* paradigm. Instead of presenting a survey of existing work on concept-learning, we will develop a unifying framework for concept-learning algorithms and systems. The framework is centered around the generic concept-learning algorithm Gencol. The algorithm is generic in the sense that it makes abstraction of the most important issues in concept-learning. This includes : the description languages, the search-strategy, the heuristics and the operators. To demonstrate the generality of the method, it is proven that 5 seemingly disparate

concept-learning algorithms are instantiations of Gencol. Our synthesis of concept-learning in the generic algorithm Gencol is unique in the sense that it can be be instantiated to almost any concept-learning system that employs generalization and specialization and, that it shows how seemingly disparate algorithms are instantiations of a more general algorithm.

In chapter 3, we lay the foundations of our approach to interactive concept-learning. Firstly, the problem of interactive concept-learning is specified at an abstract level by using two different models of concept-learning. These two models highlight important issues of concept-learning in our context. They are used in further chapters to reason about concept-learning. Secondly, a practical approach to interactive concept-learning is developed within a logical framework. It is incorporated in the system Clint. The distinctive features of the basic algorithm of Clint are that Clint is both user-friendly and opportunistic. One reason for this is that it generates examples and presents them to the user for classification. Furthermore, Clint uses the available knowledge, which is contained in the knowledge base. Although the approach is practical, we will also investigate its theoretical properties. It will be shown that – under certain conditions – Clint converges to a correct knowledge base. I will also discuss the potential use of Clint for program synthesis.

In chapter 4, we attack the problem of indirect relevance and the limitations imposed by a fixed description language of concepts. The problem of indirect relevance is concerned with the formalism in which examples are described. Most approaches require that an example explicitly mentions all properties of the example that are relevant for the example as an instance (or non-instance) of the concept. In the context of Clint, this is not necessary as examples are ground facts and further properties are derived from the knowledge base. A related issue in concept-learning is the concept-description language. Usually it is assumed that the concept can be described in a certain description language of concepts. In case the concept to be learned falls outside the given description language, the learner faces a difficult problem. In Clint this problem is solved by having the ability to shift the bias, i.e. to change the description language of concepts. Clint will employ a different description language when it detects that its current description language is not sufficient to learn the current concept.

In chapter 5, a novel approach to using meta-knowledge in concept-learning is presented. The approach is fully integrated in the Clint system and allows the user to state properties about the knowledge base to be learned. Such properties can be seen as a generalization of examples. Indeed, whereas an example states that an individual fact is true or false, a property states that certain relations hold among different concepts of the domain. In our framework, properties are expressed as general clauses. These clauses act as constraints on the knowledge base to be learned. The constraints have to be satisfied by the learned knowledge base. We will see how to adapt Clint to cope with such constraints and present some theorems concerning the resulting convergence behaviour.

In the next chapter, an approach to acquire fact knowledge about objects is presented. The fact knowledge about an object is the set of all facts in the knowledge base that mention the object. Fact knowledge is an important part of the knowledge base. In chapter 6, two algorithms are presented that ask questions to the user about objects. Their aim is to acquire fact knowledge. We also show how these algorithms can be integrated into Clint.

In chapter 7, we first show that there are some major problems with negation from the

point of view of concept-learning. These problems are pragmatic as well as conceptual. We will show that current approaches to negation, in particular negation as failure, are not feasible when learning concepts. This motivates a different approach which is based on a multi-valued logic. The multi-valued logic which we will use, has a sound semantics and allows to alleviate the problems with negation. The multi-valued logic is easy to integrate in Clint.

Constructive induction is addressed in chapter 8. Constructive induction techniques propose new concepts to the user rather than learning existing concepts from examples. The user can then evaluate whether the proposed concepts are useful and if so, the concept-descriptions are asserted in the knowledge base for future use. In our approach to constructive induction, the Constructive Induction by Analogy (Cia) technique, we propose new concepts to the user by analogy to known ones. The technique can be used in basically two different ways : to guess concept-descriptions or to generate new concepts. The approach is fully integrated in Clint. To evaluate the resulting system, two experiments were performed. They show the usefulness of the technique in domains with structurally related predicates.

To demonstrate the use of the concept-learning environment Clint, we outline some possible applications in chapter 9.

Finally, chapter 10 contains our conclusions and some directions for further research.

Chapter 2

Principles of Concept-Learning.

2.1 Introduction.

Concept-learning from positive and negative examples occupies a central position in the field of machine learning. Concept-learning techniques have been successfully applied in a wide range of situations like learning heuristics for symbolic integration [Mitchell, Utgoff and Banerji 83], deriving diagnostic rules for soy-bean diseases [Michalski, Davis, Bisht and Sinclair 85], automatic discovery of mathematical theorems [Lenat 83], ... Learning concepts from examples has developed into a broad area of research; numerous concept-learning techniques are described in the literature (see e.g. [Michalski, Carbonell and Mitchell 83, Michalski, Carbonell and Mitchell 86, Kodratoff and Michalski 90]). As for other broad research areas, distinguished researchers [Mitchell 82, Michalski 83, Dietterich and Michalski 83, Laird 86, Angluin and Smith 83, Gams and Lavrac 87, Shapiro 81, Bundy, Silver and Plummer 85, Rendell 86],... have analyzed different algorithms in order to extract the underlying principles, to point out the similarities and differences between various approaches and to explain systems independent of their implementation or application. Although the mentioned work led to a better understanding of the field, most of it focussed on analysis rather than synthesis. Indeed, concept-learning has been analyzed as a search-problem [Mitchell 82] and as an application of generalization and specialization rules [Mitchell 82, Laird 86, Michalski 83]. Furthermore, a few comparative surveys have been made [Bundy, Silver and Plummer 85, Dietterich and Michalski 83, Cohen and Feigenbaum 81, Angluin and Smith 83, Langley and Carbonell 87].

As opposed to these analytical investigations, we present a synthesis of concept-learning in the form of the generic concept-learning algorithm Gencol (GENeric COncept-Learning algorithm). By the term generic we mean that abstraction is made on important parameters characterizing specific algorithms. Gencol builds further on Mitchell's landmark paper, *Generalization as search* [Mitchell 82], as Gencol makes abstraction of the search-strategy and the description languages. I conjecture that Gencol can be instantiated to most existing concept-learning algorithms. To support this claim I indicate how it can be used to obtain seemingly disparate systems: the TDIDT-family [Quinlan 86], the Model Inference System [Shapiro 83], the AQ methodology [Michalski 83] and the candidate elimination algorithm or versionspace approach of [Mitchell 77]. The generic algorithm Gencol does not only serve synthetic purposes but also analytic ones since formulating different concept-learning algorithms in the same framework facilitates the understanding and appreciation of the similarities and differences between the various approaches. Fur-

thermore, because Gencol synthesizes many concept-learning systems, it provides a good introduction to the area of concept-learning and mirrors its principles.

Up till now there has been little synthetical work on concept-learning, with a few notable exceptions like [Shapiro 81, Gams and Lavrac 87, Michalski 83, Laird 86]. The approach taken in this chapter is unique in the sense that it is

1. *general enough to cover (to the best of my knowledge) all algorithms that use specialization and generalization to search the space*
2. *that it shows how seemingly disparate techniques are instantiations of the same algorithm*

This chapter is organized as follows: in section 2.2, the problem of concept-learning is clarified and defined; in section 2.3, the generic concept-learning algorithm Gencol is presented and explained; in the next section it is shown how Gencol can be instantiated to obtain a wide range of algorithms; in section 2.5 we briefly discuss a practical tool based on Gencol and finally, in section 2.6 we touch upon related work and formulate some conclusions.

2.2 The problem of concept-learning.

Intuitively, concept-learning [Mitchell 82] is the ability to generalize specific observations by extracting common and important features characterizing these observations.

Definition 2.1 *A concept C is a set of observations.*

For each concept C there are two associated kinds of observations o :

Definition 2.2 $\forall o \in C$: *o is a positive observation for C.*

Definition 2.3 $\forall o \notin C$: *o is a negative observation for C.*

The aim of concept-learning is to find a rule (or a set of rules) which is consistent (in some sense) with the positive and negative observations.

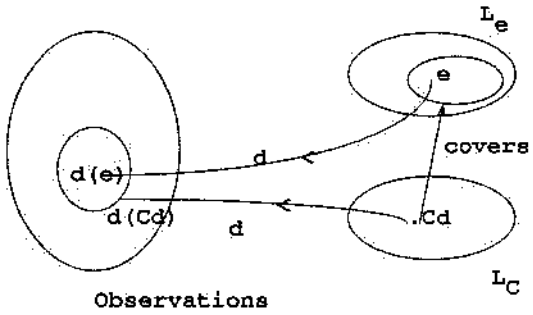
For the purpose of learning concepts with machines, suitable representations for observations and concepts are needed. Therefore, it is assumed that the observations are described (i.e. represented) in a language of observation-descriptions L_e. Descriptions of observations are *examples*. Similarly, the concepts are described in a language of concept-descriptions L_C. For the language of examples L_e and the language of concepts L_C, there is a function d that maps descriptions of observations (resp. descriptions of concepts) on observations (resp. concepts). The function d can be regarded as a denotational function that maps syntax on semantics.

Definition 2.4 $\forall e \in L_e$: *$d(e) = o$ iff e is the description of the observation o.*

Definition 2.5 $\forall Cd \in L_C$: *$d(Cd) = C$ iff Cd is the description of the concept C.*

Positive examples correspond to positive observations and negative examples to negative observations.

Figure 2.1: The covers relation.



Definition 2.6 *$p \in L_e$ is a positive example for a concept C iff $d(p) \in C$*

Definition 2.7 *$n \in L_e$ is a negative example for a concept C iff $d(n) \notin C$*

Obviously, the learner must be able to relate the language of concept-descriptions L_C to the language of examples L_e. So, we also assume a predicate *covers*(Cd, e), where $Cd \in L_C, e \in L_e$:

Definition 2.8 *covers(Cd, e) = if $d(e) \in d(Cd)$ then true else false.*

Roughly speaking, *covers*(Cd, e) succeeds iff the observation $d(e)$ described by the example e is in the concept $d(Cd)$ described by the concept-description Cd. We also define the related mapping *covers*(Cd) from L_C to the set of all subsets of L_e :

Definition 2.9 $\forall Cd \in L_C$: *covers*$(Cd) = \{e \in L_e \mid$ *covers*$(Cd, e) =$ *true* $\}$

These definitions are illustrated in figure 2.1.

The aim of concept-learning is to find a concept-description in L_C which satisfies some quality criterion. The quality criterion [Mitchell 82, Shapiro 81, Michalski 83] often states that the concept-description must be complete and consistent. From now on we will ignore the observations and work with concept-descriptions and examples.

Definition 2.10 *A concept-description $Cd \in L_C$ is complete w.r.t. a set of positive examples P iff $\forall p \in P$: covers$(Cd, p) =$ true.*

Definition 2.11 *A concept-description $Cd \in L_C$ is consistent w.r.t. a set of negative examples N iff $\forall n \in N$: covers$(Cd, n) =$ false.*

Complete and consistent concept-descriptions do not always exist. The reason may be that L_C is too restricted or that $P \cap N \neq \emptyset$. This last reason may also occur indirectly because of a discrepancy between observations and examples or between concepts and concept-descriptions. This occurs when the mapping d lacks precision, for instance when there is noise.

Because complete and consistent concept-descriptions do not always exist, there are also relaxed quality-criteria (see e.g. [Buchanan and Mitchell 78, Valiant 84]). In general, the *Quality-Criterion* depends on the sets of positive and negative examples (P and N).

By now we are able to define a concept-learning problem more formally as:

Problem Statement 2.1 *Concept-Learning*

- **Given:**
  - a language of examples L_e
  - a language of concepts L_C
  - the *covers* relation between L_C and L_e
  - a set of positive examples P
  - a set of negative examples N
  - a *Quality-Criterion* Q defined on strings of L_C and taking into account P and N
- **Find:** concept-description(s) $C \in L_C$ satisfying the *Quality-Criterion*

This problems statement is illustrated in example 2.1.

2.3 Concept-learning as search.

Most of the concept-learning techniques described in the literature, can be viewed as search-techniques (cf. [Mitchell 82]), in which the states are the strings in L_C and the aim is to find one or more states satisfying the quality criterion. With this in mind, it is easy to develop a simple generate and test algorithm, which generates all concept-descriptions and tests whether they satisfy the quality criterion. This algorithm is known as the enumeration technique and has been studied by [Angluin and Smith 83, Biermann 86, Gold 67]. Although the enumeration technique has some interesting properties it is, as for other search-problems [Angluin and Smith 83], more advantageous to structure the search-space.

2.3.1 Structuring the search-space.

For concept-learning, the *is more specific than* relation provides a natural and powerful basis to organize the search (see [Mitchell 82]):

Definition 2.12 $\forall C_1, C_2 \in L_C$: *C_1 is more specific than C_2 iff* $covers(C_1) \subseteq covers(C_2)$. *We also say that C_1 is a specialization of C_2, C_2 is a generalization of C_1 or C_2 is more general than C_1.*

Example 2.1: A concept-learning problem and its solution.

Consider the following concept-learning problem (the taxonomies are shown in figure 2.2):

- $L_e = \{x \wedge y \mid x$ is a leaf of the shape taxonomy and y is a leaf of the color taxonomy$\}$
- $L_C = \{x \wedge y \mid x$ is a node of the shape taxonomy and y is a node of the color taxonomy$\}$
- $covers(cx \wedge cy, ex \wedge ey)$ = true iff cx is an ancestor of ex for shape and cy is an ancestor of ey for color
- P = { `square` $\wedge$ `red`, `rectangle` $\wedge$ `blue` }
- N = { `circle` $\wedge$ `orange`, `triangle` $\wedge$ `pink` }
- The quality criterion is completeness and consistency

This problem has solutions, for instance: `4-sided` $\wedge$ `color` and `shape` $\wedge$ `mono`.

Figure 2.2: The taxonomies of the concept-learning problem

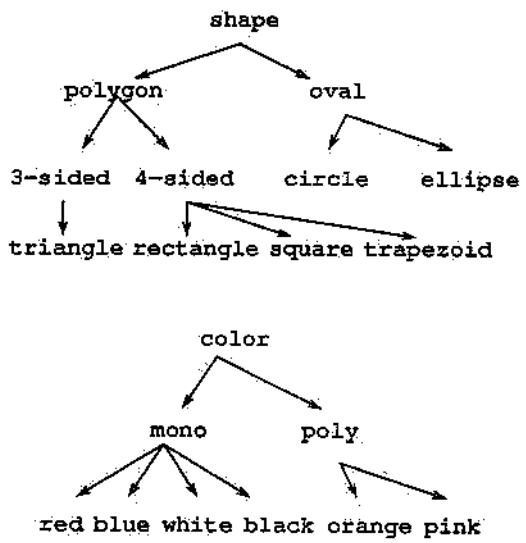
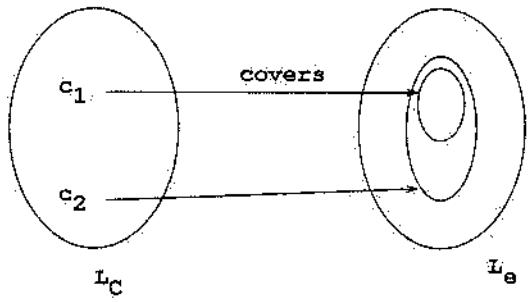


Figure 2.3: C_1 **is more specific than** C_2



This definition is illustrated in figure 2.3. On example 2.1, we have for instance that `oval` $\wedge$ `poly` is more general than `circle` $\wedge$ `pink`.

Definition 2.13 *A generalization C_1 (resp. specialization) of C_2 is strict when covers(C_1) $\neq$ covers(C_2).*

Notice that the relation *is more specific than* is transitive and reflexive. Unfortunately, it is not always anti-symmetric since there may exist several concept-descriptions which cover the same set of examples. Such descriptions are called syntactic variants:

Definition 2.14 $\forall C_1 \neq C_2 \in L_C$: *$C_1$ is a syntactic variant of C_2 iff covers(C_1) = covers(C_2).*

In theory, one can obtain a partial order by introducing an equivalence class and working with a canonical form as a representative of the equivalence class. In practice, there is not always a decision procedure to check whether two descriptions are syntactic variants (cf.[Buntine 88]). This sometimes leads to problems when searching the space of concept-descriptions.

It is convenient to introduce a special top element $\top$ (cf. [Shapiro 83]) and a special bottom element $\bot$ such that:

Definition 2.15 *covers($\top$) = L_e and covers($\bot$) = $\emptyset$*

$\top$ and $\bot$ may but need not denote strings belonging to L_C. The relation *is more specific than* can be used to prune the search provided that the *Quality-Criterion* is well-behaved with respect to the covers relation.

Definition 2.16 *A Quality-Criterion Q behaves well with respect to the covers relation iff for all $C_1, C_2 \in L_e$ such that C_1 covers more positive examples than C_2 without covering more negative ones or C_1 covers less negative examples than C_2 without covering less positive ones, $Q(C_1) \geq Q(C_2)$.*

The notation $Q(C_1) \geq Q(C_2)$ means that the quality of C_1 is higher or equal than the quality of C_2.

The properties which allow the *is more specific than* relation to prune the search are:

Lemma 2.1 *Pruning*

1. *If a concept-description covers an example then all generalizations of the concept-description will also cover the example*
2. *If a concept-description does not cover an example then no specialization of the concept-description will cover the example.*

In example 2.1, if we have that `4-sided` $\wedge$ `mono` covers the negative example `square` $\wedge$ `red` then we also know that all generalizations of `4-sided` $\wedge$ `mono` will cover the example. Therefore we should look for specializations of `4-sided` $\wedge$ `mono` such as `4-sided` $\wedge$ `white`.

In order to facilitate the formulation of the generic algorithm, transformers are introduced:

Definition 2.17 *A Generalization-transformer gt is a mapping from L_C to 2^{L_C} such that* $\forall C' \in gt(C) : covers(C) \subseteq covers(C')$

Definition 2.18 *A Specialization-transformer st is a mapping from L_C to 2^{L_C} such that* $\forall C' \in st(C) : covers(C') \subseteq covers(C)$

One can distinguish different kinds of transformers and languages. In Gencol, they give rise to different algorithms. Therefore, a language-framework is presented. It highlights important issues in the knowledge-representation of concept-learning and is to a large degree independent of particular knowledge-representation mechanisms. It can be used to describe a wide range of concept-learning systems (cf. section 2.4). The language-framework allows us to characterize different instantiations of Gencol easily and by no means restricts Gencol, since other frameworks might be used as well.

Most concept-learning systems use subsets (or variants) of first order logic [Genesereth and Nilsson 87] as description-languages. Without loss of generality, we assume that concept-descriptions in these systems are of the following form:

$$F_1 \vee F_2 \vee ... \vee F_n; \quad \text{such that } \forall F_i : F_i \text{ is a conjunctive formula}$$

Generalizations (specializations) of strings of this type can be obtained by generalizing (specializing) individual disjuncts of the string or by adding (deleting) a disjunct to the string. Generalization and specialization transformers that work this way are called *derived* transformers. The corresponding transformers on formulas are *formula-transformers.*

Definition 2.19 *Strict (generalization or specialization) transformers t are transformers such that :* $\forall C \in L_C : \forall C' \in t(C) : covers(C) \neq covers(C')$

Definition 2.20 *t is a null generalization (resp. specialization) transformer iff* $\forall C \in L_C : t(C) = \{\top\}$ *(resp.* $\{\bot\}$*).*

Definition 2.21 *The generalization (resp. specialization) operator is a strict transformer t such that $t(C)$ is the set of most specific generalizations (most general specializations) of C. Formally, the generalization operator $t(C) = \{C' \in L_C \mid C'$ is strictly more general than C and there is no $C'' \in L_C$ such that C'' is strictly more general than C and C'' is strictly more specific than $C'\}$.*

The formal definition of the specialization operator is obtained dually.

Definition 2.22 *An example-driven generalization (resp. specialization) transformer t_e with respect to an example e is a strict transformer such that all elements in $t_e(C)$ cover e (resp. do not cover e).*

2.3.2 The GENeric COncept-Learning algorithm

Below, a detailed description of the generic algorithm Gencol is presented. Possible instantiations for the parameters of Gencol are briefly discussed in section 2.3.3. Variables of Gencol are in roman and parameters in *italic*. It is assumed that all procedures (i.e. generic parameters) of Gencol have access to P, N, L_e, L_C and the Memory.

Partial_Solutions is used as a queue which contains the concept-descriptions that are currently under consideration. Complete_Solutions contains the output of Gencol. The Memory is (sometimes) used to keep track of concept-descriptions that cannot lead to a solution. P is the set of positive examples and N the set of negative examples. ps is a variable which contains the concept-description (partial solution) that is currently under consideration and that will be expanded or moved into the set of complete solutions. In ex, Gencol may store an example to be used by an example driven transformer. *Optional* specifies that the ex := *Generate_Example* part is only invoked under specific conditions. *Oneof*[spec, gen] means that either the spec or the gen branch is invoked.

2.3.3 Possible choices for Gencol's parameters.

In this section, we indicate how Gencol can be instantiated along several dimensions, in order to familiarize the reader with the algorithm.

Languages and transformers

The choice of the languages and transformers is a very important one. The languages determine the knowledge representation of the problem and the form of the search space. The transformers specify the basic steps which the system can take in this search-space. When comparing concept-learning with graph-searching, the nodes are the strings in the language of concepts L_C and the arcs are determined by the transformers. In Gencol, we can use any language and transformer. Of course, different choices give rise to different algorithms.

The Initialization.

The function *Initialize* determines the starting point of the search for a concept. The initialization may yield one or more initial concept-descriptions. Systems that start at $\top$

Algorithm 2.1: Gencol

```
procedure Gencol
    Partial_Solutions := Initialize
    Complete_Solutions := ∅
    Memory := ∅
    while not Stop-Criterion (Partial_Solutions, Complete_Solutions) do
        ps := Select (Partial_Solutions)
        Partial_Solutions := Partial_Solutions − { ps }
        if Quality-Criterion (ps) is satisfied
        then Complete_Solutions := Complete_Solutions ∪ { ps }
        else if Optional
            then ex := Generate_Example(Partial_Solutions, Complete_Solutions)
            endif
            Oneof[
                spec: Partial_Solutions :=
                  Partial_Solutions ∪ Specialization-transformer (ps,ex),
                gen: Partial_Solutions :=
                  Partial_Solutions ∪ Generalization-transformer (ps,ex)
                ]
            Memory := Memory ∪ Remember (ps)
        endif
        Memory := Adapt (Memory)
        Partial_Solutions := Filter (Partial_Solutions)
    endwhile
endproc
```

and only specialize (e.g. the Model Inference System [Shapiro 83]) are often called *general-to-specific* systems. The dual kind of system, searching by generalizing from $\perp$, is known as a *specific-to-general system* (Clint of chapter 3 and [De Raedt and Bruynooghe 89e]).

The Stop-Criterion.

The *Stop-Criterion* specifies the conditions which terminate the search. By defining it appropriately it is possible to obtain one (perhaps the best), a few or all solutions.

The Quality-Criterion.

The *Quality-Criterion* characterizes complete solutions to the concept learning problem. If a partial solution satisfies these conditions it will be moved to the set of complete solutions which contains the result of the algorithm. Quality-criteria that can be used in Gencol include completeness and/or consistency [Mitchell 82], achieving at least a specified score and coverage on the examples [Buchanan and Mitchell 78], heuristically significant and most compact concept-descriptions [Quinlan 86], maximally general or maximally specific concept-descriptions [Dietterich and Michalski 83] and probably approximate correct concept-descriptions [Valiant 84]. The only requirement on the quality-criterion is that it is well-behaved with respect to the covers relation.

The search-strategy.

According to the function *Select* one obtains different search-strategies (cf.[Nilsson 80]): when *Select* is first_in_first_out Gencol performs a breadth-first search, for last_in_first_out a depth-first search and for maximization (or minimization) of a heuristic evaluation function (satisfying certain criteria cf. [Nilsson 80]) a best-first search; splitting Partial_Solutions into two dual versions results in a bidirectional search (see section 2.4.3) etc.

Another aspect of the search-strategy concerns the choice of the transformation to apply (i.e. the definition of *Oneof*). Some possibilities include: only generalization or specialization, preferring generalization over specialization (or vice versa), or a mixture of both. In many so-called data-driven algorithms *Oneof* depends on whether a negative example is covered or a positive example is not covered. In the former case, the concept-description is specialized while in the latter it is generalized.

Using Heuristics.

As the search space may be very large, it is often desirable to guide the search by means of domain-knowledge and heuristics. Such guidance can be built in Gencol in a variety of ways :

- *Select* : can use a heuristic evaluation function, selecting the most interesting partial solutions first.
- *Filter* : can be used to retain only the most promising partial solutions. If only a fixed number of partial solutions is retained, then a beam-search is realized (e.g. [Michalski 83]). *Filter* can also be used to discard partial solutions which contradict

a domain-theory (e.g. [Buchanan and Mitchell 78, Subramanian and Feigenbaum 86]).

- The transformers may also use knowledge or heuristics. Such heuristics often deal with the selection of the formula to be replaced when applying a derived transformer. For instance, when a concept-description is generalized in order to cover an example, this can be done by generalizing one of the formulas of the description. According to conceptual distance measures [Michalski and Stepp 83, Michalski 87] one may identify the best formula to be replaced.

The Memory.

Some systems, like the Model Inference System [Shapiro 81, Shapiro 83] store incorrect concept-descriptions in order to avoid infinite loops and considering unpromising concept-descriptions more than once. Others, like e.g. [Quinlan 86, Michalski 83], do not keep track of incorrect concept-descriptions. This behavior can be realized by defining *Remember* and *Adapt* appropriately.

Examples.

P and N are the sets of positive and negative examples. They may be given at the start or acquired incrementally. When they are acquired incrementally, the system and/or the user may generate additional examples at certain moments in time. P and/or N then have to be updated (this is not shown explicitly in the algorithm as it is trivial). *Generate_Example* may be used for the purpose of letting the system or the user generate an example. *Optional* then states the conditions under which this is done. However, *Generate_Example* is also used in connection with example-driven transformers. In this case, it initializes the example ex which is used in the example-driven transformers. The conditions are again specified by *Optional*. If the transformers are not example-driven then the value of ex is void.

2.4 Well-known instantiations of Gencol

In this section we show how Gencol can be instantiated to obtain a wide variety of systems. To the best of my knowledge, all algorithms that have a well-behaved *Quality-Criterion* and that search the space by applying generalization and specialization can be obtained. Evidence for this conjecture is presented by showing that Gencol instantiates to the following seemingly disparate algorithms: the TDIDT-family [Quinlan 86], the Model Inference System [Shapiro 81], the AQ methodology [Michalski 83] and the versionspace algorithm [Mitchell 77]. These systems were chosen because they cover a wide range of concept-learning approaches and because each of them is thoroughly described, well-known and has been successfully applied. During the presentation of these systems, I will take the freedom to omit technical details of minor importance. Furthermore, I only discuss these systems as instantiations of Gencol. A discussion of all aspects of these systems is outside the scope of this chapter. For more details I refer to the original papers.

2.4.1 Typical concept-learning systems.

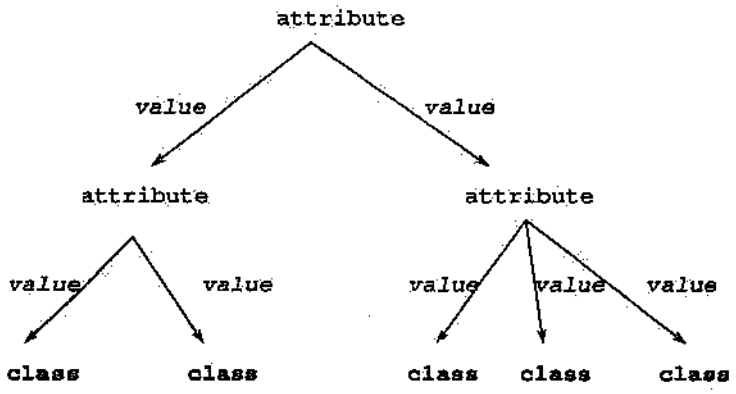
Typical concept-learning systems, like e.g. Rulegen and Rulemod [Buchanan and Mitchell 78], Sprouter [Hayes-Roth and McDermott 78], Thoth [Vere 75], Marvin [Sammut and Banerji 86], Clint [De Raedt and Bruynooghe 88, De Raedt and Bruynooghe 89e] and the algorithms in [Laird 86] and [Valiant 84] have a similar structure in the Gencol framework:

- they process one example at a time (all but Rulegen and Rulemod).
- they work general-to-specific (Rulegen) or specific-to-general (Sprouter, Thoth, Valiant's A and B algorithms, [Laird 86] and Clint) or apply both specialization and generalization (Marvin and Rulemod).
- most of the systems search breadth-first (all except Marvin, which uses depth-first search)
- they all use a variant of first order logic (all except Rulegen and Rulemod) in which an example corresponds to a conjunction and a concept-description to a disjunction of conjunctions or a conjunction.
- the transformers in most of these systems are example-driven operators (all except Rulegen and Rulemod)
- some systems like Marvin, Clint and Valliant's B algorithm generate their own examples
- in some systems all possible partial solutions are kept (as in [Laird 86]) while in Rulegen, Rulemod and Sprouter only the best ones are kept
- also the quality criterion differs: in Sprouter, Thoth, [Laird 86], Clint and Marvin it is completeness and consistency, while in Valiant's A and B algorithms it is pac-learning and in Rulegen and Rulemod local maximization of a heuristic scoring function

2.4.2 The Top-Down Induction of Decision Trees family.

The TDIDT-family of algorithms [Quinlan 86] induces decision trees from a given set of positive and negative examples. Examples consist of attribute-value pairs. A decision tree is shown in figure 2.4. It has attribute-based tests as nodes and for each possible outcome of the test there is a branch in that node. Leaves of the tree correspond to classes. In order to classify an example using a decision tree, one starts in the root of the tree and always takes the branch corresponding to the value for the attribute in the example, until one arrives at a leaf. The TDIDT-family induces trees by the following simple algorithm. It starts with all examples in the root of the tree. There it selects the best attribute to test upon according to statistical heuristics and then it constructs a branch for each value of the attribute and propagates all examples with that value to the node on the corresponding branch. If the node and the examples in it satisfy a *node-quality-criterion*, the node becomes a leaf with as label the class assigned by the node-quality-criterion. The node-quality-criterion states for instance that all examples in the node have to belong to the same class; this class is then taken as the label. Otherwise, the process is recursively

Figure 2.4: The structure of a decision tree.



applied on that node with the propagated examples. Nodes which still have to be handled will be called partial nodes. Nodes which are already handled are completed nodes.

The TDIDT-family of algorithms can easily be simulated by Gencol. One only has to instantiate Gencol as follows :

- L_C = the set of decision trees
- L_e = $\{e \mid e$ is a conjunction of attribute value pairs $\}$
- P and N are given initially and fixed
- *Quality-Criterion* (ps) = true iff all nodes in ps are completed
  The completion of all nodes implies that all leaves must satisfy the *Node-Quality-Criterion*. The *Node-quality-criterion* states e.g. that all examples in the node must belong to the same class (this results in complete and consistent decision trees) or that the node cannot be expanded in a statistically significant way (see [Quinlan 86]).
- *Stop-Criterion* (Partial_Solutions , Complete_Solutions) =
  | Complete_Solutions | = 1
- *Select* ({ps}) = ps
  Each time *Select* is called, the Partial_Solutions will be a singleton.

- *Oneof* (spec,gen) = spec
  The algorithm only specializes.
- *Optional* = false
  This implies that ex and *Generate_Example* are void.
- *Initialize* = { $\top$ }
  $\top$ is the tree consisting of one partial node
- *Generalization-transformer* (ps) = void
- *Specialization-transformer* (ps) =

```
select a partial node n of ps
if n satisfies the Node-quality-criterion
then turn n into a leaf by assigning a class to it
     n becomes completed
else for all attributes a that do not occur on the path from the root of ps to n do
          expand the node n using attribute a
          return the set of all these expansions
     endfor
endif
```

Expanding a partial node using an attribute a is done by testing on attribute a in that node and making a branch for each value of the attribute. The node n is completed and the new nodes are partial ones. The *Specialization-transformer* will expand one node of the tree. It will yield as many new trees as there are attributes which do not yet occur on the selected branch. A specialized tree tests on the value of one more attribute. It is not important which branch of the tree is selected for expansion, as every possible choice will eventually yield the same result.

Technical note : in order to obtain a real specialization C of a tree C', in the sense that $covers(C) \subseteq covers(C')$, it is necessary to define *covers* also for trees that contain incomplete nodes. The above specified transformer computes real specializations when *covers* is defined as follows : $covers(C) = \{ e \in L_e \mid$ when classifying e using C, e is not assigned the label *negative* $\}$; C may be an incomplete tree : nodes that are incomplete are treated as if they were *positive* leaves by default.

- *Remember* (ps) = $\emptyset$
- *Adapt* (Memory) = $\emptyset$
  So, nothing is remembered.
- *Filter* (Partial_Solutions) = { ps }, where ps is the best decision tree according to a heuristic that measures the heuristic gain of expanding a node of the tree. Several heuristics have been proposed (cf. [Gams and Lavrac 87, Quinlan 86]). This instantiation of Gencol simulates the basic TDIDT-algorithm and the extension for pre-pruning [Cestnik, Kononenko and Bratko 87]. It can also be extended to handle post-pruning [Cestnik, Kononenko and Bratko 87] and to simulate the method of [Quinlan 87].

2.4.3 The candidate elimination approach.

The versionspace method of [Mitchell 77, Mitchell 82] occupies a special position along the spectrum of concept-learning algorithms since it is the only one that searches bidirectionally. It has been successfully applied in the Meta-Dendral project [Buchanan and Mitchell 78] and the LEX-project [Mitchell, Utgoff and Banerji 83]. The versionspace method is to a large degree independent of the description languages L_e and L_C although it only works well when L_C does not allow for external disjunction (see [Murray 87]). Hence, our presentation of the versionspace method is language independent.

Roughly speaking, external disjunction (see [Michalski 83]) allows for disjunction among various values for the same attribute, whereas internal disjunction also allows for disjunction among the values of values for different attributes. As an example, reconsider example 2.1, where the string `4-sided` $\wedge$ `red` allows for internal disjunction (`4-sided` actually means `rectangle` $\vee$ `square` $\vee$ `trapezoid`) but the string (`triangle` $\wedge$ `red`) $\vee$ (`square` $\wedge$ `pink`) employs external disjunction. All formula's that use internal disjunction can be rewritten as using external disjunction but the opposite does not hold. External disjunction is more expressive than internal disjunction.

The key idea behind the versionspace algorithm is to keep track of the sets S and G, defined below. S and G together determine the set of solutions to the problem. Indeed, any description lying between S and G is complete and consistent.

Definition 2.23 $S = \{ s \mid s \in L_C$ *and* s *is complete and consistent w.r.t. the observed examples, and there is no concept-description which is both strictly more specific than* s *and complete and consistent w.r.t. the observed examples* $\}$

Definition 2.24 $G = \{ g \mid g \in L_C$ *and* g *is complete and consistent w.r.t. the observed examples, and there is no concept-description which is both strictly more general than* g *and complete and consistent w.r.t. the observed examples* $\}$

The sets S and G are modified each time a new example is encountered. To update S and G, when the new example e is negative, only those elements that do not cover e are retained in S; the elements of G that cover e are specialized such that they no longer cover e and still remain more general than some element in S and finally all redundant elements of G (those that are strictly more specific than another element of G) are removed. Updating for a positive example is completely dual (cf. [Mitchell 82]).

To obtain the versionspace algorithm in Gencol, one has to divide the set of Partial_Solutions and the set of Complete_Solutions into two sets: one corresponding to G and one to S. For some functions in Gencol, this also results in two different versions, one that is invoked on the S and one on G.

- L_C is a language that allows for conjunction but not for external disjunction [Michalski 83]
- L_e is a subset of L_C
- P and N are usually acquired incrementally; examples can be given by the user and in some versions [Subramanian and Feigenbaum 86] generated by the system.
- *Quality-Criterion* (ps) = true iff ps is complete and consistent with all examples The quality criterion can only be true when all examples have been acquired.

- *Stop-Criterion* (Partial_Solutions , Complete_Solutions) =
  | Partial_Solutions-S | = 0 and | Partial_Solutions-G |= 0

  There are two cases to consider :

  1. | Complete_Solutions-S | = 0 or | Complete_Solutions-G | = 0
     in this case there are no solutions to the problem
  2. | Complete_Solutions-S |≠ 0 and | Complete_Solutions-G |≠ 0
     this implies that all $c \in L_C$ for which there is a ps in Complete_Solutions-S and a pg in Complete_Solutions-G such that c is more general than ps and less general than pg, are solutions to the problem.

- *Select* (Partial_Solutions-S ∪ Partial_Solutions-G) = first_in_first_out
  For each example, all partial solutions are selected exactly once (this is also important for *Optional* and *Filter*). In case the example is positive, the elements of Partial_Solutions-G are selected before the elements of Partial_Solutions-S. For a negative example the order is the reversed one.

- *Oneof* (spec,gen) =

  if ps ∈ Partial_Solutions-S
  then gen
  endif
  if ps ∈ Partial_Solutions-G
  then spec
  endif

- *Optional* = true iff all strings that were elements of Partial_Solutions-S and Partial_Solutions-G at the time of the previous invocation of *Optional* have been selected once

- *Generate_Example* (Partial_Solutions-S ∪ Partial_Solutions-G,
  Complete_solutions-S ∪ Complete_Solutions-G)

  An example is generated if the treatment of the previous example is finished. In the basic versionspace algorithm, examples are supplied by the user; in some variants [Subramanian and Feigenbaum 86] they are generated by the system.

- *Initialize-S* = { ⊥ }
  Initialize-G = { ⊤ }
  Both functions are invoked at the start of the algorithm. Additionally, ps is assigned the first example, as in Mitchell's version of this algorithm.

- *Generalization-transformer* (ps,ex) =

  if ps covers ex
  then if ex is positive

```
        then return {ps}
        else return ∅
        endif
    else if ex is positive
        then return Example-driven generalization operator(ps,ex)
        else return {ps}
        endif
    endif
```

- *Specialization-transformer* (ps,ex) =

```
if ps covers ex
then if ex is positive
        then return {ps}
        else return Example-driven specialization operator(ps,ex)
        endif
        if ex is positive
        then return ∅
        else return {ps}
        endif
endif
```

- *Adapt* (Memory) = ∅
  Remember (ps) = ∅
  So, nothing is remembered in the versionspace technique. One may even forget the already treated examples.

- *Filter-S*(Partial_Solutions-S) =

```
if all strings that were elements of Partial_Solutions-S and Partial_Solutions-G
   at the time of the previous invocation of Optional have been Selected once
   and ex is positive
then return Partial_Solutions-S - { ps ∈ Partial_Solutions-S |
          there is a pg in Partial_Solutions-G ∪ Complete_Solutions-G
          which is strictly less general than ps or
          there is a ps' in Partial_Solutions-S ∪ Complete_Solutions-S
          which is strictly more specific than ps }
else return Partial_Solutions-S
endif
```

- *Filter-G*(Partial_Solutions-G) =

```
if all strings that were elements of Partial_Solutions-S and Partial_Solutions-G
   at the time of the previous invocation of Optional, have been Selected once
```

```
    and ex is negative
then return Partial_Solutions-G - { pg ∈ Partial_Solutions-G |
        there is a ps in Partial_Solutions-S ∪ Complete_Solutions-S
        which is strictly more general than pg or
        there is a pg' in Partial_Solutions-G ∪ Complete_Solutions-G
        which is strictly more general than pg }
else return Partial_Solutions-G
endif
```

2.4.4 The AQ methodology using Induce.

[Michalski 83] describes the AQ concept-learning algorithm. It uses the Induce program [Michalski 83] as a subroutine. It has been used to learn diagnostic rules for soy-bean diseases [Michalski and Chilausky 80]. Although there are many versions of the system we will consider the one described in [Michalski 83] on pages 112 -115, and will present only a simplification without truth-preserving transformations or constructive induction. These features are omitted since they are not essential to the algorithm as an instantiation of Gencol and presenting these features would make the presentation less readable and understandable.

The AQ algorithm learns a complete and consistent concept-description from a given (and fixed) set of positive and negative examples. Concepts are represented in Annotated Predicate Calculus, which is a variable-valued logic language [Michalski 75]. For our purposes, it suffices to regard them as variants of first order logic in which concepts are represented by a disjunction of conjunctions and examples by conjunctions. AQ learns as follows: it selects a positive example and invokes Induce in order to generate a number of maximally general formulas which cover the positive example and which do not cover negative examples; then it selects the best formula in the result of Induce according to a lexicographic evaluation function (LEF) and adds it as a disjunct to the current concept-description; if the current concept-description is complete AQ stops; otherwise it generates a positive example that is not covered by the current concept-description and restarts.

Induce learns as follows: it first uses the positive example to define a specialization-operator, which refines by adding conjuncts to a formula. The conjuncts are added in a heuristic order (the most interesting first) and only conjuncts which are of lower preference than conjuncts already in the formula are added (this to avoid generating the same formulas more than once); the algorithm starts at $\top$ and refines those formulas which are not consistent. Also, all consistent concept-descriptions are locally generalized in order to be as general as possible, while remaining consistent. During the search Induce only keeps track of the m best formulas and searches for n_1 complete and consistent solutions or for n_2 consistent solutions. m, n_1 and n_2 are parameters of Induce.

In order to obtain this algorithm in Gencol, it is convenient to define also two different algorithms in Gencol: one for Induce and one for AQ.

The AQ method.

- L_C = Annotated Predicate Calculus [Michalski 83]

- L_e = the conjunctive statements (c-expressions) in Annotated Predicate Calculus [Michalski 83]
- P and N are given initially and fixed
- *Quality-Criterion* (ps) = true iff ps is complete and consistent
- *Stop-Criterion* (Partial_Solutions , Complete_Solutions) = | Complete_Solutions | = 1
- *Select* ({ps}) = ps
  each time *Select* is called, Partial_Solutions is a singleton.
- *Oneof* (spec,gen) = gen
  AQ always generalizes.
- *Optional* = true
- *Generate_Example* (Partial_Solutions , Complete_Solutions)) yields a positive example which is not covered by ps.
- *Initialize* = { $\perp$ }
- *Generalization-transformer* (ps,ex) = Induce is called and { ps $\vee$ f | f is a formula returned by Induce } is returned.
  So, the *Generalization-transformer* is derived from Induce (a formula-transformer).
- *Specialization-transformer* (ps,ex) = void
- *Remember* (ps) = $\emptyset$
  Adapt (Memory) = $\emptyset$
  So, nothing is remembered.
- *Filter* (Partial_Solutions) = {ps} where ps is the best element of Partial_Solutions according to the LEF

The Induce Method.

- L_G = a subset of the Annotated Predicate Calculus without external disjunction [Michalski 83]
- L_e = the conjunctive statements (c-expressions) in Annotated Predicate Calculus [Michalski 83]
- P and N are given initially and fixed; P is a singleton $\{p\}$
- *Quality-Criterion* (ps) = true iff ps is consistent and locally maximal, i.e. it cannot be generalized without becoming inconsistent
- *Stop-Criterion* (Partial_Solutions , Complete_Solutions) = true iff there are n_1 complete and consistent solutions on Complete_Solutions or there are n_2 consistent solutions on Complete_Solutions

- *Select* (Partial_Solutions) = best_first according to some heuristic
- *Oneof* (spec,gen) =

  if ps is consistent
  then gen *(in order to find a local optimum)*
  else spec *(in order to find a consistent description)*
  endif

- *Optional* = false
  This implies that ex and *Generate_Example* are void.
- *Initialize* = { $\top$ }
- *Generalization-transformer* (ps,ex) : some of Michalski's generalization rules are applied [Michalski 83]
- *Specialization-transformer* (ps,ex) = { ps $\wedge$ c | c is a literal in ex of lower preference than predicates already occurring in ps }
- *Remember* (ps) = $\emptyset$
  Adapt (Memory) = $\emptyset$
  So, nothing is remembered.
- *Filter* (Partial_Solutions) = the set of m best elements of Partial_Solutions

  In this way a beam-search is realized. The best formulas are those that score best on the LEF. A typical LEF is the maximization of the number of covered positive examples and minimization of the complexity of the expression.

It turns out that many algorithms use a variant of the AQ algorithm to learn disjunctive concepts. For instance the Model Inference System [Shapiro 83], Clint [De Raedt and Bruynooghe 88, De Raedt and Bruynooghe 89e] and Marvin [Sammut and Banerji 86] can be seen as variants of AQ: indeed, they all learn conjunctive formulas by invoking a special algorithm. In the Model Inference System, this is done by refining from $\top$ and in Marvin and Clint by generalizing from $\bot$. The result of the special algorithm is then added as a disjunct to the current concept-description.

2.4.5 The Model Inference System.

[Shapiro 81, Shapiro 83] describes the Model Inference System. He uses it to synthesize Prolog-programs [Lloyd 87, Sterling and Shapiro 86, Bratko 86] from examples. Shapiro uses the Model Inference System with various subsets of Prolog and several transformers. Each concept-description language described by Shapiro allows a description to be composed of several clauses, allowing for disjunctive concept-descriptions. Here we will make abstraction of the languages.

The basic Model Inference System works (cf. algorithm 2 in [Shapiro 81]) as follows. It always has a current theory to explain the already handled examples. When the user

supplies an additional example, the current theory is modified if it incorrectly handles the example. For a covered negative example, a guilty clause is identified and retracted using the contradiction backtracing algorithm. Clauses covering a negative example are stored in the Memory. If there is a positive example which is not covered by the current theory, the most general clause is added to the theory. This clause usually covers negative examples (it will be in the Memory) and is therefore refined in a next step. The Model Inference System only refines clauses which are in the Memory. Since refinements of clauses in the Memory, may yield other clauses of the Memory, refining is done recursively until a clause is obtained that is not in the Memory and that covers the positive example. Of course, after each modification (assertion or retraction of a clause), the system must verify all previous examples in order to guarantee completeness and consistency. Using Gencol one can easily simulate the Model Inference System as follows :

- L_C = a subset of Prolog; concept-descriptions may consist of several formulas and a formula corresponds to a Horn-clause
- L_e = the facts of L_C
- P and N are acquired incrementally; the examples may be given by the user. For certain operators, the system generates some examples (cf. [Shapiro 83]).
- *Quality-Criterion* (ps) = true iff ps is complete and consistent with all examples. The *Quality-Criterion* can only be true when all examples have been acquired.
- *Stop-Criterion* (Partial_Solutions , Complete_Solutions) =
  | Complete_Solutions | = 1 ;
  the result of the search is one complete and consistent concept-description
- *Select* (Partial_Solutions) = first_in_first_out
  The Model Inference System performs a breadth-first search on concept-descriptions (actually on formulas cf. the *Generalization-transformer*).
- *Oneof* (spec,gen) =

```
if ps covers ex and ex is negative
then   spec
endif
if ps does not cover ex and ex is positive
then   gen
endif
```

- *Optional* =

```
if  ex is handled correctly by ps
then return true
else  return false
endif
```

- *Generate_Example* (Partial_Solutions , Complete_Solutions)=

  if ps is complete and consistent w.r.t. the known examples
  then the user is asked for a new example
  else one of the known examples is selected
  endif

  The else branch is used to retain completeness and consistency w.r.t. the previous examples; the transformers are not example driven, so ex is void in the transformers;

- *Initialize* = $\{\emptyset\}$
  The Partial_Solutions are initialized with the empty theory $\emptyset$.

- *Generalization-transformer* (ps) = $\{ ps \vee \top \}$
  the most general clause is added to the theory, i.e. the transformer is derived from the null formula transformer; since the result is too general, it will be refined in the next step by applying the formula specialization operator on $\top$; during the search, all concept-descriptions on Partial_Solutions are identical except for one clause and it is this clause which is refined in order not to cover negative examples; each time this transformer is applied the Partial_Solutions is a singleton.

- *Specialization-transformer* (ps,ex) =

  if ps contains a clause c of the Memory *(cf. Remember)*
  then return the result of applying a formula-specialization operator on c
  else return {ps - {c'}}; where c' is a clause responsible
  for the coverage of the negative example ex
  endif

  When an incorrect clause (a clause covering a negative example) is encountered for the first time it is stored in the Memory; when it is generated a second time, it will be refined. In this way refinements of previously refuted hypotheses are added to the theory. Notice that this transformer is not example driven since the formula-specialization operator is not (it does not always yield clauses that do not cover the negative example ex).

- *Adapt* (Memory) = Memory
  everything which is stored in the memory, is remembered;

- *Remember* (ps) = { c' | c' is a formula of ps that covers a negative example}
  So, the already encountered formulas that cover negative examples are remembered. This allows to avoid infinite loops and determines the starting points of the specialization transformer (cf. also the specialization transformer).

- *Filter* (Partial_Solutions) =

```
if s ∈ Partial_Solutions and s covers the last positive example ex
   and no negative ones
then return {s}
else  return { s ∈ Partial_Solutions | s covers the last positive example ex }
endif
```

If there is more than one such s in the then-part then the first one is chosen. The reason for defining *Filter* in this way is that one searches for a description covering the (last) positive example. In the then branch, the system has found such a description. In the else branch one can safely discard the descriptions that do not cover the example, since refinements of these descriptions will never cover the example.

2.5 Applying Gencol in practice.

As part of the undergraduate thesis of Erwin Declercq [Declercq 90], a tool to aid in the design of concept-learning systems was developed. The tool was based on a straightforward implementation of Gencol in Prolog. To obtain a real tool, facilities were provided to choose the generic parameters in a declarative way:

- A logic language based on Prolog was used as the representation mechanism for L_e and L_C. Different classes of description languages can be obtained by appropriately choosing certain parameters. Indeed, it is possible to state whether a concept-description:
  - may be disjunctive or conjunctive
  - may use tail-recursion, recursion or no recursion
  - may use local variables or not (see also chapter 4 for this property of languages)
  - may use a form of explicit negation as in [De Raedt and Bruynooghe 90c] and chapter 7
  - may use functors and if so, which ones
- Once appropriate languages are defined, the user can specify which transformer(s) should be used. The user only needs to declare the kind of transformer she wants to use. For example, she can state that a generalization transformer based on the null formula transformer should be applied. The available transformers correspond basically to the ones discussed above.
- The tool can cope with mode and type information (see also chapter 3) concerning predicates that may be used in concept-description languages. This allows to optimize the search.
- The system uses a depth-bounded interpreter in order not to run into infinite loops while verifying the coverage of certain examples.

- The search strategy can be chosen : depth-first, breadth-first, beam-search or best-first. In the last two cases a lexicographic evaluation function [Michalski 83] is used to identify the best elements. A default LEF is built-in but can easily be modified.
- The conditions under which to specialize, to generalize, to discard a description from the list of partial solutions (*Filter*), to satisfy the stop and quality criterion, can be assembled using the following primitives :
  - consistent
  - complete
  - maximally general
  - maximally specific
  - the best one(s) according to the LEF
  - minimally disjunctive
  - covers (resp. does not cover) a positive (resp. negative) example; the example is a parameter, e.g. ex .

  The primitives can be used in conjunction with each other. It is e.g. possible to state that all descriptions that are not minimally disjunctive or cover the negative example ex have to be discarded from the list of partial solutions. The formalism to specify these conditions is quite friendly to the user.

When the builtin formalism and/or facilities are not sufficient to specify a desired algorithm, the user can easily modify certain parameters by creating Prolog code for these parameters.

The tool has been used to obtain a simple implementation of the algorithms discussed above: versionspaces, AQ, and a simple version of the Model Inference System and TDIDT. This provides clear evidence for the claim that Gencol can be instantiated to these algorithms and for the generality and usefulness of Gencol as a framework for concept-learning.

Secondly, the tool can be used for experimenting with variants of a particular algorithm. This is useful for the development of practical algorithms w.r.t. certain applications.

2.6 Related work.

Gencol builds further on the work of [Shapiro 81, Shapiro 83, Laird 86, Mitchell 82, Gams and Lavrac 87, De Raedt, Krekels, Bruynooghe and Van Meir 87] by gradually abstracting more features of concept-learning.

The presented work is related to all surveys of concept-learning. This includes: [Biermann 86, Mitchell 82, Kodratoff 88, Michalski 83, Angluin and Smith 83, Shapiro 81, Shapiro 83, Laird 86, Dietterich and Michalski 83, Cohen and Feigenbaum 81, Rendell 86, Gams and Lavrac 87, Bundy, Silver and Plummer 85]. However the focus of all these approaches is different: [Cohen and Feigenbaum 81, Biermann 86] give a didactical survey of the field, [Dietterich and Michalski 83, Bundy, Silver and Plummer 85, Angluin and Smith 83] compare different approaches, [Rendell 86, Michalski 83] present general frameworks and theories for concept-learning, [Michalski 83] also presents the AQ program

and the other papers, [Gams and Lavrac 87, Shapiro 81, Shapiro 83, Laird 86, Mitchell 82], extract underlying principles of concept-learning and attempt to formulate a general algorithm that accounts for these principles. Shapiro presents his Model Inference System and demonstrates it at work on two different concept-learning tasks: he makes abstraction of the description languages and the operators; Laird focuses on a semantic theory of concept-learning, which allows him to formulate a general bottom-up system which is roughly equivalent to a version of Gencol, that starts from $\perp$ and uses only generalization by means of a generalization operator; and Gams and Lavrac implemented a practical algorithm in order to be able to test different heuristic criteria for concept-learning: their algorithm can make abstraction of the AQ and TDIDT family since it is model-driven; Mitchell compares and characterizes different algorithms in terms of their search-strategy: he defines an algorithm that searches breadth-first, one that searches depth-first and also the versionspace method.

Gencol goes significantly further than the approaches mentioned above, since it makes abstraction of the whole spectrum of concept-learning algorithms that use generalization and specialization, rather than just a part of that spectrum. Moreover, Gencol is more detailed than the general algorithms given by Shapiro, Laird and Gams and Lavrac. As opposed to Shapiro, Laird and Mitchell, Gencol also allows for quality-criteria which are different from completeness and consistency. The three algorithms in [Mitchell 82] are combined into a more general one, that can be instantiated to algorithms that are not instantiations of Mitchell's ones (e.g. the AQ program). Similarly, Gencol can be instantiated to obtain systems which cannot be obtained with the Model Inference System, or the algorithms of Laird and Gams and Lavrac. Laird's algorithm does not allow the use of generalization as well as specialization operators; Gams and Lavrac's algorithm assumes that all examples are given initially, that learning starts at $\perp$ and that the languages only allow attribute-value pairs; and the Model Inference System is an incremental algorithm, and its search-strategy uses a set of Partial_Solutions, whose elements are identical except for one formula because the transformer is a derived one. Also it assumes that the *Generalization transformer* is the one derived from the null formula operator.

2.7 Conclusions.

This chapter is a slightly extended and revised version of [De Raedt and Bruynooghe 89a]. It has been shown that the formulation of Gencol is an interesting contribution to the theory and practice of concept-learning since it provides a unifying framework for concept-learning algorithms. Gencol builds further on Mitchell's *Generalization as search*-paradigm as it makes abstraction of this search-strategy.

In contrast to most previous surveys of concept-learning we have synthesized a wide variety of concept-learning systems into Gencol. Gencol does not only serve synthetic or didactic purposes but also analytic and practical ones. It is useful for analysis, because it offers a general framework in which concept-learning systems can be compared and evaluated. Also, by reformulating systems in Gencol-terms, one can easily appreciate the differences and similarities between different approaches. Furthermore, Gencol forms the basis of a tool that aids in the design of novel concept-learning algorithms or applications. One only has to choose Gencol's parameters appropriately w.r.t. the application. In this way new and interesting systems can be derived from Gencol.

Chapter 3

Interactive Concept-Learning.

3.1 Introduction.

This chapter presents the basic algorithms of Clint in a logical framework for concept-learning. As Clint is designed for use in the learning apprentice context, its basic algorithms take into account some specific requirements for learning apprentices:

- The knowledge base of the learning apprentice should be used and adapted during the learning process.
- The learning process should be *incremental*, i.e. examples should be processed one by one, as they are supplied to the system.
- A general requirement for intelligent systems and therefore also for learning apprentices is *reliability*.
- The learner should be *interactive*. Within a concept-learning context, an important form of interaction is the generation of examples. Furthermore, interactive systems should be easy to communicate with. An implication is that examples should be formulated in a simple manner.

These requirements are direct consequences of the definition and the user-friendly and opportunistic nature of a learning apprentice. The first three requirements are also present in the related *sloppy modeling* paradigm [Morik 89] for knowledge acquisition.

A prerequisite for the first requirement to apply, is that the learner employs the same representation formalism as the underlying knowledge based system. It has often been argued in the literature [Genesereth and Nilsson 87, Nilsson 80, Kowalski 79] that logic and logic programming [Lloyd 87, Kowalski 79, Sterling and Shapiro 86, Bratko 86] provide an outstanding representation formalism for knowledge based systems and artificial intelligence. As logic and logic programming also have a well understood semantics, which is very important for concept-learning, they were choosen as the representation formalism of Clint. Another important demand, following from the first requirement, is the need for a *closed-loop system* [Michalski 87]. Closed-loop systems are learners that use and assimilate previously learned concepts. Closed-loop learning means that one should not only study how to learn one concept but also how to learn a set of concepts.

The second requirement is incrementality as it does not make sense to ask the user to accumulate her examples until she thinks there are enough examples to let the system

learn. Rather, it must be possible to supply examples to the system at any time and by doing so pinpoint errors in the system's knowledge base.

The minimal condition for a system to be reliable, is to have a clear idea of the system's abilities and limitations. It is with this in mind that we will prove certain properties about Clint's convergence. Doing so will also reveal some remaining problems with Clint when complex sets of concepts are learned. Seen in this light, I believe that reliability has not received the attention it deserves in machine learning.

The fourth requirement, interaction, is certainly one of the most important ones for learning apprentices. Interactive learners generate their own examples and allow the user to express examples in a simple formalism.

Requiring the user to supply the complete training set of examples is not feasible in the learning apprentice context as this is a difficult and time-consuming task for the user. Basically, the user has to address the following two questions:

1. *Which examples provide relevant new information to the system ?*
2. *When does the training-set contain enough information to learn a correct concept-description ?*

Both questions are very hard to answer because of a fundamental and practical problem. A fundamental problem[1] with these questions is that the user only has limited knowledge about the concept she wants to learn. (If the user would have complete knowledge, there is no need for learning it.) On the other hand, even if the user would have complete knowledge about the concept, it would still be very hard to answer the questions because this requires a firm understanding of the system's learning procedures. To answer the questions one has to relate the examples to each other. Because the questions are so hard, most users do not address them. They merely supply a training-set to the learner and analyze the result of the learner. This is a purely ad hoc method and undesirable because it may result in unnecessary work: if the result of the learner is satisfactory, the user may have supplied too many examples and unnecessary work may have been performed to collect the examples. If, on the other hand, the result is not satisfactory, the reason may be that the training-set did not contain enough information. Then the user may attempt to improve the results by supplying more examples to the system. There is – however – no guarantee that better results will be obtained using a larger training-set.

The task of the user becomes much simpler if the learner generates the examples it needs and asks the user to classify them. This also allows to avoid unnecessary work. For the learner, it is often easy to decide which examples are relevant and when enough information has been received to produce a good concept-description. As an example, consider versionspaces [Mitchell 77, Mitchell 82, Subramanian and Feigenbaum 86]. In versionspaces, an example is relevant if it is not covered by all elements in the S-set and covered by at least one of the elements in the G-set. The versionspace algorithm has converged on a solution and does not need additional examples when the G-set is a singleton that is identical to the S-set.

Clint was also designed to provide the user with an easy way to formulate examples. In the learning apprentice context, it is not acceptable that the user has to explicitly list the features which make an example an instance of a concept, as required

[1] This observation was made by Yves Didier Willems.

by many classical systems such as the TDIDT family [Quinlan 86] and Marvin [Sammut and Banerji 86]. Formulated in logic this means that examples are ground clauses, e.g. `father(etienne, luc)` ← `male(luc)` ∧ `male(etienne)` ∧ `parent(etienne, luc)` ∧ `married(rose, etienne)`. For learning apprentices this requirement is unacceptable, as this knowledge (`male(luc)`, `male(etienne)`, `parent(etienne, luc)`, `married(rose, etienne)`) is in the knowledge base. Therefore examples should be ground facts such as `father(etienne, luc)` and further knowledge should be derived from the knowledge base. The issues involved in the representation of examples as ground facts will be elaborated in chapter 4.

This chapter is organized as follows. In section 3.2, the problem of incremental and interactive concept-learning is formalized within a logical framework. To this aim, problem statement 2.1 is adapted towards logic and the specific needs of learning apprentice systems. In section 3.3, two abstract concept-learners are defined within the logical framework: $\mathcal{VS}$ for learning conjunctive concepts and $\mathcal{DVS}$ for learning disjunctive ones. These abstract concept-learners serve as a simple formalism to reason about concept-learning and make it possible to present certain techniques as the shift of bias in chapter 4 and the multi-valued logic in chapter 7, independently of Clint. The abstract concept-learners are also used to demonstrate the need for biases in concept-learning. This is done in section 3.4. The basic algorithms of Clint are presented in section 3.5. Their theoretical properties are analyzed in section 3.6. It is shown that — under certain conditions — Clint identifies concepts in the limit. The use of the basic algorithms for program synthesis from examples will also be studied. The major problem for program-synthesis is the halting problem, which is in general undecidable. Given a perfect query answering procedure (cf. infra), the basic algorithms of Clint can be used for program-synthesis (section 3.7). The complexity of Clint is studied in section 3.8 and in section 3.9 it is shown that Clint is an instantiation of Gencol. Section 3.10 contains some conclusions and related work.

3.2 A logical framework for concept-learning.

In this section, the problem of concept-learning is formalized in a logical framework while taking into account the requirements imposed by the learning apprentice context.

In chapter 2, a concept-learning problem (problem statement 2.1) was defined as finding a concept-description $c \in L_C$ that satisfies the quality-criterion Q, from the language of examples L_e, the language of concept-descriptions L_C, the *covers* relation between L_e and L_C, the quality-criterion Q and the sets of positive and negative examples P and N. As the learner has to use the available knowledge, it is also assumed that there is a knowledge base KB available. To reformulate concept-learning in terms of logic, it is sufficient to define P, N, Q, L_e, L_C and *covers* in a logical framework, while taking into account KB.

This motivates the following definitions:

Definition 3.1 *A concept is a predicate.*

The intended interpretation τ is used to denote truth and falsity in the target knowledge base. The target knowledge base is the desired knowledge base, which contains correct definitions for all learned concepts. We will use two different versions of τ : τ_2 and τ_M. τ_2 is used to denote the intended interpretation in the normal two-valued framework.

τ_M will be used in chapter 7 to obtain a multi-valued interpretation. When there is no subscript in τ, we make abstraction of the particular intended interpretation being used. This is done to avoid the repetition of certain definitions.

Definition 3.2 *τ denotes the intended interpretation. The notation $\tau(f) = t$ denotes that t is the truthvalue of the logical formula f in the intended interpretation τ. In the two-valued framework, t = true or t = false, whereas in the multi-valued framework, t can also be unknown and inconsistent.*

The multi-valued intended interpretation τ_M will be further explained in chapter 7.

Definition 3.3 defines the language of examples L_e.

Definition 3.3 *An example is a ground fact. Positive examples are true and negative examples are false in the intended interpretation. If $\tau(p(t_1,...,t_n))$ = true then $p(t_1,...,t_n)$ is a positive example for p w.r.t. τ; if $\tau(p(t_1,...,t_n))$ = false, it is a negative example for p w.r.t. τ.*

Definition 3.4 *(see also [Genesereth and Nilsson 87, Sterling and Shapiro 86]) A clause is the disjunction of a set of literals. A literal is an atom $p(x_1,...,x_n)$ or the negation of an atom $\neg p(x_1,...,x_n)$ in first order logic.*

Definition 3.5 *A Horn-clause is a clause that contains at most one positive literal.*

Horn-clauses correspond to the definite clauses in Prolog [Sterling and Shapiro 86, Bratko 86, Lloyd 87]. A Horn-clause c can be written as an implication $p \leftarrow q_1 \wedge ... \wedge q_k$ or as a disjunction $p \vee \neg q_1 \vee ... \vee \neg q_k$, where p is the atom of the positive literal in c and the q_i are the atoms of the negative ones in c. The positive literal p will be denoted by *head*(c) and the goal consisting of the negative literals $\leftarrow q_1 \wedge ... \wedge q_k$ by *body*(c). *Body*(c) will often be used as a set of literals and sometimes as a goal. Roughly speaking, the semantics of Horn-clause c are such that *head*(c)θ is true for some substitution θ whenever all literals in *body*(c)θ are true. In this way, the body forms a sufficient set of conditions for the head. For more details concerning logic and logic programming, see [Genesereth and Nilsson 87, Lloyd 87, Sterling and Shapiro 86].

Definition 3.6 *A concept-description of a predicate p is a set of Horn-clauses with as head a positive literal for p.*

This defines L_C. In the sequel, we will impose certain syntactic restrictions on the form of Horn-clauses in a concept-description (see section 3.4.2). To achieve this, specific languages will be defined (see chapter 4). Because the language of examples L_e remains the same throughout this thesis but different concept-description languages are used, we will sometimes delete the subscript C in L_C and talk about language rather than concept-description language when the context is clear.

Definition 3.7 *A knowledge base is a set of Horn-clauses.*

φ is defined to make abstraction of the semantics.

Definition 3.8 *$\varphi(KB, e)$ denotes the truthvalue of the ground fact e in the knowledge base KB under the semantics φ.*

As for the intended interpretation τ, we will use two different versions of φ : φ_2 and φ_M. φ_2 is used to denote the usual two-valued minimal Herbrand model semantics. φ_M will be defined in chapter 7 and will be used to obtain a multi-valued logic. When there is no subscript in φ, we make abstraction of the semantics.

In this thesis, we will mostly use τ_2 and φ_2. It is only in chapter 7 that τ_M and φ_M will be used. Most of the definitions that follow are specific to the two-valued framework. They will – however – also be of use for the multi-valued framework because the mult-valued framework will be translated into the two-valued one (see chapter 7).

φ_2 is defined as follows :

Definition 3.9 *$\varphi_2(KB, e) = true$ iff $KB \models e$ and $\varphi_2(KB, e) = false$ iff $\neg(KB \models e)$.*

The notation $KB \models e$ denotes that e is true in the minimal Herbrand model of the knowledge base KB (cf. [Lloyd 87]). Roughly speaking, it means that e is implied by KB.

The definition of φ_2 may give some problems. The reason is that the problem of determining whether $KB \models e$ is only semi-decidable. Therefore $KB \models e$ should be replaced by $KB \vdash e$ in practice. There are three possible outcomes of $KB \vdash e$: true, if $\leftarrow e$ is provable; false, if $\leftarrow e$ finitely fails and undefined, when $\leftarrow e$ loops. In the remainder of this thesis, however, I will act as if there are only two possible truthvalues for φ_2. The reason is that

- we will assume that there is a perfect query answering procedure, which always terminates (see section 3.7)
- a large part of the thesis assumes that Datalog, i.e. Prolog without functors (see later), is used. For Datalog, φ_2 is decidable.

A knowledge base satisfies an example when the semantics of the example in the knowledge base agree with the intended interpretation. Otherwise, it violates the example.

Definition 3.10 *A knowledge base KB satisfies an example e iff $\varphi(KB, e) = \tau(e)$.*

Definition 3.11 *A knowledge base KB violates an example e iff $\varphi(KB, e) \neq \tau(e)$.*

Definition 3.12 *An example $p(t_1, ..., t_n)$ is covered by a concept-description C for p w.r.t. a knowledge base KB (not containing clauses for p) iff $\varphi_2(KB \cup C, p(t_1, ..., t_n)) = true$. This will be denoted as $covers_{KB}(C, p(t_1, ..., t_n))$.*

Definition 3.13 *An example $p(t_1, ..., t_n)$ is covered by a clause $c \in C$, where C is a concept-description for p, w.r.t. a knowledge base KB (not containing clauses for p) iff $head(c)\theta = p(t_1, ..., t_n)$ for some substitution θ such that $c\theta$ is ground and $\varphi_2(KB \cup C, l) = true$ for all $l \in body(c)\theta$. This will be denoted as $covers_{KB}(c, p(t_1, ..., t_n))$.*

Definition 3.14 *A concept-description C is complete w.r.t. a knowledge base KB (not containing clauses for the concept) and a set of examples E iff $\forall e \in E$ such that $\tau_2(e) = true$: $\varphi_2(KB \cup C, e) = true$.*

Definition 3.15 *A concept-description C is consistent w.r.t. a knowledge base KB (not containing clauses for the concept) and a set of examples E iff $\forall e \in E$ such that $\tau_2(e) = false$: $\varphi_2(KB \cup C, e) = false$.*

The quality-criterion Q used by Clint is completeness and consistency. Clint attempts to derive a knowledge base that is complete and consistent w.r.t. all examples, or equivalently, a knowledge base that satisfies all examples.

The following notions will be used to reason about correctness of the learned knowledge w.r.t. the target knowledge base in the two-valued framework.

Definition 3.16 *A clause c is correct iff $\tau_2(c) = true$.*

Notice that the correctness of a clause depends only on the intended interpretation and not on the knowledge base!

Definition 3.17 *A concept-description C is correct for a predicate p iff*

1. *all clauses in C are correct and*
2. *for all examples $p(t_1,...,t_n)$ such that $\tau_2(p(t_1,...,t_n)) = true$, there is a clause $c \in C$ such that $\exists\theta$: $head(c)\theta = p(t_1,...,t_n)$, $c\theta$ is ground and $\tau_2(body(c)\theta)$ is true.*

In this thesis, *correctness* will be used to reason about truth in the intended interpretation whereas *completeness* and *consistency* will be used to reason about the behaviour of descriptions on the examples. To be able to treat completeness and consistency using the same terminology, the notions of *satisfaction* and *violation* were introduced.

To be able to talk also about knowledge bases that are correct as well as complete and consistent w.r.t. all possible examples, we introduce the following definitions:

Definition 3.18 *A predicate p depends on a predicate q in a knowledge base KB iff*

1. *there is a clause c for p in KB such that q occurs in the body of c or*
2. *there is a clause c for p in KB such that there is a predicate r that occurs in the body of c and such that r depends on q.*

Definition 3.19 *A knowledge base KB is definitely correct for a set of predicates P, if all clauses in KB for predicates in P and predicates on which a predicate in P depends, are correct and KB covers all (known and unknown) positive examples of predicates in P.*

Notice that definitely correct takes into account all possible examples; this includes the examples that have not yet been received by the system. Roughly speaking, definitely correct means that the intended interpretation τ_2 and the semantics φ_2 agree.

It is necessary to define a notion of dependency as the interpretation according to φ of one predicate depends on the definition of other ones. This dependency notion can be represented graphically as in figure 3.1. There is a directed arc from one predicate to another if the latter occurs in a clause of the former. These definitions and the dependency graph are illustrated in example 3.1.

In practice, it is helpful to distinguish learned from basic predicates.

Definition 3.20 *A basic predicate is a predicate, whose definition (description) is given by the user and is assumed to be definitely correct.*

This implies that all predicates on which a basic predicate depends are also definitely correct!

Example 3.1: Illustrating correctness, completeness and consistency.

Assume the intended interpretation, τ_2, is defined as follows:

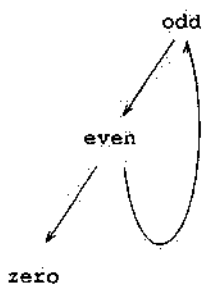
$\tau_2(e)$ = true iff
e = `zero(0)`
e = `odd(s`^{n}`(0))` where n is odd
e = `even(s`^{n}`(0))` where n is even

When $\tau_2(e) \neq$ true, it is false. Consider the following knowledge base:

```
zero(0) ←
even(X) ← zero(X)
odd(s(X)) ← even(X)
```

Given the examples τ_2(`even(0)`) = `true`, τ_2(`odd(s(0))`) = `true` and τ_2(`odd(0)`) = `false`. The knowledge base is complete and consistent (all examples are satisfied). Furthermore, all clauses in the knowledge base are correct and the concept-descriptions for `zero` and `odd` are correct. However, the concept-description for `even` is not correct as the clause `even(s(X))` ← `odd(X)` is missing. Notice that the concept-description for `odd` is correct (in the intended interpretation) although the description is not definitely correct as it does not cover τ_2(`odd(s(s(s(0))))`) = `true`, an example that is in the intended interpretation but not yet in the set of known examples. The knowledge base is only definitely correct for the predicate `zero`. However, if we also add `even(s(X))` ← `odd(X)` the knowledge base is definitely correct for all predicates. The dependency graph for this extended knowledge base is shown in figure 3.1.

Figure 3.1: A dependency graph.



Definition 3.21 *A learned predicate is a predicate, whose definition (description) has been learned by the system. Learned predicates may be incorrect.*

Interactive concept-learners generate examples and ask the user to classify them. Therefore they assume an oracle to answer membership questions [Angluin 87, Shapiro 83].

Definition 3.22 *A membership question for an example e asks the oracle for the truth-value of e in the intended interpretation. The oracle must answer $\tau(e) = t$, where t is the truthvalue. In the two-valued framework, $t = true$ or $t = false$.*

The problem of incremental and interactive concept-learning can now be formalized in logic as follows:

Problem Statement 3.1 *Interactive and incremental concept-learning.*

- **Given:**
  - a set of examples E
  - a knowledge base KB of learned and basic predicates;
  - a language L_C defining Horn-clauses
  - an example e and its truthvalue $\tau(e)$
  - an oracle willing to answer membership questions
- **Find:** a knowledge base KB', obtained by retracting and asserting clauses belonging to L_C from KB, such that KB' satisfies all examples in $E \cup \{e\}$

In example 3.2, problem statement 3.1 is illustrated. The major difference between this problem statement and the general specification of concept-learning in chapter 2 (problem 2.1) lies in the presence of the knowledge base. Here, the learning task depends very much on the available knowledge. Two concept-learning problems that are identical except for the knowledge base can be very much different. Learning `grandfather` in terms of `grandparent` and `male` is easy, whereas learning `grandfather` in terms of `male`, `female` and `parent` is harder and learning `grandfather` in terms of `female`, `male` and `married` is impossible. The presence of the knowledge base was also responsible for the more complicated definitions, given above.

Notational Remark : In this section many definitions were given in which the knowledge base was involved. In the future, for ease of reading and writing, I will often take the freedom to shorten sentences of the form x *w.r.t. the knowledge base* to x, for convenience. Nevertheless, the knowledge base is implicitly assumed to be present in these situations.

Example 3.2: Illustrating incremental concept-learning

Suppose the following knowledge base is used:

```
license(katharina, car) ←
license(yves, truck) ←
license(yves, car) ←
license(stephan, car) ←

isa(mercedes, car) ←
isa(dyane, car) ←
isa(eddy_merckx_bike, bike) ←
isa(peugeot, car) ←

owns(stephan, eddy_merckx_bike) ←
owns(katharina, mercedes) ←
owns(yves, peugeot) ←
owns(luc, dyane) ←

is_allowed_to_drive(X, Y) ← owns(X, Y)
```

Suppose that `is_allowed_to_drive` is a learned predicate, that it is known that τ_2(`is_allowed_to_drive(katharina, mercedes)`)= true and that L_C allows to express any Horn-clause. Under these circumstances, the current knowledge base is correct on all examples. However, suppose that a new example τ_2(`is_allowed_to_drive(luc, dyane)`)= false is supplied to the algorithm. Then, the knowledge base is incorrect on this example and must be modified. After asking some questions, as e.g. `is_allowed_to_drive(stephan, bike)` (true in the intended interpretation), the system may derive the following complete and consistent definition for `is_allowed_to_drive` :

```
is_allowed_to_drive(X,Y) ← license(X, car) ∧ owns(X,Y) ∧ isa(Y, car)
is_allowed_to_drive(X,Y) ← owns(X, Y) ∧ isa(Y, bike)
```

3.3 Abstract concept-learners.

In this section, two functions that model a concept-learner for Horn-clauses are defined: $\mathcal{VS}$ for learning conjunctive concepts and $\mathcal{DVS}$ for learning disjunctive ones. The abstract learners serve as a theoretical model for concept-learners and allow to present certain techniques, in particular the shift of bias in chapter 4 and the multi-valued logic in chapter 7, independently of Clint.

A language of concept-descriptions L, a knowledge base KB and a set of examples $P \cup N$ of a predicate p together determine a versionspace of complete and consistent concept-definitions (see also [Mitchell 77, Mitchell 82]). The functions $\mathcal{VS}$ and $\mathcal{DVS}$ are defined to capture these notions. The functions take four arguments: a knowledge base KB, a language of concept-descriptions L and a set of positive P and negative examples N. The result of the functions is the set of concept descriptions that are expressible in the language and that are complete and consistent w.r.t. P and N, given the knowledge base KB. $\mathcal{VS}$ yields a set of clauses whereas $\mathcal{DVS}$ yields a set of sets of clauses (see below).

The arguments of the functions $\mathcal{VS}$ and $\mathcal{DVS}$ should be interpreted as follows :

- KB is a knowledge base
- L is a language specifying syntactic restrictions on clauses for the predicate p; L can best be seen as a set of clauses for the predicate p. L does not contain clauses for other predicates. In this way the language depends on the concept to be learned and — as we will see later — on the given knowledge base.
- P and N are sets of examples of the predicate p

3.3.1 The conjunctive versionspace.

The conjunctive versionspace is defined as follows:

Definition 3.23 $\mathcal{VS}(KB, L, P, N) = \{c \in L \mid c$ *is a clause for a predicate* p *which is complete and consistent with* $P \cup N$ *and* $KB\}$

Theorem 3.1 *For VS the following properties hold:*

$$(1) \quad \mathcal{VS}(KB, L, \emptyset, \emptyset) = L$$

$$(2) \quad \mathcal{VS}(KB, L, P \cup \{p\}, N) = \mathcal{VS}(KB, L, P, N) \cap \mathcal{VS}(KB, L, \{p\}, \emptyset)$$

$$(3) \quad \mathcal{VS}(KB, L, P, N \cup \{n\}) = \mathcal{VS}(KB, L, P, N) - \mathcal{VS}(KB, L, \{n\}, \emptyset)$$

The proof of this theorem is omitted as it is trivial. Rule (1) states that any concept-description in L is complete and consistent with an empty training set. Rule (2) states that in order to update the versionspace for a new positive example, the intersection of the old versionspace with the set of clauses covering the positive example, must be computed. Rule (3) states that in order to update the versionspace for a new negative example, one must delete the clauses covering the negative example from the versionspace. Note that the rules may be applied in any order. They will always yield the same result. Theorem 3.1 specifies an (inefficient) algorithm to learn conjunctive concepts if $\mathcal{VS}(KB, L, \{e\}, \emptyset)$ is computable. $\mathcal{VS}(KB, L, \{e\}, \emptyset)$ is the set of clauses in L that cover e for the knowledge base KB. The issues, involved in computing this set of clauses, will be discussed in section 3.4 and in chapter 4. In example 3.3, the use of the function $\mathcal{VS}$ is illustrated.

Example 3.3: The function VS and its use.

Suppose the knowledge base KB contains the following definitions:

```
female(alice) ←                    male(luke) ←
female(kate) ←                     male(morris) ←
female(rose) ←                     male(jeffrey) ←
female(mary) ←                     male(steve) ←

parent(jeffrey, kate) ←            parent(alice, kate) ←
parent(alice, luke) ←              parent(mary, luke) ←
parent(rose, mary) ←               parent(morris, jeffrey) ←
parent(alice, steve) ←
```

Suppose the concept-description language L is the language of *completely bound* Horn-clauses. This language contains only Horn-clauses without functors nor constants. Furthermore, all variables occurring in the body of a clause also occur in the head of the clause and vice versa. This implies that there are no local variables.
Suppose that the aim is to learn the father concept from the positive examples `father(jeffrey, kate)` and `father(morris, jeffrey)` and the negative example `father(rose, mary)`.
Then, because of theorem 3.1 :

(1) $VS(KB, L, \emptyset, \emptyset) = \{c \mid head(c) =$ `father(X,Y)` and $body(c) \subseteq B$ such that all variables of $head(c)$ occur in $body(c)\}$ where
B = {`male(X), male(Y), female(X), female(Y), parent(X,Y), parent(Y,X), parent(X,X), parent(Y,Y)`}

(2) $VS(KB, L, \{$`father(jeffrey, kate)`$\}, \emptyset) =$
$\{c \mid head(c) =$ `father(X,Y)` and $body(c) \subseteq C$ such that all variables of $head(c)$ occur in $body(c)\}$ where $C =$ {`male(X), female(Y), parent(X,Y)`}

Written explicitly $VS(KB, L, \{$`father(jeffrey, kate)`$\}, \emptyset) =$

```
{father(X,Y)  ← male(X) ∧ female(Y) ∧ parent(X,Y) ;
father(X,Y)  ← male(X) ∧ parent(X,Y) ;
father(X,Y)  ← female(Y) ∧ parent(X,Y);
father(X,Y)  ← male(X) ∧ female(Y);
father(X,Y)  ← parent(X,Y)}
```

(3) $VS(KB, L, \{$`father(jeffrey, kate)`$\}, \{$`father(rose, mary)`$\}) =$

```
{father(X,Y)  ← male(X) ∧ female(Y) ∧ parent(X,Y);
father(X,Y)  ← male(X) ∧ parent(X,Y);
father(X,Y)  ← male(X) ∧ female(Y)}
```

(4) $VS(KB, L, \{$`father(jeffrey,kate), father(morris,jeffrey)`$\}, \{$`father(rose,mary)`$\}) =$

```
{father(X,Y)  ← male(X) ∧ parent(X,Y)}
```

3.3.2 The disjunctive versionspace.

For disjunctive concepts, the situation is more complex. The function $\mathcal{DVS}$ defines the space of complete and consistent disjunctive concept-descriptions as follows:

Definition 3.24 $\mathcal{DVS}(KB, L, P, N) = \{S \mid S$ *is a minimal set of clauses for a predicate* p *in* L*, such that* S *is complete and consistent with* KB *and* $P \cup N\}$

The term minimal *means that there is no proper subset of* S *which is complete and consistent. If a clause is deleted from a minimal concept-description then there will be positive examples that are no longer covered.*

Theorem 3.2 *For DVS the following relations hold:*

$$(1)\ \mathcal{DVS}(KB, L, \emptyset, \emptyset) = \{\emptyset\}$$

$$(2)\ \mathcal{DVS}(KB, L, P \cup \{p\}, N) =$$
$$\{D \cup S \mid D \in \mathcal{DVS}(KB, L, P, N) \text{ and } S \in \mathcal{DVS}(KB \cup D, L, \{p\}, N)$$
$$\text{such that } S \cup D \text{ is minimal for } P \cup \{p\}, N \text{ and } KB\}$$

$$(3)\ \mathcal{DVS}(KB, L, P, N \cup \{n\}) =$$
$$\{S \mid S \in \mathcal{DVS}(KB, L, P, N) \text{ and } \varphi_2(KB \cup S, n) = \text{ false }\}$$

The proof of theorem 3.2 is omitted as it is straightforward. Rule (1) states that the only complete and consistent concept description w.r.t. the empty training set is the empty concept description. In order to update a disjunctive versionspace for a positive example, one must compute the descriptions that are the union of a description that belongs to the old versionspace and a description that covers the new example w.r.t. the updated knowledge base. Only the minimal elements of these newly computed descriptions should be retained. Observe that S can be empty (when p is covered by D), can have one element (usually) or more than one element (in the recursive case). Rule (3) states that to update the versionspace for a new negative example, all descriptions that cover the negative example in the old versionspace must be deleted.

Some concept-learning systems, such as Clint, replace rule (2) by the following one :

$$(4)\ \mathcal{DVS}(KB, L, P \cup \{p\}, N) =$$
$$\{D \cup S \mid D \in \mathcal{DVS}(KB, L, P, N) \text{ and}$$
$$S \in (\{\{c\} \mid c \in \mathcal{VS}(KB \cup D, L, \{p\}, N)\} \cup \{\emptyset\}) \text{ and } S \cup D \text{ is minimal }\}$$

Rule (4) has a lower computational complexity than rule (2) as only singleton sets are considered for addition to the description. Using rule (4) instead of (2) is correct for non-recursive concepts. The reason why it is correct for non-recursive predicate, is that for each fact of a non-recursive predicate there is only one clause of the predicate that is used in a proof of the fact. Rule (4) can also be extended with mechanisms such as postponing (see section 3.5) to be correct in the case of recursion. We will come back to these issues in the context of the system Clint (see sections 3.5 and 3.6).

The use of $\mathcal{DVS}$ is not illustrated as it is a simple extension of $\mathcal{VS}$. Under the same conditions as theorem 3.1, theorem 3.2 specifies an (inefficient) algorithm for computing $\mathcal{DVS}$.

3.3.3 The meaning of $\mathcal{VS}$ and $\mathcal{DVS}$.

Notice that the abstract concept-learners of theorem 3.1 and 3.2 are not interactive as they do not rely on an oracle. However, they are incremental since they specify how to update the versionspace to accommodate a new example. Notice also, that the resulting algorithms can be realized as an instantiation of Gencol. For instance, the function $\mathcal{VS}$ can easily be obtained by instantiating Gencol as follows:

- the list of partial solutions contains the complete and consistent descriptions
- for a new example, all descriptions on the list of partial solutions are selected once
- if the description satisfies the new example, it is retained (the result of the transformer is the singleton containing the description itself); otherwise, the element is discarded from the list (the result of the transformer is the empty set)
- the other generic parameters are straightforward.

A related instantiation can be used to obtain $\mathcal{DVS}$.

The relevance of $\mathcal{VS}$ and $\mathcal{DVS}$ is that they make abstraction of how specific concept-learners learn concepts. The rules given in theorems 3.1 and 3.2, are valid for all sound concept-learners that learn minimally complete and consistent concept-descriptions. However, the algorithms implied by the theorems should not be used in practice because they are inefficient. In practical algorithms, as the versionspace method of [Mitchell 77, Mitchell 82], the space of complete and consistent concept-descriptions is not stored explicitly. Instead, only boundary sets determining this space are computed. By ignoring these practical considerations in the definitions of $\mathcal{VS}$ and $\mathcal{DVS}$, it will be possible to present certain learning techniques of Clint independently of Clint. As a consequence, it will be easy to integrate these techniques in other sound concept-learners that compute elements of $\mathcal{VS}$ and $\mathcal{DVS}$. This includes systems such as the versionspace method [Mitchell 77, Mitchell 82], The Model Inference System [Shapiro 83], incremental versionspace merging [Hirsh 90] and learning by enumeration [Gold 67, Angluin and Smith 83, Biermann 86].

3.4 Bias in concept-learning.

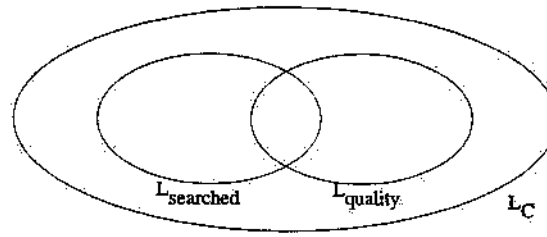
This section contains a discussion of the very important notion of bias in concept-learning.

3.4.1 Language bias and search bias.

[Utgoff and Mitchell 82] define bias as anything which influences how the concept-learner draws inductive inferences based on the training-examples.

There are basically two different classes of bias: on the one hand there is the bias of the description language of concepts, which determines the space of possible concept-descriptions; and on the other hand there is the bias which determines how the space of descriptions is searched. In terms of the abstract concept-learners $\mathcal{VS}$ and $\mathcal{DVS}$, the first bias is concerned with the form of descriptions in the language L. Choosing to use $\mathcal{VS}$ or $\mathcal{DVS}$ is also a form of language bias because it determines whether conjunctive or disjunctive concepts are considered. The second bias determines which part of the

Figure 3.2: Illustrating language bias



versionspace is searched. Some systems (see chapter 2) that use heuristics consider only part of the search-space. In a sense, the language bias determines *what* to search (L_C) and the search bias *how* to search ($L_{searched}$). These notions are illustrated in figure 3.2.

[Mitchell 80] demonstrates that bias is necessary and inevitable. First, without bias there is no proper criterion to decide which concept-descriptions are interesting. In principle, all concept-descriptions satisfying the quality criterion are acceptable ($L_{quality}$). Consider the function $\mathcal{DVS}$ and imagine that $L_C = 2^{L_e}$. In this language, all concepts can be expressed. (One can argue that this language does not bias the learning task, as it does not favor any concept-description.) If the quality criterion is completeness and consistency, the concept-description consisting of the disjunction of the positive examples is a solution to the concept-learning task. Although this description is – under normal circumstances – not interesting (it does not allow to draw conclusions beyond the training-set), there is – without bias – no means to prefer other concept-descriptions. A second reason why bias is necessary, is that the concept-description language may be very large (it frequently occurs that it contains an infinite number of concept-descriptions). Without bias or preferences the complete space of descriptions has to be searched; this may be impossible or require too much computation. Bias can then be used to prune the search.

Although bias is necessary to make the search manageable, it also has the disadvantage that it may bias the search in such a manner that no solutions are found. In figure 3.2, this situation would arise when $L_{searched} \cap L_{quality} = \emptyset$. There may be two different causes for this problem: the language bias may not allow to express the correct concept-description ($L_C \cap L_{quality} = \emptyset$) or the search process may not consider the right descriptions ($L_{searched} \cap L_{quality} = \emptyset$ but $L_C \cap L_{quality} \neq \emptyset$). The first cause is illustrated in example 3.4. This example motivates the following definition:

Definition 3.25 *A language bias L is correct w.r.t. a knowledge base KB for defining a predicate p if there is a correct concept-description for p in L that defines p using the predicates in KB.*

Notice that the correctness of a language bias also depends on the predicates in the knowledge base! However, it does not depend on whether the predicates in the knowledge base are definitely correct or not. This will allow to distinguish two different reasons why a concept has no complete and consistent description at a certain moment in time: either the knowledge base is not definitely correct or the language is not expressive enough. This reveals that the knowledge base can (and should sometimes) be seen as a third form of bias.

Example 3.4: Illustrating language bias.

Given the knowledge base of example 3.3, the language of completely bound Horn-clauses and the facts:

τ_2(`grandparent(rose, luke)`) = true, τ_2(`grandparent(morris, kate)`) = true and τ_2(`grandparent(alice,luke)`) = false, τ_2(`grandparent(steve, luke)`) = false

there is no complete and consistent concept-description for `grandparent`. The reason is that the most specific clause covering the example `grandparent(rose, luke)` is `grandparent(X,Y)` ← `female(X)` ∧ `male(Y)`. This clause also covers the negative example `grandparent(alice, luke)`.
In the description language $L_{e(1)}$ `grandfather` can be described as: `grandparent(X,Y)` ← `parent(X,Z)` ∧ `parent(Z,Y)`. The language $L_{e(n)}$ contains all Horn-clauses that are linked and range-restricted (see definitions 3.26 and 3.27 below), contain neither constant nor function symbols and have at most n local variables. The correct clause for `grandparent` can be computed using the algorithm $\mathcal{VS}$.

In the remainder of this thesis, we will concentrate on language bias, rather than search bias. The reason for this is that language bias can be declaratively represented [Russell and Grosof 90, Grosof and Russell 90]. Declarative representations of bias allow to reason about bias at the meta-level, e.g. to realize a shift of bias. Reasoning about language bias will be achieved in chapter 4. Reasoning and manipulating search bias is more difficult as search bias is a kind of procedural knowledge.

3.4.2 Language bias: some assumptions.

Although it is not necessary to choose a particular language bias now, we will impose certain restrictions on the languages that will be considered in the remainder of the thesis. These restrictions make it possible to prove certain properties about the algorithms defined in the next sections and chapters. Variants of these restrictions have been made by other researchers.

Definition 3.26 *A clause is range restricted if all variables occurring in head(c) also occur in body(c).*

Definition 3.27 *A clause c is linked if all of its variables are linked. A variable v is linked in a clause c iff v occurs in head(c) or there is a literal l in c, that contains the variables v and w ($v \neq w$) and w is linked in c.*

The range restriction and the linkage requirement exclude usually useless clauses as for instance p(X) ← r(Z) and p(X) ← r(X,Y) ∧ q(Z). The first clause is not range restricted whereas the second clause contains the unlinked variable Z.

All languages used in this thesis satisfy the following assumptions:

Assumption 3.1 *Language Assumptions.*

A language L satisfies the language assumptions if:

1. *there is only a finite number of clauses in L*
2. *all clauses in L are range-restricted*
3. *all clauses in L are linked*

Furthermore, as in chapter 2, we will augment each language with a top element $\top$ and a bottom element $\bot$ for convenience.

The first assumption allows to detect when there is no complete and consistent concept-description w.r.t. the knowledge base, the language and the examples (see also chapter 4). Furthermore, it allows to prove the existence of a generalization operator as in lemma 3.2 (see below).

In order to make the language assumptions less abstract, the reader may wish to think of a real language definition when certain techniques are explained. Examples of such languages are the language of *completely bound* Horn-clauses (see example 3.3) and the languages $L_{\epsilon(n)}$ (see example 3.4). Given a certain knowledge base, these languages are finite, as it does not make sense to allow duplicate literals in the body of a clause.

3.4.3 Generalization.

A definition of generalization w.r.t. clauses is given and issues concerning generalization operators on clauses are discussed.

Let us first define, what is meant by generalization of clauses:

Definition 3.28 *A clause c' is a generalization of a clause c if* $\exists\theta$: $head(c) = head(c')\theta$ *and* $body(c')\theta \subseteq body(c)$. *We will sometimes say that c is a specialization of c'.*

Definition 3.29 *A clause c' is a proper generalization of a clause c iff c' is more general than c but c is not more general than c'.*

These definitions are very important because generalization and specialization are the key operations in concept-learning (see also chapter 2). The definition of generalization corresponds to the notion of θ-subsumption introduced by [Plotkin 70] and often used in theorem proving [Loveland 78]. The definition indicates for instance that father(X,Y) ← male(X) ∧ parent(X,Y) is a proper generalization of father(X,Y) ← male(X) ∧ male(Y) ∧ parent(X,Y) and also that ancestor(X,Y) ← person(X) is a proper generalization of ancestor(X,X) ← person(X). Some properties of the introduced notion are listed in lemma 3.1.

Lemma 3.1 *(see [Plotkin 70, Loveland 78])*

1. *The relation* is more general than *is a pre-order, i.e. it is reflexive and transitive.*
2. *The relation* is a proper generalization of *is anti-reflexive, transitive and anti-symmetric.*
3. *The relations* is a proper generalizaton of *and* is a generalizaton of *are decidable (for an algorithm see e.g. [Loveland 78, Plotkin 70]).*
4. *If c is a generalization of c' then* $\models (c \rightarrow c')$.

Two clauses can be more general than each other as for instance c = `p(X,Y) ← q(X,Z) ∧ q(W,V)` and d = `p(X,Y) ← q(X,Z)`. Indeed, $d \subseteq c$ and $c\{$ `W = X, V = Z` $\} \subseteq d$. Therefore it is useful to introduce a notion of equivalence.

Definition 3.30 *Two clauses c and c' are equivalent iff c is more general than c' and c' is more general than c.*

Notice that if two clauses c and c' are equivalent, we also have that c and c' are syntactic variants (see chapter 2), i.e. they always cover the same set of examples. This follows from property (4) in lemma 3.1 and implies that we need not be concerned with generalizations as such but rather with proper generalizations.

Definition 3.31 *A minimal proper generalization g of a clause c is a proper generalization of c such that there is no other proper generalization g' of c that is a proper specialization of g.*

Clint uses a generalization operator to generalize clauses.

Definition 3.32 *A generalization operator* λ_g *for a language L is a function from L to* 2^L *such that* $\forall c \in L : \lambda_g(c) = \{c' \in L \mid c' \text{ is a minimal proper generalization of } c\}$.

Because of the assumption that each language is finite, we can prove the existence of a generalization operator for each language:

Lemma 3.2 *If L is finite there is a generalization operator for L.*

Proof : There is a (very naive) generalization operator for L that works as outlined in algorithm 3.1. The algorithm terminates as L is finite and testing for generalization (and proper generalization) is decidable. □

Implementation Note.

Observe that algorithm 3.1 is very inefficient. In practice it is often possible to optimize the algorithm. However, these optimizations depend on the particular language being considered. For instance, for the language of completely bound Horn-clauses and for the languages defined in chapter 4, a simple generalization operator such as $\mathcal{L}_g$ is sufficient:

Definition 3.33 $\mathcal{L}_g(p \leftarrow q_1 \wedge \ldots \wedge q_k) = \{c = (p \leftarrow q_1 \wedge \ldots \wedge q_{i-1} \wedge q_{i+1} \wedge \ldots \wedge q_k) \mid c \in L\}$

Algorithm 3.1: A naive generalization operator for a finite language.

```
procedure λ_g(c : clause ∈ L) returns λ_g(c)
    Q := {c' ∈ L | c' is a proper generalization of c }
    while there is a proper generalization g ∈ Q of g' ∈ Q do
        delete g from Q
    endwhile
    return Q
endproc
```

The operator $\mathcal{L}_g$ works by deleting conditions from the body of the original clause. The operator $\mathcal{L}_g$ is also the one that is applied in the actual implementation of Clint and that will be used in the examples. $\mathcal{L}_g$ is sufficient to correctly handle all languages defined in chapter 4. Nevertheless, for the sake of generality, we will describe Clint's algorithms in general and show (see section 3.6) that they work for any language (satisfying the assumptions) and generalization operator on that language.

When implementing a generalization operator, it is often also convenient to write unification explicitly. Consider for example the clause `grandparent(X,Y)` ← `parent(X,W)` ∧ `parent(W,Y)`. With explicit unification it becomes `grandparent(X,Y)` ← `parent(X,W1)` ∧ `parent(W2,Y)` ∧ `eq(W1, W2)`, where `eq` is a predicate implementing unification. The advantage of writing unification explicitly is that it makes the generalization process easier. For instance, for the clause with explicit unification we can compute all generalizations that are linked and range-restricted using $\mathcal{L}_g$. This is not the case when unification is implicit.

Mathematical Remark.

Definition 3.30 can be used to define equivalence classes of clauses:

Definition 3.34 $[c] = \{c' \in L \mid c' \textit{ is equivalent to } c\}$

Plotkin [Plotkin 70] claims that the algebraic structure w.r.t. these equivalence classes and the relation *is more general than* is a lattice. In practical applications it may be much more efficient to work with the language L *modulo* [] and to use representatives of each equivalence class in the generalization operator. This is more efficient as the system will never consider equivalent descriptions, i.e. descriptions that are syntactically different but equivalent according to the generalization structure. However, again the feasibility of this approach depends on the particular language.

3.4.4 Justifications.

The language assumptions made above allow to simplify the computation of $\mathcal{VS}(KB, L, \{e\}, \emptyset)$ in section 3.3. Indeed, $\mathcal{VS}(KB, L, \{e\}, \emptyset)$ is completely determined by $\mathcal{JUS}(KB, L, e)$ as stated in lemma 3.3.

Definition 3.35 *The set of justifications of e w.r.t. L and a knowledge base KB, $\mathcal{JUS}(KB, L, e)$, is the set of minimal elements under the* is more general than *relation within the set of all clauses of L covering e, i.e. the set of most specific clauses in L that cover e.*

Lemma 3.3 $\mathcal{VS}(KB, L, \{e\}, \emptyset) = \{c \in L \mid c \text{ is a generalization of a } c' \in \mathcal{JUS}(KB, L, e)\}$

Proof : The lemma follows from the transitivity of *is a generalization of.* □

The clauses in $\mathcal{JUS}(KB, L, e)$ are called *justifications* because they express the relations that are valid for the example given the knowledge base. If the concept language bias is correct and the given knowledge base is definitely correct, some of the justifications are correct (see lemma 3.4). A correct justification specifies why the example belongs to the concept. The use of $\mathcal{JUS}(KB, L, e)$ is illustrated in example 3.5.

Example 3.5: Illustrating $\mathcal{JUS}(L, e)$.

Reconsider example 3.3. $\mathcal{VS}(KB, L, \{\texttt{father(jeffrey, kate)}\}, \emptyset)$ is completely determined by $\mathcal{JUS}(KB, L, \texttt{father(jeffrey, kate)}) = \{\texttt{father(X,Y)} \leftarrow \texttt{male(X)} \wedge \texttt{female(Y)} \wedge \texttt{parent(X,Y)}\}$.

Lemma 3.4 *If the language bias L is correct for learning a predicate p from KB and the knowledge base KB is definitely correct for the predicates used in a correct definition of p in L then*

1. *there is a positive example e for p for which at least one of the justifications in $\mathcal{JUS}(KB, L, e)$ is correct*
2. *when p is a non-recursive predicate, (1) holds for all positive examples of p*

Proof : If the language bias and the knowledge base are correct for the predicates used in a correct definition for p and e is an example covered by a non-recursive clause in this definition of p, at least one of the clauses in $\mathcal{VS}(KB, L, \{e\}, \emptyset)$ belongs to a correct concept-description. Therefore it must be true in the intended interpretation τ_2. Because this clause is the generalization of a clause in $\mathcal{JUS}(KB, L, e)$ and all clauses that are more specific than a correct clause are also correct, the result follows. □

A possible strategy to compute the clauses in $\mathcal{JUS}(KB, L, p)$ is given in algorithm 3.2. It starts from the most specific clause in L, $\bot$, and repeatedly generalizes clauses in all possible directions by applying a generalization operator λ_g. If a clause c covers p and no proper specialization of c covers p, c must belong to $\mathcal{JUS}(KB, L, p)$.

The efficiency of the algorithm can be improved if the language characteristics of L are taken into account and care is taken that no clause is generated more than once. Notice also that it is easy to derive the dual algorithm of algorithm 3.2. Then one uses a specialization operator and starts from $\top$. In certain situations this may be more efficient.

Because of the properties of *is more general than* we have:

Lemma 3.5 *Repeated application of the operator λ_g on the clause $\perp$ in L allows to compute all clauses in L.*

Proof : The result follows from the transitivity of the relation *is more general than* and the finiteness of L. □

Algorithm 3.2: Computing $\mathcal{JUS}(KB, L, p)$ specific to general.

```
procedure compute_jus(L: language , p: example) returns : JUS(KB, L, p)
    Q := {⊥ ∈ L}
    jus := ∅
    repeat
        delete a clause c from Q
        for all clauses c' ∈ λ_g(c) do
            if c' covers p
            then add c' to jus
            else add c' to Q
            endif
        endfor
    until Q = ∅
    return jus
endproc
```

3.5 Clint's basic algorithms.

In this section, the basic algorithm of Clint is presented. It learns concepts within a given language, is incremental and generates examples. The presented algorithm is more general than the ones previously published in [De Raedt and Bruynooghe 88, De Raedt and Bruynooghe 89d, De Raedt and Bruynooghe 89e]. Instead of merely dropping conditions as before, it generalizes using a generalization operator. Also, the use of postponing (cf. infra) is new. Clint's properties will be studied in sections 3.6, 3.7 and 3.8.

3.5.1 Approaching the problem.

Analyzing problem statement 3.1, chapter 2 and the abstract concept-learners $\mathcal{VS}$ and $\mathcal{DVS}$ reveals that there are two possible approaches to the problem: a global approach, taking into account all elements in $\mathcal{VS}(KB, L, P, N)$ or $\mathcal{DVS}(KB, L, P, N)$ and a local approach, taking into account only a single (current) concept-description. Existing methods along the first line include the versionspace algorithm [Mitchell 77, Mitchell 82] and its variants [Murray 87, Subramanian and Feigenbaum 86, Hirsh 90]. Well-known methods that follow the second direction are the Model Inference System [Shapiro 83], Marvin [Sammut and Banerji 86] and Cigol [Muggleton and Buntine 88]. An argument in favor

of the global approach is that decisions can be made in an optimal way. It is for instance possible to select the most informative examples [Subramanian and Feigenbaum 86] (the ones that split the versionspace in two equal parts or best approximate this). In contrast, the local approach does not allow to take optimal decisions. Usually however, good decisions are possible. For the generation of examples, it has been empirically shown [Ruff and Dietterich 89] that the local strategy of selecting relevant examples performs almost as well as the global strategy of selecting optimal examples. The disadvantage of the global methods is that they are computationally very expensive. As a consequence they are often not feasible for all but the simplest description languages. It is well-known that there are problems with the global approaches for learning disjunctive concepts [Murray 87, Kodratoff 88]. Another example of the cost of the global approach is the computation of the most informative examples, which is exponential (see [Subramanian and Feigenbaum 86]) Also, in many cases the decisions taken by the global approach are less understandable for a user than the ones of the local approach. Because of these arguments, I choose to follow the local direction. An additional motivation for this choice was that for systems such as Marvin and Cigol it is not clear which class of concepts can be learned and what these systems need to learn a certain concept. While developing Clint, I have always tried to also address these theoretical concerns and to show that the local direction can also be theoretically appealing.

In the next sections we will lay the foundations for the algorithms of Clint. There are basically two procedures in Clint : a generalization procedure that starts from an uncovered positive example and derives a clause that will cover the example and a specialization procedure that starts from a covered negative example and removes an incorrect clause from the knowledge base. The first procedure computes the clause in two steps : in a first step, a consistent starting clause is derived from the positive example; this starting clause is then generalized in the second step of the generalization procedure.

3.5.2 Generalizing examples.

In this section, the generalization algorithm of Clint is developed. This algorithm starts from a positive example and generates a clause covering the example.

What knowledge can be derived from positive examples ? Let us for the moment assume that the language bias L and the knowledge base are correct for the concept. Lemma 3.4 then allows to conclude that for some positive examples p, at least one clause $c \in \mathcal{JUS}(KB, L, p)$ will be correct. One possible approach is to add one of the clauses in $\mathcal{JUS}(KB, L, p)$ to the knowledge base when a new positive example p is acquired. However, clauses in $\mathcal{JUS}(KB, L, p)$ are most specific in L while covering the example p. Therefore it is not a good idea to add these clauses directly to the knowledge base. The reason is that in this way many clauses may have to be added; in some cases even one clause for each example. Also, adding clauses in $\mathcal{JUS}(KB, L, p)$ is neither opportunistic, as it only infers the minimal conclusions, nor interactive, as no examples are generated. It is much more interesting to attempt to generalize the clause as much as possible in a cautious way. Generalizing as much as possible means computing maximally general clauses.

Definition 3.36 *A clause c is maximally general w.r.t. a set of examples E and a knowledge base KB, if the clause is consistent with E and there is no generalization g of c such that g covers more examples than c and g is consistent with E.*

Cautious generalization requires evidence for each generalization step. There is evidence for generalizing a clause c to a clause c' if there is a positive example p that is covered by c' but not by c. Of course, the generalized clause c' should never cover negative examples.

Another important aspect of generalization is the size of the individual generalization steps. Systems such as Marvin and Cigol sometimes take non-minimal steps. Informally, they compute the generalization of a clause $p \leftarrow q_1 \wedge ... \wedge q_m \wedge s_1 \wedge ... \wedge s_n$ taking into account another clause $r \leftarrow q_1 \wedge ... \wedge q_m$. Then they compute the generalized clause $p \leftarrow r \wedge s_1 \wedge ... \wedge s_n$[2]. In such an approach to generalization, the resulting generalizations depend very much on the clauses that are in the knowledge base. More importantly it is not clear whether all generalizations of a clause in the language can be obtained by applying such a generalization operator. As this property is important for convergence and learnability results, I decided to consider a different generalization transformer. The clause transformer used for Clint is a strict generalization operator such as the λ_g introduced above.

Our notion of generalization defined in definition 3.28 is purely syntactic. Therefore it may occur that for a certain knowledge base KB, a clause c and a proper generalization c' of c there cannot exist an example e such that $covers_{KB}(c', e)$ and not $covers_{KB}(c, e)$. If this is the case, then c and c' are syntactic variants for the knowledge base KB (see chapter 2). The reason is that our notion of generalization does not take into account the given knowledge base. Therefore it may occur that although one clause is a syntactic generalization of another clause, the first clause does not cover more examples than the second one. Consider e.g. `grandfather(X,Y)` ← `father(X,Z)` ∧ `parent(Z,Y)` ∧ `parent(X,Z)` and `grandfather(X,Y)` ← `father(X,Z)` ∧ `parent(Z,Y)`.

Based on these observations, algorithm 3.3 was designed. It generalizes a clause $c \in \mathcal{JUS}(KB, L, p)$ to obtain a clause that is maximally general and does not cover negative examples. Before presenting the algorithm in more detail, let us make a very important convention:

> *All algorithms of Clint store all the acquired examples and use them for testing completeness and consistency of concept-descriptions.*

This convention is useful to simplify the description of the algorithms. Indeed, we will assume that all examples and their intended interpretation are remembered automatically, without having to write statements to realize this.

Algorithm 3.3 starts from the given clause c and applies the operator λ_g on c. If this results in a clause c' that does not cover negative examples and covers an example e that was not covered by the given clause c, the example e is presented to the oracle. If the example e is positive, the algorithm generalizes c by replacing it by c' and the process is repeated for the generalized c. If the example is negative, the system discards the generalization and backtracks. Because there is the possibility of syntactic variants, it may be necessary to apply the operator λ_g recursively on an element of $\lambda_g(c)$ before a clause c' is found that covers more examples than c.

At point $\triangleright 2 \triangleleft$ the following invariant holds:

[2]Note that according to the definition of generalization in 3.28, this clause would not be a generalization. However in other more semantical theories of generalization such as the one of [Buntine 88], this clause is a generalization.

1. *c' is a minimal proper generalization of c such that* (2),(3) *and* (4) *hold*
2. *e is covered by c'*
3. *e is not covered by c*
4. *c' does not cover negative examples*

Property (1) implies that all proper generalizations c'' of c that are a proper specialization of c' are syntactic variants of c for the given knowledge base. This also explains why these clauses are not used to generate relevant examples. This invariant makes the assumptions underlying generalization in the algorithm explicit.

When posing questions to the oracle, we will always ask for the truthvalue according to τ, which can be τ_2 or τ_M in practice. The algorithms — however — always base their actions on the two-valued intended interpretation τ_2. The reason why we may ask for τ, which can be τ_M, is that knowledge about τ_M will be translated into knowledge about a derived τ_2. In this way τ_M acts as a kind of meta-level for the object-level τ_2. These issues will be further explained in chapter 7.

Algorithm 3.3: Generalization in Clint.

```
procedure generalize_clause( c : consistent clause ): returns maximally general clause
    Q = λ_g(c)
    while Q ≠ ∅ do
        delete c' from Q  ▷ 1 ◁
        if c' does not cover negative examples
        then if c' covers an example e that is not covered by c
             then ask the oracle for τ(e)  ▷ 2 ◁
                  if τ_2(e) = true
                  then return generalize_clause(c')
                  endif
             else Q := Q ∪ λ_g(c')
             endif
        endif
    endwhile
    return c
endproc
```

Algorithm 3.3 will be illustrated in section 3.5.5. The algorithm does not specify which clauses to delete from Q at point ▷ 1 ◁ nor how the example is computed at point ▷ 2 ◁. Computing the example is quite easy: rename the variables in c and c' such that no variable of c occurs in c' and vice versa, then answer the following query $\leftarrow$ *body*(c') $\wedge \neg$(*head*(c)=*head*(c')$\wedge$ *body*(c)). The = in the query denotes unification; the $\neg$ denotes negation as failure. If there are no solutions to the query, there is no example that is covered by c' and not by c. In case there is a solution to the query in the form of a ground substitution θ, *head*(c')θ is an example fulfilling the necessary conditions.

The query can be executed by theorem provers or deduction mechanisms such as Prolog [Lloyd 87]. In general, however, there may be problems with answering such queries. These problems will be studied in section 3.7. Until then, we will assume a perfect query answering mechanism.

Definition 3.37 *A perfect query answering procedure generates for each knowledge base KB and for each query of the form* $\leftarrow p_1 \wedge ... \wedge p_k \wedge \neg(q_1 \wedge ... \wedge q_m)$, *where* $1 \leq k$, *one of the following answers after a finite computation:*

$$\begin{array}{rl} \textit{a ground substitution } \theta & \textit{if } KB \models (p_1 \wedge ... \wedge p_k \wedge \neg(q_1 \wedge ... \wedge q_m))\theta \\ \textit{otherwise} & \textit{false} \end{array}$$

Notice that a perfect query answering mechanism is necessary to guarantee that the example generating process terminates!

Let us now briefly address the second issue, i.e. which elements to delete from the queue Q at $\rhd 1 \lhd$. In the implementations of Clint, we have discovered that a breadth-first search is most useful. However, in some situations the search can be improved by using knowledge and heuristics. In [De Raedt and Bruynooghe 88], we have presented several strategies to make the search more efficient in the case of simple languages and generalization operators such as the language of completely bound Horn-clauses (see example 3.3).

An improvement of the search that is often applicable is the following one:

> If $c' \in \lambda_g(c)$ is obtained by deleting a literal $p(t_{i_1}, ..., t_{i_k})$ from c and there is a substitution θ such that $p(t_{i_1}, ..., t_{i_k}) = p(x_{i_1}, ..., x_{i_k})\theta$ and $KB \models (q(x_1, ..., x_n) \rightarrow p(x_{i_1}, ..., x_{i_k}))$ and $q(x_1, ..., x_n)\theta\rho \in c$ then delete also $q(x_1, ..., x_n)\theta\rho$ from c'. (All the x's are terms such that $vars(\{x_{i_1}, ..., x_{i_k}\}) \subseteq vars(\{x_1, ..., x_n\})$). In this case, $q(x_1, ..., x_n)$ is a semantic generalization of $p(x_{i_1}, ..., x_{i_k})$. If $q(x_1, ..., x_n)\theta\rho$ would not be removed from c' then c' would be a syntactic variant of c w.r.t. KB. So, the optimization can be used without loss of generality. In practice, it is only applied when the $\{x_{i_1}, ..., x_{i_k}\}$ are different variables.

Lemma 3.6 *Algorithm 3.3 returns a maximal generalization of the starting clause c w.r.t. the given knowledge base and set of examples if c is consistent.*

Proof : The result follows from the facts that (1) the result of algorithm 3.3 does not cover negative examples and (2) the returned clause cannot be generalized to a clause that is not a syntactic variant of the returned clause without covering negative examples and (3) the algorithm terminates.

(1) follows from the consistency of the starting clause (assumed), the safe generalization strategy (the algorithm does not generalize to inconsistent clauses) and the fact that the generated examples are by definition not covered by the returned clause c.

(2) follows from the observation that all relevant generalizations of the returned clause c are tested for consistency and that c is generalized whenever one of its generalizations is consistent. If a minimal generalization is a syntactic variant of the clause c, the minimal generalizations of the syntactic variant will be considered. If a minimal generalization covers a negative example, it is not necessary to consider its generalizations as they will also be inconsistent. So, this minimal generalization may be discarded from Q. If a

generalization on Q is found to be consistent, the system generalizes. This case analysis reveals that any potentially consistent generalization of the returned clause c is considered by the algorithm and will be found on Q. Because the termination condition for the repeat loop is $Q = \emptyset$, it must be the case that no generalization of c that covers more examples than c is consistent (this holds for the deepest call of generalize_clause). This means that the returned clause c is maximally general.

(3) the algorithm terminates because there is only a finite number of generalizations of c in L and a perfect query answering procedure is used to construct examples. □

Observe that algorithm 3.3 assumes that the starting clause c is consistent. If c covers a negative example, no generalization of c can be consistent. This property makes it possible to prune some of the clauses in $\mathcal{JUS}(KB, L, p)$ from further consideration. It also motivates algorithm 3.4, which computes a maximally general clause covering a positive example p and asserts it in the knowledge base. This algorithm is used by Clint to recover from an uncovered positive example.

The system must take care when it learns disjunctive concepts. While learning one clause, it may gather negative examples that are covered by another clause of the same concept. This explains the for loop just after asserting the clause. Algorithm 3.6 may be called there to recover from a covered negative example.

A second point to note is that in contrast to our theoretical model of disjunctive learners ($\mathcal{DVS}$), this algorithm does not learn minimal concept-descriptions. In order to learn minimal concept-descriptions it would be necessary to check whether other clauses of the same concept become redundant when asserting c'. This is not incorporated in the algorithm because such tests are computationally expensive and would also complicate the theoretical treatment of the algorithm in section 3.6.

Another difference with respect to our theoretical model $\mathcal{DVS}$ is that $\mathcal{DVS}$ may add a set of clauses to the knowledge base when a new positive example is processed. In the algorithm of Clint this is not realized because of complexity reasons. Clint uses the simplified model of $\mathcal{DVS}$, i.e. it uses rule (4) of theorem 3.2 instead of rule (2) (see section 3.3). The algorithm of Clint does not derive sets of clauses for one positive example but it has the ability to detect anomalies. This ability will turn out to be nearly as powerful.

Definition 3.38 *An anomaly occurs when $\mathcal{VS}(KB, L, \{p\}, N) = \emptyset$ for a language L, a positive example p, the set of negative examples N and the knowledge base KB.*

When an anomaly arises, it means that there is no complete and consistent concept-description given the language, the examples and the knowledge base. Anomalies are illustrated in example 3.6. There may basically be two reasons for this: either the language is too restricted or there is not enough knowledge in the knowledge base. The first cause will be handled in chapter 4; the second one will be dealt by algorithm 3.7. The issues concerning anomalies will be extensively discussed in section 3.6.

3.5.3 Specialization.

In the previous section it was shown how to update the knowledge base for an uncovered positive example. Here, a method to recover from a covered negative example is presented. The presented method is based on the work of Shapiro [Shapiro 83], which often serves as the basis for treating covered negative examples (see e.g. [Sammut and Banerji 86]).

Example 3.6: $\mathcal{VS}$ depends on the knowledge base.

Reconsider example 3.3. Assume that the aim is to learn the concept `grandfather`. Given the examples τ_2(`grandfather(morris, kate)`)= true and τ_2(`grandfather(luke, kate)`)= false, and the language of completely bound Horn-clauses L (see example 3.3), $\mathcal{VS}(KB, L, P, N) = \emptyset$. However, for the knowledge base KB', where $KB' = KB \cup \{c\}$ and c =`grandparent(X,Y)` ← `parent(X,Z)` ∧ `parent(Z,Y)`, $\mathcal{VS}(KB', L, P, N)$ contains `grandfather(X,Y)` ← `male(X)` ∧ `grandparent(X,Y)`. So, in the first case, the concept would not be learnable (the language bias is incorrect), while in the second case it can be learned.

Algorithm 3.4: Computing a clause for an uncovered positive example.

```
procedure handle_uncovered( p(t_1,...,t_n): uncovered positive example, L: language bias )
    S := JUS(KB, L, p(t_1,...,t_n))
    c' := void
    repeat
        delete a clause c from S
        if c does not cover negative examples
        then c' := generalize_clause( c )
        endif
    until c' ≠ void or S = ∅
    if c' ≠ void
    then assert c' in KB
        for all negative examples n of p obtained in the calls to generalize_clause
            of the repeat loop such that φ_2(KB, n) do
                call handle_covered( n )
        endfor
    else signal anomaly
    endif
endproc
```

Shapiro proves the following lemma:

Lemma 3.7 *(adapted from [Shapiro 83], theorem 3.2)*
If there is a negative example f covered by clauses in the knowledge base, there is a clause in the knowledge base that is incorrect in the intended interpretation (τ_2) and that is used to prove f.

The incorrect clause can be located by asking questions about the nodes of a proof-tree for the negative example f. Algorithm 3.5 describes Shapiro's method to locate the incorrect clause. Once the incorrect clause has been found, it is retracted from the knowledge base.

Algorithm 3.5: Specialization in Clint.

```
procedure specialize(f: covered false fact) returns incorrect clause
        b := a ground proof tree for f
        p := root of the tree
        while there is a son t of p such that τ2(t) is not known or false do
              if there is a son s of p such that τ2(s) = false
              then p := s
              else if there is a son s of p for which τ2(s) is not known
                   then ask the oracle for τ(s)
                   endif
              endif
        endwhile
        c := the clause that is used to prove p in the proof tree
        retract c from the knowledge base
        return c
endproc
```

Algorithm 3.5 first computes a proof tree (an AND-tree) for the false instance f. This can be done using well-known techniques such as SLD-resolution [Lloyd 87] or a forward chaining routine (see e.g. [Genesereth and Nilsson 87]). The algorithm then analyses the proof tree in a top-down manner by asking membership questions about the nodes in the tree. Notice that in general, the nodes of a proof tree may contain variables. These variables are interpreted to be universally quantified, which means that a node holds for all its instances. In order to generate membership questions, the variables in the proof tree are instantiated to ground terms; (this explains why b is a *ground* proof tree). This is done without loss of generality because of the universal quantification.

The algorithm starts querying the oracle about the sons of the proven false fact. The process continues until a node has been found that is false in the intended interpretation and whose sons are all true in the intended interpretation. The clause used in the proof for this node is then clearly incorrect. Hence, it is retracted from the knowledge base. Notice that the retracted clause may also be a fact.

The algorithm can be optimized if the distinction between basic and learned predicates is taken into account. As it is assumed that basic predicates are definitely correct, all

examples e for basic predicates are treated as if $\tau(e)$ is known and $\tau(e) = \varphi(KB, e)$. This reduces the number of queries to the oracle.

Shapiro [Shapiro 83] presents also a divide and conquer version of this algorithm. It optimizes the number of questions to the oracle. Instead of being linear in the depth of the proof tree, the divide and conquer strategy is logarithmic in the size of the proof tree (cf. [Shapiro 83, Sterling and Shapiro 86]). The optimized algorithm can – of course – also be used in the context of Clint. However, I preferred to present the top-down strategy because it is easier to understand. The correctness of the algorithm as presented in lemma 3.8 follows from the work of Shapiro (see [Shapiro 83, Sterling and Shapiro 86]).

Lemma 3.8 *(from [Shapiro 83, Sterling and Shapiro 86])*
Algorithm 3.5 retracts a clause that is incorrect in the intended interpretation (τ_2) if it starts from a fact f such that $\tau_2(f) = false$ and $\varphi_2(KB, f) = true$.

Algorithm 3.5 retracts a clause from the knowledge base. As a result, there may be positive examples that were covered by the clause, but which are not covered by the remaining clauses. Therefore it is necessary to verify whether the knowledge base satisfies these examples and to derive new clauses for positive examples that are not covered any more. This explains why algorithm 3.6 calls the procedure handle_example to verify these examples. The procedure handle_example will be explained later.

Algorithm 3.6: Recovering from a covered negative example.

```
procedure handle_covered( n: covered negative example )
    c := specialize( n )
    for all examples e such that covers_KB(c, e), τ2(e) = true and φ2(KB, e) ≠ true do
        call handle_example( e )  see algorithm 3.7
    endfor
endproc
```

3.5.4 Clint's basic algorithm.

In this section, the specialization and generalization algorithm are put together to form a working whole.

Problem specification 3.1 can be divided into two special cases according to the nature of the example for which the knowledge base is incorrect. The first case occurs when the example is positive but not covered and the second case arises when the example is negative but covered. These two cases are precisely the ones that have been treated in the previous sections. If however, the example is positive and covered or negative and not covered, it is not desirable to modify the knowledge base. This motivates algorithm 3.7, which is the basic algorithm of Clint. As modifications to one predicate may also affect other predicates, it is necessary to verify the predicates that depend on a modified one. (For the definition of *depending on* see definition 3.18).

A second enhancement, needed to guarantee the convergence of Clint, concerns the use of the set of positive examples called postpone. This enchancement will be extensively analyzed in section 3.6. The reason for this enhancement is that at a certain point in time during the learning there may be positive examples for which $\mathcal{VS}(KB, L, \{p\}, N) = \emptyset$. This is an anomaly. Anomalies are signaled by algorithm 3.4. If an anomaly arises, it means that there is missing or incorrect knowledge that makes it impossible to derive a correct concept-description. This also implies that Clint is unable to directly cope with the situation. Therefore, it postpones the treatment of these positive examples until it has acquired more knowledge (i.e. its knowledge base has been modified). Then, it will again attempt to learn a clause for these examples and by doing so it may recover from the anomaly.

Algorithm 3.7: Basic algorithm of Clint.

```
procedure Clint
    postpone := ∅
    while the oracle wants to input examples do
        read the next example e and its truthvalue τ(e)
        call handle_example(e)
    endwhile
endproc

procedure handle_example( p(t1,...,tn): example )
    if τ2(p(t1,...,tn)) ≠ φ2(KB,p(t1,...,tn))
    then if τ2(p(t1,...,tn)) = false
        then call handle_covered( p(t1,...,tn) )
        else call handle_uncovered( p(t1,...,tn), L )
            if handle_uncovered signals anomaly
            then add p(t1,...,tn) to postpone
            endif
        endif
        for all predicates q that depend on the predicate p do
            for all examples e of q do
                call handle_example(e)
            endfor
        endfor
    endif
    if ∃e' ∈ postpone such that KB was modified
        since the previous attempt to call handle_uncovered(e', L)
    then delete e' from postpone
        call handle_example( e', L )
    endif
endproc
```

Notice that conjunctive as well as disjunctive concepts can be learned using Clint. Indeed, if after learning one clause for a predicate, the user supplies an uncovered positive example of the predicate, a new clause for the same predicate may be learned.

3.5.5 Some sessions with Clint.

A prototype of algorithm 3.7 without the use of postponing has been implemented in BIMprolog for some specific languages (see chapter 4 for the language specifications). A simple session using this algorithm and the family knowledge base of example 3.3 is shown in example 3.7 and an example that illustrates the potential use of Clint for program synthesis in example 3.8. This last example is beyond the scope of the implementation of Clint because of the requirement of a perfect query answering mechanism (see also section 3.7). These sessions indicate that Clint generates an acceptable number of relevant questions to the user. Other sessions will be shown in chapters 4 and 8.

Example 3.7: An example session with Clint.

Suppose the knowledge base of example 3.3 is used and that Clint has to learn the `father` concept in the language of completely bound Horn-clauses. Remember that the generalization operator is $\mathcal{L}_g$ (see definition 3.33)! A session to learn `father` goes as follows: (user input is in **bold** while system output is in `typewriter` font):

```
?-learn.
Next example ?  @ father(jeffrey,kate).
Classification ?@ true.
Starting clause for father(jeffrey,kate) in L:
```

$$\texttt{father(X,Y)} \leftarrow \texttt{male(X)} \wedge \texttt{female(Y)} \wedge \texttt{parent(X,Y)}$$

This clause can be generated using algorithm 3.2. The system then tries to delete literals, starting with `male(X)`:

```
Is father(rose, mary) true ?  { male(X) } deleted @ no.
Is father(morris, jeffrey) true ?  { female(Y) } deleted @ yes.
Generalized clause:
```

$$\texttt{father(X,Y)} \leftarrow \texttt{male(X)} \wedge \texttt{parent(X,Y)}$$

As the example is positive, the system generalizes. Now, it cannot generalize any further as the deletion of `parent(X,Y)` would make the clause not range restricted and the deletion of `male(X)` would give rise to a covered negative example. Therefore the system decides that it has found a maximally general clause and asserts it in the knowledge base:

```
No more generalizations.  Asserted clause:
```

$$\texttt{father(X,Y)} \leftarrow \texttt{male(X)} \wedge \texttt{parent(X,Y)}$$

Example 3.8: Synthesizing prolog programs using Clint's basic algorithm

Assume the following knowledge base is used:

```
nil([]) ←
list(X,Y,[X|Y]) ←
eq(X,X) ←
```

Suppose that the language $L_{d(1)}$ is used. $L_{d(n)}$ contains a clause c if c does neither contain function nor constant symbols, c is range restricted and linked and the variables in $head(c)$ are decomposed at most n times. A variable v in a clause c is decomposed n times if v occurs in a literal $F(x_1, ..., x_k, v) \in body(c)$ where F is a predicate defining a functor f/k ($k \neq 0$) (see also chapter 4) and at least one of the variables x_i is decomposed $n-1$ times. An example of such a predicate $F/k+1$ is the predicate `list/3`. For convenience, we will again use the generalization operator $\mathcal{L}_g$. Given these assumptions and the use of a perfect query answering procedure, the following session could be derived using algorithm 3.7.

```
?-learn.
Next example ?  @ append([],[a,b],[a,b]).
Classification ?  @ true.
Starting clause for append([],[a,b],[a,b]) in L_d(1)

      append(X,Y,Z) ← nil(X) ∧  list(CARY, CDRY, Y) ∧  list(CARZ,CDRZ,Z)
            ∧  eq(Y, Z) ∧  eq(CARY, CARZ) ∧  eq(CDRY, CDRZ)
```

Although in principle the starting clause should also contain the literals `eq(Z, Y)`, `eq(CARZ, CARY)` and `eq(CDRZ, CDRY)` and `eq(V, V)` for all variables occurring in the starting clause, these literals can be automatically deleted if it is known that `eq` is a symmetric and reflexive predicate. In the actual implementation symmetric and reflexive predicates can be declared to the system. Furthermore, the predicate `eq` is considered to be a special predicate in the sense that no literals `eq(X, Y)` are allowed in a clause unless both variables `X` and `Y` occur somewhere else in the clause. Assume that the system first tries to delete the literal `nil(X)`. This results in a kind of explicit unification (see also our note on implementing generalization operators).

```
Is append([a],[a,b],[a,b]) true ?  { nil(X) } deleted @ no.
Is append([],[a,a],[a,b]) true ?  { eq(Y, Z), eq(CDRY, CDRZ) } deleted
@ no.
Is append([],[b,a],[a,a]) true ?  { eq(Y, Z), eq(CARY, CARZ) } deleted
@ no.
Is append([],[],[]) true ?  {list(CARY, CDRY, Y), list(CARZ, CDRZ, Z),
eq(CARY, CARZ), eq(CDRY, CDRZ)} deleted @ yes.
```

Notice that the algorithm might as well have generated `append([],a,a)`. However, this is also true in the normal intended interpretation of append.

```
Generalized clause:
```

```
append(X,Y,Z) ← nil(X) ∧ eq(Y,Z)

No more valid generalizations. Asserted clause:

append(X,Y,Z) ← nil(X) ∧ eq(Y,Z)

Next example ? @ append([a,b],[c],[a,b,c]).
Classification ? @ true.
Starting clause for append([a,b],[c],[a,b,c]) in L_d(1):

append(X,Y,Z) ← list(CARX, CDRX, X) ∧ list(CARY, CDRY, Y) ∧
      list(CARZ, CDRZ, Z) ∧ eq(CARX, CARZ) ∧ nil(CDRY)
      ∧ append(CDRX, Y, CDRZ)
```

Notice that the recursive condition on append is found because the definition of coverage (see definition 3.13). The definition stated that the clause itself may also contribute to proving properties in the body such as append([b],[c],[b,c]). This last fact was then provable because append([],[c],[c]) is provable using the already learned clause.

```
Is append([a],[b,c],[a,b,c]) true ? { nil(CDRY) } deleted
@ yes.
Generalized clause:

append(X,Y,Z) ← list(CARX, CDRX, X) ∧ list(CARY, CDRY, Y) ∧
      list(CARZ, CDRZ, Z) ∧ eq(CARX, CARZ) ∧
      append(CDRX, Y, CDRZ)

Is append([a],[],[a]) true ? { list(CARY, CDRY, Y) } deleted @ yes.
Generalized clause:

append(X,Y,Z) ← list(CARX, CDRX, X) ∧ list(CARZ, CDRZ, Z) ∧
      eq(CARX, CARZ) ∧ append(CDRX, Y, CDRZ)

Is append([b],[],[a]) true ? { eq(CARX, CARZ) } deleted @ no.
No more generalizations. Asserted clause:

append(X,Y,Z) ← list(CARX, CDRX, X) ∧ list(CARZ, CDRZ, Z) ∧
      eq(CARX, CARZ) ∧ append(CDRX, Y, CDRZ)
```

The system discovers that it cannot generalize further without covering negative examples or going outside the language. Notice that the found program is correct for append.

3.6 Theoretical aspects of Clint's basic algorithm.

In this section, the convergence behaviour of Clint is analyzed while assuming the existence of a perfect query answering procedure. It is shown that – under certain conditions – Clint identifies concepts in the limit. There is a difference in convergence behavior according to whether the learned clauses are independent of each other or not. Independent clauses can be learned without postponing while dependent clauses are only learnable by Clint when there is a possibility to postpone the learning of certain examples.

3.6.1 Inductive inference: some concepts.

The field of inductive inference [Angluin and Smith 83, Biermann 86] studies the mathematical properties of learning problems and algorithms. For a particular algorithm A, researchers in this field want to find answers to the following questions:

1. *What can be learned using algorithm A ?*
2. *What is required to learn a specific concept using algorithm A ?*

To address the first question, researchers have developed different criteria of learnability (see e.g. [Biermann 86, Angluin and Smith 83] for a summary) such as identification in the limit [Gold 67], finite identification and probably approximate correct learning [Valiant 84]. The notion of identification in the limit is adapted towards the specific needs of Clint as follows (the resulting framework is very much related to the one of [Shapiro 83]):

Definition 3.39 *A presentation P of a set of concepts C is a possibly infinite sequence of examples e and their truthvalue $\tau(e)$ such that for all examples e of predicates in C, the example e together with its truthvalue $\tau(e)$ occurs in the sequence P.*

As their name indicate, presentations are presented to a learning system, which should find (correct) definitions of the predicates in C.

Definition 3.40 *An algorithm identifies a set of concepts C error-free in the limit starting from a knowledge base KB and a presentation P of C, if*

1. *for all examples $e \in P$, after the system has acquired e, it always outputs a knowledge base that satisfies e*
2. *there is a (finite) time after which the learned knowledge base KB' is definitely correct for the concepts in C and KB' will not be modified afterwards*[3]

The second condition in this definition demands that φ coincides with τ.

This definition means that the algorithm must converge on a correct knowledge base but it does not have to know when it has converged. So, the time of convergence is unknown to the algorithm (and the user). If the time of convergence is known, the algorithm

[3]Logically speaking, one might believe that the condition *KB' will not be modified afterwards* is unnecessary. I have only included it for completeness, as in the literature one distinguishes behaviourally correct identification, i.e. identification without the condition, and exact identification, which corresponds to our notion. This is relevant because some classes of concepts can be behaviourally identified but not exactly.

achieves *finite identification* [Angluin and Smith 83]. Furthermore, after receiving an example from the user, the system must output a knowledge base that satisfies this example and all previously acquired ones. Definition 3.40 is the usual definition of identification in the limit. In the framework of Clint, we will also consider a relaxation of this definition: *postponed* identification.

Definition 3.41 *An algorithm identifies a set of concepts C with postponing in the limit starting from a knowledge base KB and a presentation P of C, if there is a (finite) time after which the learned knowledge base KB' is definitely correct for the concepts in C and KB' will not be modified afterwards.*

In postponed identification, the system may output a knowledge base that only satisfies part of the example-set. However, in the end, i.e. after a finite time, the output knowledge base must satisfy all examples!

The convergence behavior depends in a natural way on the language being used and the starting knowledge base. This motivates the following definition:

Definition 3.42 *A set of concepts C is expressible in terms of a language bias L and a starting knowledge base KB if it is possible to correctly define C by adding clauses in L to KB.*

To address the second question, researchers (e.g. [Shapiro 83]) impose certain restrictions on the starting knowledge base KB and the allowed presentations. Usually, they define *acceptable* presentations, i.e. presentations that are sufficient to learn a set of concepts.

3.6.2 Convergence of Clint.

During the discussion of Clint's convergence, it will be assumed that a perfect query answering mechanism (see section 3.5) is employed. Later, in section 3.7, the implications of this assumption will be studied in detail.

The difficulty with analyzing the convergence behavior of Clint is that the versionspace VS used by Clint is a function of the knowledge base. This was already illustrated in example 3.6, which clearly shows that Clint depends on the order in which the predicates are learned. Order dependency is inevitable if the learning procedure uses what it already knows. The example also illustrates that Clint is unable to learn certain concepts. A concept cannot be learned if an *anomaly* occurs (see section 3.5), i.e. if at a certain point in time, there is a positive example p and a set of negative examples N, such that $VS(KB, L, \{p\}, N) = \emptyset$.

Anomalies indicate that there is vital information, necessary to learn the concept, missing. They also bring the system in a state from which it cannot proceed. A naive approach might try to proceed by changing the order in which the examples and predicates are handled. However, there is no guarantee that this computationally expensive method will always work. Let us therefore address the following questions:

1. *How do anomalies arise ?*
2. *Can they be avoided ?*
3. *Is it possible to recover from anomalies ?*

How anomalies arise

An anomaly arises when the system processes an uncovered positive examples p (in handle_uncovered) and tries to find a clause for it, but $\mathcal{VS}(KB, L, \{p\}, N) = \emptyset$. There are two classes of situations that may lead to anomalies:

- The user supplies an uncovered positive example p to the system such that $\mathcal{VS}(KB, L, \{p\}, N) = \emptyset$.
- The system retracts an incorrect clause c from the knowledge base KB because c covers a negative example n ($covers_{KB}(c, n)$). After the retraction of the clause c, Clint verifies the positive examples that were covered by c (see procedure handle_covered). Some of these positive examples p may result in an anomaly because $\mathcal{VS}(KB - \{c\}, L, \{p\}, N \cup \{n\})$ may be $\emptyset$ for the resulting knowledge base $KB - \{c\}$.

Anomaly Avoidance

A problem, revealed by the second cause for anomalies, is that at a certain point in time there can be clauses explaining p in $\mathcal{VS}(KB, L, \{p\}, N)$ while at a later point in time $\mathcal{VS}(KB', L, \{p\}, N')$ may be $\emptyset$. $\mathcal{VS}(KB', L, \{p\}, N')$ may become empty after the retraction of a clause from KB ($KB' \subset KB$) or the addition of a new negative example to N'. Example 3.9 illustrates this. The question is whether this can be avoided. The answer is yes provided that assumption 3.2 holds as indicated in lemma 3.9.

Example 3.9: How anomalies may arise

Reconsider the knowledge base KB of example 3.3. Assume that KB' is used, where KB is obtained from KB by retracting `male(jeffrey)` ← . Given this situation, imagine that Clint learns an incorrect clause for `father` starting from the example τ_2(`father(jeffrey, kate)`) = true. The clause which would be learned could then be `father(X,Y)` ← `parent(X,Y)`. If the user would now supply a covered negative example such as e.g. τ_2(`father(rose, mary)`), the clause for `father` would be retracted. When deriving a clause for `father(jeffrey, kate)` the system runs into an anomaly. The example also reveals that anomalies are often due to missing knowledge, in this case `male(jeffrey)` ← .

Assumption 3.2 *Anomaly avoidance.*
For all positive examples p

1. *at the time p is first encountered, $\mathcal{VS}(KB, L, \{p\}, N)$ contains a correct clause c*
2. *the knowledge base $KB \cup \{c\}$ allows to derive a proof for p using only clauses that are correct*

Lemma 3.9 *In the context of* Clint, *no anomalies can arise when the anomaly avoidance assumption holds.*

Proof : The first cause for an anomaly is excluded by assumption (1).

The second cause is excluded because Clint never retracts clauses that are correct in the intended interpretation (see lemma 3.8). Therefore c will always remain in $VS(KB', L, \{p\}, N')$ because no clause needed for c to cover p will ever be retracted from the knowledge base. This holds for all N' and for all KB' derived from KB.

□

The anomaly avoidance assumption states precise conditions under which a positive example is permitted in a presentation. If the assumption holds, a negative example is allowed to occur at any time in a presentation.

Lemma 3.9 looks nice, but it does not solve the problems of anomalies. The reason is that Clint generates its own examples. This is a complication as Clint has no control over the above listed assumption. The point is that the anomaly assumption requires complete knowledge about the intended interpretation. Now, Clint never obtains complete knowledge about the intended interpretation. Indeed, the only knowledge about the intended interpretation it acquires, are the examples. Even if it could be assumed that the user supplied examples satisfy the assumptions, Clint would be unable to guarantee that the examples it generates satisfy the assumption[4].

There is however one interesting special case, which allows to avoid anomalies. This is treated in theorem 3.3, which was also proven in [De Raedt and Bruynooghe 88]. Observe that theorem 3.3 states acceptable conditions for learnability. Indeed, it basically says that a concept can be learned if there is enough correct information in the knowledge base to express the concept in L.

Theorem 3.3 *Convergence of a single non-recursive concept.*
Any non-recursive predicate q will be error-free identified using Clint from any presentation of q if the language bias L is correct for the concept and the knowledge base is definitely correct for the predicates that occur in a correct concept-description of q in L w.r.t. the knowledge base.

Proof : The proof follows from (1) the fact that no anomaly can arise and (2) that the correct concept-description will be found if enough examples are supplied.

For (1), it is sufficient to show that the anomaly avoidance assumption holds. If the language bias is correct and the knowledge base is definitely correct for the predicates occurring in a correct concept-description in L w.r.t. KB, there must be a correct clause $c \in VS(KB, L, \{p\}, N)$ for all positive examples p. This implies that there is for each positive example, a correct proof in terms of the knowledge base and the clause c.

(2) follows from the observations that Clint will learn a clause for each user-supplied uncovered positive example, that it will never retract a correct clause and that Clint will retract all clauses that cover a negative example as long as it encounters enough negative examples. As all clauses that cover a negative example will eventually be retracted and there is only a finite number of clauses in L, only clauses that do not cover negative examples will be learned. However, if the starting knowledge base is correct in the intended interpretation, clauses that do not cover negative examples are correct. Therefore, if Clint is supplied with enough positive and negative examples, it will learn a correct clause for

[4] Of course, one might want to rely on some kind of oracle for deciding whether the assumption holds for a certain example or not. However, doing so would render the example generation process useless.

each positive example, and hence also a correct concept-description. This will be done after a finite number of examples because the number of clauses in L is finite, a finite number of positive and negative examples is sufficient to identify one correct clause and a correct concept-description has only a finite number of clauses. □

Theorem 3.3 does not hold when q is a recursive predicate because in order to prove a positive example p one may have to rely on more than one clause of the predicate q. If some clause for q has not yet been learned correctly, a correct proof may not exist for p. This may give rise to anomalies. As an example, learning a correct clause for `append([a,b], [c], [a,b,c])` is impossible when the base clause is not known (cf. also example 3.8). A similar argument applies when two predicates are learned such that one predicate depends on the other. In our theoretical model of concept-learning $\mathcal{DVS}$ the problem would be solved by adding two clauses to the knowledge base. Clint – however – only adds one clause at a time to the knowledge base.

In general, we cannot do much to avoid anomalies. The reason is that when there are incorrect clauses in the knowledge base, Clint may run into anomalies. Anomalies may arise because an incorrect clause is potentially overly general.

Definition 3.43 *A concept (or a clause) is overly general if it covers a (possibly unknown) negative example.*

An incorrect clause $p \leftarrow q_1 \wedge ... \wedge q_k$ need not always cover negative examples but may do so when the definitions for the q_i are modified. Once there are overly general clauses Clint may run into anomalies because overly general clauses are the second cause for anomalies. As a consequence, if anomalies are to be avoided one should not learn incorrect clauses. The question then is whether it is feasible for Clint to avoid to learn incorrect clauses. Unfortunately, the answer to this question is *no* because doing so requires the user to contribute too much to the learning process. Let us indicate why:

A prerequisite to avoid learning incorrect clauses is to avoid learning overly general clauses (or concept-descriptions).

- In [De Raedt and Bruynooghe 88], it was shown that overgeneralization of disjunctive concepts can be avoided if the computation of a clause covering a positive example p proceeds in two steps: (1) find a correct starting clause c in $\mathcal{JUS}(KB, L, \{p\}, N)$ and (2) generalize c by applying a safe generalization mechanism. The safe generalization mechanism requires algorithm 3.3 to generalize only when *all* examples that are covered by c' and not by c are positive! This generalization strategy is not feasible in practice as it would result in too many questions to the oracle. (In cases where functors are allowed, one might even have to answer an infinite number of questions.) Furthermore, if $|\mathcal{JUS}(KB, L, \{p\}, N)| > 1$, a special algorithm should be designed in order to find the correct justifications in $\mathcal{JUS}(KB, L, \{p\}, N)$.

- A second approach relies on the user to assure that the system does not overgeneralize. Indeed, the user can prevent overgeneralization if she assures that the system knows enough negative examples before supplying a positive example to the system. If Clint knows that a clause covers a negative example, it will never attempt to generalize to that clause. Following this approach, the user is requested to supply enough negative examples such that no clause in $\mathcal{VS}(KB, L, \{p\}, N)$ is overly general when the positive example p is supplied to the system.

These approaches to overgeneralization are very demanding for the user. Furthermore, even if they would be followed, it is not clear whether there would exist an acceptable presentation to learn any knowledge base expressible in the language L. Secondly, anomalies could still occur if the user would not present an acceptable presentation to the system. Therefore it is necessary to devise recovery measures for anomalies.

Anomaly Recovery

Algorithm 3.7 was designed to take the possibility of anomalies into account. Upon the occurrence of an anomaly, the system adds the responsible positive example to the list of postponed positive examples and proceeds as if there were no problem. When it later revises its knowledge base, it will reconsider the postponed examples, and try to derive clauses for them. Because of this, the system may sometimes generate knowledge bases that do not satisfy all known examples. However, postponing certain examples allows to prove postponed identification in the limit for Clint.

Theorem 3.4 *Any set of concepts C, expressible in L and KB, will be postponed identified from a definitely correct starting knowledge base KB and from any presentation P of the concepts C.*

Proof : It is sufficient to prove that: (1) no overly general rule will remain in the knowledge base and (2) that for each positive example, there will eventually be a time after which it is covered by a correct clause. If no clause in the knowledge base is overly general and all positive examples in the intended interpretation are covered, the knowledge base must be correct.

(1) directly follows from the observation that any overly general clause covers a negative example; if such a negative example is supplied to the system, it will be removed from the knowledge base (lemma 3.7). As a conclusion, if enough negative examples are supplied to the system, all overly general clauses will be removed from the knowledge base. Because P is a presentation, it follows that all examples will eventually be supplied to the system.

To prove (2), observe that a positive example p is provable in the target knowledge base and that a proof (an AND-tree) consists of a finite number of true nodes. If all nodes below p are derivable in the current knowledge base, it must be the case that – according to our assumptions – $VS(KB, L, \{p\}, N)$ contains a correct clause, therefore $VS(KB, L, \{p\}, N) \neq \emptyset$. At that point in time, p can be correctly handled. So, if all these nodes are supplied as positive examples to the system and a clause has been learned for all these positive examples, it is sure that p can be treated.

It remains to be shown that there is a good order of treating the examples such that all nodes, needed to prove p, can be learned before handling p. Clearly, there is such an order as given by the proof tree. First the nodes that are provable in terms of the given correct knowledge base, are learnable. These nodes n only rely on correct knowledge. Therefore, it must be the case that $VS(KB, L, \{n\}, N)$ contains a correct clause. Given that those nodes are provable, it is possible to learn a clause for at least one father node of one of these nodes. As this process can be repeated and no node occurs as a subgoal of itself in a proof, induction allows to conclude that our original node p can be learned, after all its subnodes are handled. Also, Clint will find a good sequence of processing the examples.

This is guaranteed by the fact that it will learn a clause whenever $\mathcal{VS}(KB', L, \{e\}, N) \neq \emptyset$ for a particular positive example e.

One remaining point to note is that although Clint will not always derive a correct clause for an example p, there must be a time after which there are only correct clauses in $\mathcal{VS}(KB, L, \{p\}, N)$. This is the case when there are enough known negative examples for the involved predicate. As this will eventually be the case for all subnodes of the goal p, and for p itself, Clint learns correct clauses in the limit. □

This proof and the above discussion also provides some heuristics to the user for decreasing the number of anomalies:

- Supply as much negative examples as possible. This allows to reduce the number of overly general clauses in $\mathcal{VS}(KB, L, \{p\}, N)$ and the probability of having to retract the found result later. As shown above, retracting clauses may result in later anomalies.
- For the same reason, if a clause is incorrect, try to supply a negative example that is covered by the incorrect clause.
- Supply the facts covered by base clauses before the ones that are covered by recursive clauses. In this way, it is also easier to learn recursive concepts (see example 3.10).

Example 3.10: Illustrating how to learn mutually recursive concepts

Suppose the aim is to learn the following definitions:

(1) `male_ancestor(X,Y)` ← `father(X,Y)`
(2) `male_ancestor(X,Y)` ← `father(X,Z)` ∧ `male_ancestor(Z,Y)`
(3) `male_ancestor(X,Y)` ← `father(X,Z)` ∧ `female_ancestor(Z,Y)`
(4) `female_ancestor(X,Y)` ← `mother(X,Y)`
(5) `female_ancestor(X,Y)` ← `mother(X,Z)` ∧ `male_ancestor(Z,Y)`
(6) `female_ancestor(X,Y)` ← `mother(X,Z)` ∧ `female_ancestor(Z,Y)`

and that the predicates `father` and `mother` are known. A good strategy to learn the above clauses is to start learning from examples of `male_ancestor` or `female_ancestor` that are covered by the base cases (1,4). Afterwards, an example that can be proven using only clauses (1) and (2) could be given. Then an example that is covered by (3) and (4). After this, any example covered by clause (5) and then any example covered by (6) could be input. During the learning care should be taken not to overgeneralize any clause.

3.7 (Im)perfect query answering procedures.

In this section, we investigate the issues concerning perfect query answering mechanisms. Remember from definition 3.37 that a perfect query answering mechanism always finds a

correct answer after a finite computation to a query of the form $\leftarrow q_1 \wedge ... \wedge q_n \wedge \neg(p_1 \wedge ... \wedge p_k)$. It is well-known [Genesereth and Nilsson 87] that in general there is no such mechanism. It will be shown that Clint needs a perfect query answering mechanism to terminate. Furthermore, because there are no general perfect query answering mechanisms, we will indicate how Clint's algorithms might be enhanced to cope with the problem of an imperfect query answering mechanism. The ideas concerning these possible extensions are preliminary because of the difficulty of the problem.

3.7.1 Problems with perfect query answering mechanisms.

For full first order logic or Horn-clause logic there is no perfect query answering procedure [Genesereth and Nilsson 87]. There are two problems with perfect query answering procedures:

- The problem of answering a query $\leftarrow p_1 \wedge ... \wedge p_n$ is only semi-decidable [Genesereth and Nilsson 87]. This means that there is a procedure that will generate all answers to the query but the procedure may not always terminate.
- The second problem is that the general form of queries, generated by Clint also uses negation as failure. A well-known drawback of negation as failure as implemented in Prolog, is that queries containing negative literals are only handled correctly when the negative literals are ground at call time [Naish 86]. This can easily be avoided by some delaying mechanism as e.g. in Nu-Prolog [Naish 86], but still the query can flounder.

The bad news concerning query answering mechanisms, is that the combination of these two difficulties makes it very hard (impossible ?) to devise an elegant solution for the general problem.

The good news is that there is a perfect query answering mechanism for an important subset of first order logic, in casu Datalog [Ceri, Gottlob and Tanca 90]:

Definition 3.44 Datalog *is the subset of pure* Prolog *without functors of arity* $k > 1$, *i.e. definite clauses without functors.*

Datalog is often used in deductive database and knowledge base contexts.

3.7.2 The need for a perfect query answering mechanism

A perfect query answering mechanism is needed by Clint to guarantee termination:

Theorem 3.5 Clint *as presented in section 3.5 always terminates if it uses a perfect query answering procedure and a finite language* L.

Proof : The procedures of Clint always terminate, under these conditions, because at each moment in time there is only a finite number of known examples and a finite number of candidate-clauses in L. Termination is only guaranteed when a perfect query answering procedure is used to (1) verify whether a clause c covers an example e and (2) to generate examples, i.e. to construct an example that is covered by one clause but not by another one. Process (1) can be implemented as follows: find θ such that $head(c)\theta = e$ and answer

the query $\leftarrow body(c)\theta$. Process (2) was explained when algorithm 3.3 was presented. Both processes always terminate when a perfect query answering procedure is used. □

So, on the one hand, Clint needs a perfect query answering mechanism, and on the other hand, there is no such mechanism except for a subclass of Prolog, i.e. Datalog. Let us therefore investigate whether Datalog is sufficient for our purposes and whether there are other ways around the problem.

To the best of my knowledge, there are only two learning systems that use pure Prolog as a representation mechanism: Cigol [Muggleton and Buntine 88] and the Model Inference System [Shapiro 83]. This seems to indicate that there is not always a need for such a rich representation formalism. Indeed, most machine approaches [Michalski, Carbonell and Mitchell 83, Michalski, Carbonell and Mitchell 86, Kodratoff and Michalski 90] to machine learning use a representation formalism that is less expressive than Datalog. Furthermore, there are many interesting learning situations in which Datalog is sufficient (see for example [Sammut and Banerji 86, Tecuci and Kodratoff 90, Morik 89]). This seems to indicate that Datalog is sufficient for our purposes.

Secondly, the systems Cigol and the Model Inference System are based on different properties and assumptions as compared with Clint. Cigol seems to use a depth-bounded interpreter to avoid infinite loops. On the other hand, Shapiro's Model Inference System requires an oracle to recover from non-terminating programs. The oracle must be able to answer questions of the form: is goal g_1 allowed as a subgoal of goal g_2 ?

This discussion leads to the question whether Clint can be extended in order to handle imperfect query answering procedures. Some possible directions to achieve this are outlined in the next subsection.

3.7.3 Allowing imperfect query answering procedures.

It seems possible to adapt Shapiro's approach to avoid the first problem with query answering mechanisms. One then has to use a depth-bounded interpreter [Sterling and Shapiro 86] and when the depth-bound is exceeded, query the user to find a diverging procedure.[5] However, there is no guarantee that this solves the problems with negation.

The problems with negation might be solved by applying a constructive form of negation along the lines of e.g. [Chan 88, Puget 89].

Another possibility that might be of help is to use type and mode information. Type information states for each argument position of a predicate what the type of the argument is. Mode information encodes information about the way terms are instantiated when solving a goal. This is illustrated in example 3.11.

Type and mode information can be used to reason about the order in which subgoals of a query should be solved. A technique related to this is applied in Nu-Prolog [Naish 86]. Given certain declarations, Nu-Prolog will find a good order in which to execute goals and by doing so, it can avoid certain infinite loops.

In an experiment with Clint in the Finix environment [Zinnsmeister and Hoppe 89], Clint was enhanced with type and mode information. The mode information was used to determine the order of the subgoals in the query. When no subgoal could be extended according to the mode information, some variables in the goals were instantiated to ground terms derived from the type information. After instantiating enough terms, it was possible

[5] There are also more advanced methods that use run-time information about the query.

Example 3.11: Illustrating types and modes in logic programming.

Consider the normal definition of the append predicate:

```
append([], X, X) ←
append([X|Y], Z, [X|W]) ← append(Y, Z, W)
```

Type information would state that all arguments of the predicate append are of type `list`, where this type can be defined as:

```
list([]) ←
list([X|Y]) ← symbol(X) ∧  list(Y)
```

The first clause states that the empty list is a list; whereas the second one states that any term that uses the list functor and has as first argument a symbol and as second argument a list, is a list.
Mode information about append would state that if the first and the second arguments are ground and the third argument is a variable, that this third argument will be instantiated (ground) after solving the goal. Similarly, if the third argument is ground, the first two arguments will be ground after solving the goal.

---

to expand certain subgoals. Although this approach solves our problems in theory, it has some important drawbacks:

- Type and mode information has to be declared or derived for all predicates in the knowledge base.
- Instantiating variables to ground terms, cannot be done exhaustively as there may be an infinite number of candidate ground terms. In the Finix application, only a finite number of terms was generated by taking into account some complexity measure on terms. Instantiating variables to ground terms in this manner, is computationally very expensive.

Although a number of possible approaches to the problem of perfect query answering procedures have been indicated, the problem is unsolvable in general. Some approaches like Shapiro's one and the one based on types and modes may be used to circumvent the general problem. However, in practice these approaches do not seem to be feasible in our context of learning apprentice systems. It is not acceptable to ask the user to supply type and mode information nor to require her to decide which goals may arise as subgoals of another goal. Furthermore, as Datalog seems sufficient for many interesting learning tasks, I will assume that hereafter Datalog is used. This implies that in the rest of this thesis, I will assume that no functors are used. Also, because the techniques to solve the problem belong to the field of logic programming and automated reasoning and do not directly contribute to the field of machine learning, I will not elaborate on them any further in this thesis.

3.8 Complexity of Clint.

Although Clint's complexity will not be formally analyzed, some factors influencing its complexity are identified. The reason for not formalizing Clint's complexity is that it relies too much on the learning situation and the available knowledge base.

From the complexity point of view, there are two different aspects of Clint that deserve attention: the time complexity of the learner and the number of examples needed to identify concepts.

Experience with the implementation of Clint and analysis of the algorithms of Clint show that the procedure generalize_clause of algorithm 3.4 consumes most of the resources. The reason is that the generation of justifications (algorithm 3.3) and the procedure to locate and retract incorrect clauses are quite efficient, in time as well as in the number of questions. Furthermore the number of consistent justifications for a specific positive example is usually not large (see also chapter 4).

Therefore I will now try to provide some insight in the factors influencing the complexity of the procedure generalize_clause and to roughly estimate some of the involved factors. To make the presentation less abstract, it will be assumed that the operator $\mathcal{L}_g$ is used by Clint. Analysis reveals that the complexities depend very much on (1) the knowledge base being used, (2) the size of the justification and (3) the nature of the obtained examples.

The complexities depend on the knowledge base as Clint generates queries to the knowledge base in order to verify whether examples are covered or not and to generate examples. Therefore, it is interesting to separate the time complexity of Clint into (1) the time needed by Clint without the time needed to answer queries and (2) the number of queries generated to the knowledge base.

The number of queries to the knowledge base to generalize a clause c is the sum of

- the number of times the system verifies whether a candidate generalization of c covers a negative example. This number is at most the number of considered candidate generalizations $\times$ the number of negative examples.
- the number of times the system attempts to generate an example. This number is at most equal to the number of considered candidate generalizations.

Observe that the queries needed to generate an example require usually much more computation than the ones that verify the coverage of an example. The reason is that in the second type of query, the query is usually more instantiated than for the first type of query. Furthermore the second type of query also uses negation as failure and contains about twice as much literals (see the proof of theorem 3.5).

Anyway, the most important factor in these estimations is the number of considered candidate generalizations. The worst case for this factor is very clear: exponential in the size s of the clause c as all subsets of the clause might be considered to obtain generalizations when the operator $\mathcal{L}_g$ is applied. However, the worst case analysis of this factor is useless because the system will only consider all these subsets if all literals in the clause c are redundant. A literal q is redundant in a clause c if c is equivalent to $c - \{q\}$ given the knowledge base, i.e. if c and $c - \{q\}$ are syntactic variants. Secondly, the worst case analysis does not take into account that negative examples are used to prune the search. Therefore it seems more feasible to estimate the number of candidate generalizations as being s^2 in the number of literals s of the clause c. This estimation is

correct if for all generalizations g of c, at least one of the literals in g is not redundant. This assumption is plausible in practice (c.f. the examples in this thesis). Given the assumption, only one literal has to be deleted from c in order to find an example (this literal is found after at most s trials). If the example is positive, the clause is generalized and a clause of size $s-1$ is found. Otherwise, if the example is negative, it is known that the involved literal should never be deleted from the clause. This means that in further steps, the system should only consider $s-1$ literals. Since in both cases the number of literals to consider for deletion is decreased by 1 after generating 1 example (and at most s trials), the total number of candidate generalizations is at most s^2. Under the same literals, the number of questions to the oracle for learning one clause is s.

Although this analysis does not take into account that incorrect clauses may be learned and may have to be retracted, that a concept may be disjunctive, and that learning one concept may influence other ones, I believe that the above estimation allows to conclude that for reasonable knowledge bases and languages, Clint works reasonably efficient and generates an acceptable number of questions.

This claim is also supported by some practical experiences with Clint where complex knowledge bases were used. Consider e.g. the chess examples in chapter 8. In that example, predicates such as `same_row`, `adjacent`, and `same_diagonal` succeed for a large number of facts. Yet the number of questions and the time consumed by Clint are acceptable. Also, Erik Baes [Baes 90] sketches a chess knowledge base where a particular class of chess endgames was encoded. Clint was given a rich enough vocabulary to learn the concept of a winning situation. Using the vocabulary, about 25000 different board situations could be distinguished. Yet Clint learned the concept in a few minutes of cpu-time.

3.9 Clint as an instantiation of Gencol

In order to support the claims of generality made about Gencol in chapter 2, it is shown how Clint can be obtained as an instantiation of Gencol.

This can be realized using the same strategy as for the AQ system of [Michalski 83] (see section 2.4), There it was shown how AQ can be obtained using two different instantiations of Gencol : one to learn disjunctive concepts and one to learn a single conjunct of a concept. Clint can be divided into two similar parts: one to learn a complete concept-description consisting of a set of clauses and another one to learn one clause from a positive example.

3.9.1 Clint's disjunctive part.

Clint's disjunctive part is obtained by using the following instantiations:

- $L_C = 2^L$, where L is the language employed by Clint
- L_e = the set of ground facts for L
- *Quality-Criterions* (ps) = true iff ps is satisfies all examples and all examples have been supplied
- *Stop-Criterion* (Partial_Solutions, Complete_Solutions) = (| Complete_Solutions | = 1)

- *Select* ({ ps }) = ps,
  each time Select is called, Partial_Solutions is a singleton
- *Optional* = true
- *Generate_Example* (Partial_Solutions, Complete_Solutions) =

  if the knowledge base does not satisfy a known example e
  then return such an example e
  else if there is an example e on postpone such that the
  knowledge base was changed since the previous attempt to cover e
  then return such an example e
  else ask the the user to supply an example
  endif
  endif

- *Initialize* = { $\emptyset$ }
  $\emptyset$ corresponds to the element $\perp$.
- *Oneof*(spec, gen) =

  if ex is positive and uncovered by the knowledge base
  then gen
  endif
  if ex is negative and covered
  then spec
  endif

- *Generalization_Transformer* (ps, ex) = **call** handle_uncovered (see algorithm 3.5 and the next subsection).
- *Specialization_Transformer* (ps, ex) = **call** specialize (ex) (see algorithm 3.6)
- *Remember* = *Adapt* = *Memory* = **void**
- *Filter* ({ps}) = ps
  There is only one current knowledge base at a time.

3.9.2 Clint's conjunctive part

The procedure outlined in algorithm 3.5 can be obtained as an instantiation of Gencol as follows:

- $L_C = L$, where L is the language considered by Clint
- L_e = the set of ground facts for L
- *Quality-Criterions*(ps) = true if ps is maximally general and consistent

- *Stop-Criterion* (Partial_Solutions, Complete_Solutions) =
  (| Complete_Solutions | = 1) or (| Partial_Solutions | = 0)
  In the first case a solution (current) is returned, in the second case the example ex is added to postpone.

- *Select* (Partial_Solutions) =

  if $\exists ps \in$ Partial_Solutions such that ps is a generalization of current
  then select the ps that is the oldest generalization
  of current on Partial_Solutions
  else select any ps on Partial_Solutions
  current := ps
  endif

  The system keeps track of a current candidate clause; this is the clause c in algorithm 3.4. This clause is initialized in the else part of the test. In the else part, it is also the case that any ps on Partial_Solutions belongs to $\mathcal{JUS}(KB, L, p)$. This explains the initialization of current.

  In order to find a clause that generalizes a certain positive example, there are two iterations : one that is used to find a consistent justification (cf. handle_uncovered) and another one that generalizes the consistent justification (cf. generalize_clause).

- *Optional* = true

- *Generate_Example* (Partial_Solutions, Complete_Solutions) =

  if ps covers an example e that is not covered by the clause current
  and *ps* does not cover negative examples
  then return such an example e and ask the user for $\tau(e)$
  else return void
  endif

- *Initialize* = $\mathcal{JUS}(KB, L, p)$
  p is the positive example passed by the generalization operator of the disjunctive learner.

- *Oneof* (spec, gen) = gen

- *Generalization_Transformer* (ps, ex) =

  if ex $\neq$ void
  then if $\tau_2(ex)$ = true
  then return { ps }
  current := ps
  ps covers a positive example
  else return $\emptyset$; *ps covers a negative example*

```
        endif
else    return λ_g ( ps ) ;
        ps is a syntactic variant of current
endif
```

- *Specialization_Transformer* (ps, ex) = **void**
- *Remember* = *Adapt* = *Memory* = **void**
- *Filter* (Partial_Solutions) =

```
if the previous invocation of Generalization_Transformer modified current
then return { current }
else return Partial_Solutions
endif
```

If the system generalizes, it cleans up the candidate generalizations.

3.10 Related work for Clint.

The algorithms of Clint are an extension of the ones presented in [De Raedt and Bruynooghe 88]. Differences with respect to [De Raedt and Bruynooghe 88] are the use of postponing, the use of a more general generalization operator, the use of more expressive languages and a more thorough analysis of the techniques. The algorithms of Clint that were presented in this chapter are related to

- Work on the versionspace method [Mitchell 77] and its extension towards interactive concept-learning [Subramanian and Feigenbaum 86].
- Interactive concept-learners such as Disciple [Kodratoff and Tecuci 87, Tecuci and Kodratoff 90], Marvin [Sammut and Banerji 86] and Alvin [Krawchuk and Witten 88].
- Program synthesis from examples, in particular the work of Ehud Shapiro on the Model Inference System [Shapiro 83].

Clint's basic algorithm is very much related to the versionspace method. The main differences are that Clint stores its negative examples instead of the set of most general descriptions and that one solution will be found by Clint instead of all. This is advantageous because it makes it possible for Clint to use rich description languages. In most languages used by Clint (see chapter 4), there are many most general specializations of a clause with respect to an example but only one most specific generalization. As a result, the G-set of the versionspace method could grow very large. Furthermore, in the learning apprentice context only one concept-description is wanted. So, it is unnecessary to derive all solutions. A further difference with Mitchell's approach is that Clint has no problems

in learning disjunctive concepts. Disjunctive concepts can also be learned by the extensions of the versionspace method discussed by [Murray 87]. However, these extensions are computationally very expensive (they require an exponential amount of computation).

As Clint generates its own examples, it is also related to the extension of versionspaces of [Subramanian and Feigenbaum 86]. Clint does not generate most informative examples as [Subramanian and Feigenbaum 86] but only relevant ones. A similar remark can be made concerning Alvin [Krawchuk and Witten 88], which also generates optimal examples. Therefore Clint's example generation process is more efficient but less powerful. There is however some evidence [Ruff and Dietterich 89] that the benefit of computing the most informative examples instead of relevant ones is marginal as compared to the computational cost.

Clint is also related to Sammut and Banerji's Marvin and Krawchuk and Witten's Alvin. Both systems also search specific to general and generate examples. However, in these systems it is less clear which concepts can be learned (see also section 3.5).

Another system that learns concepts and generates examples is Disciple [Kodratoff and Tecuci 87, Tecuci and Kodratoff 90]. Disciple is more than a concept-learner, it is a learning apprentice that has several learning modes. Nevertheless, the weak theory component resembles the learning procedure used by Clint. The way Clint constructs justifications is related Disciple's way of finding explanations and its analogy based method is similar to Clint's generalization method. However, Disciple interacts with the user to find a justification and it is less clear which concepts can be learned by Disciple because of the use of a reduced versionspace (cf. [Tecuci 88]). The relations of Clint to Disciple will be investigated further in chapters 4 and 9.

An important piece of related work for Clint, as presented in this chapter, is certainly the Model Inference System of [Shapiro 83]. Shapiro's system derives Prolog programs from examples, i.e. ground facts. In order to do so, it uses three procedures:

- A procedure to retract an incorrect clause from the knowledge base. (This procedure is similar to the one presented in algorithm 3.6).
- A procedure to add a new clause covering a positive example. This procedure searches a refinement graph of clauses in a general to specific manner. The refinement graph is derived from user supplied information and may be infinite.
- A procedure to recover from non-terminating programs.

Shapiro proves that his system error-free identifies concepts in the limit under certain conditions. The main differences between Clint and the Model Inference System are:

- Clint uses finite languages, which are derived from the knowledge base (see chapter 4), whereas the Model Inference System uses a possibly infinite language that has to be supplied to the system (the user has to declare which predicates may be used in the definition of each predicate).
- As sketched in section 3.7, the Model Inference System is able to recover from non-terminating programs. However, it requires a special oracle to do so.
- The Model Inference System always realizes error-free identification in the limit, whereas Clint achieves postponed identification in the limit. The reason why the

Model Inference System achieves error-free identification is that it uses a language in which there is always a consistent clause covering a positive example. This is only possible when searching general to specific (c.f. [Buntine 88]).

- The Model Inference System does not query the user with the purpose of generalizing clauses. It only queries the user in order to verify whether a certain clause covers an example in the intended interpretation. The Model Inference System has three different strategies to verify the coverage of an example in the intended interpretation: the lazy, adaptive and eager search strategy.

  - the lazy strategy does not ask questions to verify the coverage of an example w.r.t. a clause: it only uses the input facts.
  - the adaptive search strategy does not ask questions to realize this: it uses the input facts together with the already learned program.
  - the eager search strategy extensively queries the user in order to find a clause covering a certain positive example. Basically, it will for each candidate clause $p \leftarrow q_1 \wedge ... \wedge q_k$ and a positive example $p\theta$ ask whether the $q_i\theta$ are true. This results in a very large number of questions.

  As a result, the lazy and adaptive strategy ask less questions to the user but require the user to supply good training examples. Using Clint many of these examples are automatically derived. Also, when the lazy search strategy is employed, the Model Inference System will not find the clause $p \leftarrow q_1 \wedge ... \wedge q_k$ unless it had already been told that all $q_i\theta$ are true. For the adaptive search strategy, it will not consider the clause unless it can derive that the $q_i\theta$ are true. This derivation takes into account the already learned program together with the user supplied facts. These requirements lay a heavy burden on the user: she has to carefully select the training instances.

- A major advantage of Clint over the Model Inference System is that it is more user friendly: Clint imposes less restrictions on the training set and generates many interesting examples that allow the system to quickly generalize. Also, Clint uses the available knowledge base in order to derive its language and its examples. The learned knowledge base is only used by the Model Inference System in the adaptive search strategy. Clint's languages also evolve as the knowledge base evolves (see also chapter 4). Furthermore, there are several extensions of Clint that are not present in the Model Inference System (see the other chapters in this thesis).

3.11 Summary of the achievements.

In this chapter, the problem of interactive concept-learning for learning apprentices was formalized and extensively studied. We started by presenting two theoretical models of concept-learners: one that learns conjunctive concepts and another one that learns disjunctive ones. Then the issues involved in language bias were revisited and the basic algorithms of Clint were presented. It was shown that Clint is a kind of practical incarnation of the theoretical model $\mathcal{DVS}$. One difference between the abstract model and its incarnation was the interactive nature of Clint. The theoretical properties of Clint

were also investigated and it was shown that Clint identifies concepts in the limit with postponing. Postponed learning appears very natural if the learning process uses what it already knows to learn. It can even be argued that this also occurs in human learning: if there is vital knowledge missing to learn something, the learning of that something will usually be postponed until the necessary knowledge is acquired.

We also investigated the issues concerning perfect query answering mechanisms. It was shown that Clint would be able to synthesize Prolog programs from examples if it were equipped with a perfect query answering mechanism.

Furthermore the factors influencing the complexity of Clint were discussed and it was shown how Clint is an instantiation of Gencol. This provides further evidence for the generality of Gencol.

Chapter 4

Indirect Relevance and Bias.

4.1 Introduction.

Most concept-learning approaches (see e.g. [Mitchell 82, Quinlan 86, Michalski, Carbonell and Mitchell 83, Michalski, Carbonell and Mitchell 86]) require that the user completely specifies each example. Formulated in a logical framework, this means that examples are ground clauses as for instance `father(jeffrey, tom) ← male(jeffrey) ∧ male(tom) ∧ parent (jeffrey, tom)`. These ground clauses specify all the relevant information (`male(jeffrey)`, `male(tom)`, `parent(jeffrey, tom)`) which makes the example (`father(jeffrey, tom)`) an instance of the concept `father`. This property of concept-learning problems is called *direct relevance* by [Michalski 83]. In the previous chapter, Clint was presented as a system that assumes *indirect relevance*. When indirect relevance is assumed, examples are specified as ground facts (e.g. `father (jeffrey, tom)`) and further information concerning the examples (such as `male(jeffrey)`, `male(tom)`, `parent(jeffrey, tom)`) has to be derived from a knowledge base. Assuming direct relevance is a strong requirement for concept-learning. It charges the user with a non-trivial and tedious task: all relevant features must be determined and all examples must be described in terms of these features. This task may be time consuming, expensive and difficult. Direct relevance simplifies the problem of concept-learning as it assumes that an important aspect of the concept-learning task, the selection of the concept-description language, has already been accomplished (cf. [Utgoff and Mitchell 82, Mitchell 80, Utgoff 86, Rendell 86]). In the previous chapter it was argued that certainly in a learning apprentice context, it is not attractive to assume direct relevance. The reason is that the necessary knowledge to learn a concept from examples is in the apprentice's knowledge base. Also, the system should be as user-friendly as possible. So, the system should identify the relevant features from the knowledge base rather than requiring the user to supply them.

Despite the potential impact of indirect relevance, the problem of indirect relevance has not received much attention in the machine learning literature. A few notable exceptions are: Disciple [Kodratoff and Tecuci 87, Tecuci and Kodratoff 90] which proposes to the user features that must be classified as relevant or irrelevant for the concept-learning problem and AQ [Michalski 83] and Marvin [Sammut and Banerji 86] where the user has to supply the system with a set of basic features from which additional features may be deduced using background knowledge. The Plausible Generalization Algorithm [Buntine 87] and Clint (as presented in the previous chapter) are systems which handle indirect relevance within

a fixed predefined language. Here, Clint is extended in order to handle indirect relevance completely. This involves removing the restrictions imposed by a predefined language (see [De Raedt and Bruynooghe 90a, De Raedt and Bruynooghe 89d]). The restrictions are removed by providing a series of languages and the ability to change from one language to another one when it is detected that a language is too restricted to describe a concept. Changing from one language to another is a form of *shifting the bias*.

In the framework of Clint, examples are ground facts. Hence, an example only specifies the concept or predicate, of which it is an example, and the involved arguments. Clint derives for each example, a set of relevant features using the arguments and the knowledge base. A set of relevant features is a *justification* for the example (see chapter 3). The justifications determine the search-space for the concept. The derived justifications depend on the concept description language bias. When the justifications are derived, the user can (but need not) help the system, by providing additional hints. Such hints result in computational advantages, as they reduce the search-space for the concept. The initial justifications may not allow to find a solution for the concept-learning task as the corresponding language bias may be incorrect. Therefore a method is presented that allows to decide how and when to extend the justifications (and the corresponding language). Extending a justification has a similar effect as a shift of bias of the concept-description language [Utgoff and Mitchell 82, Utgoff 86, Grosof and Russell 90, Russell and Grosof 90]. The conditions that guarantee the existence of a correct justification (and concept-description) will be investigated. The resulting method can be applied in general concept-learning systems such as Disciple [Kodratoff and Tecuci 87, Tecuci and Kodratoff 90], Marvin [Sammut and Banerji 86], the version of Clint presented in the previous chapter and versionspaces [Mitchell 77, Mitchell 82, Hirsh 90].

As an application of our technique, it will be shown how to integrate it in versionspaces and in Clint. It is argued that the resulting version of Clint (see also [De Raedt and Bruynooghe 89e]) is quite friendly to the user as the involvement of the user is minimized: in order to learn a concept, the user only has to supply some examples (ground facts) to the system and classify examples that are generated by the system as true or false. Also, the user need not be concerned with the choice of the language bias. This contributes to the applicability of Clint in learning apprentices that operate in a weak theory domain such as Disciple.

This chapter is organized as follows: in section 4.2, the issues involving language-bias are revisited; in section 4.3, a solution for the problem of indirect relevance is presented and in section 4.4, the integration of the technique to cope with indirect relevance in versionspaces and the system Clint is discussed. Finally, in section 4.5, some conclusions are formulated and relevant related work is discussed.

4.2 Language bias revisited.

In chapter 3, several important issues concerning bias were identified (see section 3.4). One important aspect of bias was that – on the one hand – bias is necessary, while – on the other hand – bias can cause problems in finding correct concept-descriptions. Problems can arise because each language, fulfilling the language assumptions 3.1, can only express a certain class of concepts. This also means that for each language, there are concepts that cannot be described in the given language. This is illustrated in example

4.1. The example clearly shows that there are fundamental limitations imposed by each bias although the range of expressible concepts depends on the knowledge base. There is only one exception to these limitations. This exception occurs when $L_e \subseteq L_C$. For such a language, it is always possible to derive a complete and consistent concept-description, in casu the disjunction of the set of positive examples. Therefore it can be argued that such languages do not bias the induction task (cf. also section 3.4).

Example 4.1: Illustrating the limitations of bias (continued)

Reconsider example 3.4. There it was shown that it is impossible to describe the concept `grandparent` in terms of `parent`, `male` and `female` in the language of completely bound Horn-clauses.
Remember also that the range of expressible concepts in a language depends on the used knowledge base. Indeed, if the predicates `grandmother` and `grandfather` would be in the knowledge base, then the predicate `grandparent` could easily be described in the language of completely bound Horn-clauses. One approach to circumvent the problems with a fixed bias would be to learn the predicates such as `grandmother` and `grandfather` first. However, for a given knowledge base and language it is not always possible to do so. Consider e.g. the given knowledge base and language. In order to learn a `grandfather` or `grandmother` predicate one also needs to go outside the language of completely bound Horn-clauses.

The language bias determines which features in the knowledge base will be included in the justifications of examples and by doing so the search space of the concept is determined. The language bias should only include the relevant features in justifications for positive examples (cf. example 4.2). The more features that are included in a justification, the larger the search space for the concept. So, the ideal bias for learning a certain concept is the one that allows to include all relevant features in the justifications and nothing more. As a consequence of these considerations, it is impossible to design a single best description language that is well suited for learning *any* concept. Clearly, the ideal bias depends on the particular concept being learned.

If it is impossible to devise an ideal concept-description language, why not use a series of concept-description languages ? This is precisely what shall be done in the rest of this chapter. Several series of concept-description languages will be defined. The languages in the series will be ordered according to growing expressiveness. This will allow to change the bias, i.e. to *shift* the bias, when it is discovered that the given language is too restricted. At that moment, a more expressive language will be considered. In this way, the bias will be very restricted at the start, and relaxed whenever necessary, as in [Utgoff 86, Grosof and Russell 90]. [Grosof and Russell 90] show that working with different language biases is a form of default reasoning: one starts with a very restricted language by default, and only if there are indications that the bias is incorrect, another language is considered. Later, if this language also proves to be incorrect, a third language is considered, ... [Grosof and Russell 90] and [Russell and Grosof 90] spend much effort in

Example 4.2: Relevance of justifications.

Reconsider example 3.3. Imagine that there is also information concerning numbers in the knowledge base such as definitions for the predicates `less` and `equal`, ... When the justifications for a positive example such as `grandparent(rose, luke)` are computed, it is desirable to include only the relevant family features in the justifications such as `parent(rose, mary)`, `parent(mary, luke)`, `female(rose)`, `male(luke)` and `female(mary)`. This information is needed in justifications to be able to describe the concept. Yet it is desirable to keep the size of the justifications as small as possible. Indeed, imagine that also the facts `less(6,8)` and `equal(10/2, 5)` would be included in a justification. This would only result in a larger search space.

showing how language biases can be represented in a declarative way. However, they do not show how to find suitable language biases. In the next section, definitions of abstract language biases will be used together with examples and knowledge bases to compute interesting declarative biases in the sense of Grosof and Russell.

4.3 A solution for indirect relevance.

In this section, a technique is proposed to solve the problem of indirect relevance. In chapter 3, it was shown how concepts could be learned given a language (cf. $\mathcal{VS}$, $\mathcal{DVS}$ and Clint), examples and a knowledge base. In section 4.3.1, different series of abstract concept-description languages are presented. The language series are ordered according to growing expressiveness. The languages can be used by the concept-learners $\mathcal{VS}$, $\mathcal{DVS}$ and Clint, presented in chapter 3. In section 4.3.2, it is shown how and when the language bias should be shifted. We also indicate how the user can help the learner to reduce its search space.

4.3.1 Abstract languages.

Defining series of languages.

For Horn-clause logic, some useful series of abstract language biases are listed below. The languages are *abstract* in the sense that they are defined independently of any knowledge base. Given a knowledge base, the abstract languages become specific by filling in the predicate names that are in the knowledge base. The list of languages, given below, serves only as an illustration and is not meant to be exhaustive. For some applications, it may be desirable to define other languages. Nevertheless, the series defined below, clarify which operations can be used to obtain a series of languages. At first, Horn-clauses without functors and constants will be assumed. Later, it will be shown how the given languages can be extended to deal with functors and constants.

Definition 4.1 *Series 1*

$$\forall i \geq 0 : L_i = \{c \mid \quad (1) \quad head(c) = p(x_1, ..., x_n) \text{ and } body(c) \subseteq B_i(x_1, ..., x_n) \text{ and}$$

(2) *c is linked and range restricted and*
(3) *all x_r are different variables and*
(4) *all variables $y_s \in vars(body(c)) - vars(head(c))$ have only one occurrence in c* }

where

$$B_i(x_1,...,x_n) = \{q(y_1,...,y_k) \mid q \text{ is a predicate occurring in the knowledge base and } |\{y_1,...,y_k\} - \{x_1,...,x_n\}| \leq i \text{ and } \{y_1,...,y_k\} \cap \{x_1,...,x_n\} \neq \emptyset\}$$

The notation $|S|$ denotes the cardinality of the set S. What does this definition mean ? First, after filling in $i = 0$, one obtains a language whose clauses contain literals of the form $q(y_1,...,y_k)$ with the requirements that $|\{y_1,...,y_k\} - \{x_1,...,x_n\}| \leq 0$ and $\{y_1,...,y_k\} \cap \{x_1,...,x_n\} \neq \emptyset$. This is equivalent to $\{y_1,...,y_k\} \subseteq \{x_1,...,x_n\}$, which implies that there are no variables y_s allowed that are not equal to an x_r. So, L_0 is the language of completely bound Horn-Clauses defined in example 3.3. Secondly, after filling in $i = 1$, one obtains a language where the literals in the body of a clause c are allowed to contain at most $i = 1$ variable y_s that do not occur in *head*(c). The variables occurring in the head are the x_r. Furthermore, these variables y_s that are different from the x_r, i.e. the local ones, can occur at most once in a clause. Therefore i is called the *variable degree* of the clause.

Definition 4.2 *The variable degree of a clause c, i(c) is:*

$$i(c) = max\{\ local(l) \mid l \text{ is a literal in } body(c)\}$$

where local(l) is the number of local variables in l, i.e. the number of variables that occur in body(l) and not in head(c).

Notice also that the series is *abstract* as it makes abstraction of the predicates that occur in the knowledge base and the predicate being learned. It is only by filling in these predicate names q and the arity n and name of the predicate p being learned that one obtains specific language biases. An illustration of how to obtain such specific language biases is given in example 4.3.

To define series 2, the *level of existential quantification* [1] is introduced.

Definition 4.3 *$k(v)$, the level of existential quantification of a variable v in a clause c is:*
$k(v)$=0 if v occurs in head(c)
$k(v) = min\ \{\ k(y) + 1 \mid v \neq y \wedge v,y$ are variables that occur in a literal l of c $\}$ otherwise

$k(c)$, the level of existential quantification of a clause c:

$$k(c) = max\{k(x) \mid x \text{ is a variable } \in c\}$$

[1] Although all variables are universally quantified over the whole clause, one can consider the *local* variables, i.e. the ones occurring in the body but not in the head, as existentially quantified over the body because it is possible to rewrite clauses into a form where these local variables are existentially quantified. Hence, we will sometimes use the term existentially quantified variables instead of local variables.

Example 4.3: Illustrating language series 1.

Assume the knowledge base of example 3.3 is used and that the predicate `father(X,Y)` has to be learned. Using series 1, the following languages are obtained:

$$L_0 = \{c = \texttt{father(X,Y)} \leftarrow \mathit{body} \mid \mathit{body} \subseteq B_0\texttt{(X,Y)} \text{ such that } c \text{ is linked and range-restricted}\}$$

where

$$B_0\texttt{(X,Y)} = \{\texttt{male(X), male(Y), female(X), female(Y),}$$
$$\texttt{parent(X,Y), parent(Y,X), parent(X,X), parent(Y,Y)}\}$$

$$L_1 = \{c = \texttt{father(X,Y)} \leftarrow \mathit{body} \mid \mathit{body} \subseteq B_1\texttt{(X,Y)} \text{ such that } c \text{ is linked and range-restricted}\}$$

where

$$B_1\texttt{(X,Y)} = B_0\texttt{(X,Y)} \cup$$
$$\{\texttt{parent(X,Z1), parent(Y,Z2), parent(Z3,X), parent(Z4,Y)}\}$$

For all $i > 1$, $L_i = L_1$ because the maximum arity of the predicates in the knowledge base is 2. The reason is that for series 1, each literal must contain a variable occurring in the head. This implies that series 1 is always finite. However, it will be shown how the elements of series 1 are used in the definitions of series 2 and 3 to obtain infinite series.

The variable `Y` in the definition of `father` as `father(X) ← parent(X, Y) ∧ male(X)` has 1 as a level of existential quantification. The variable `Z` in the definition of `grandfather` as `grandfather(X) ← parent(X, Y) ∧ parent(Y, Z) ∧ male(X)` has 2 as a level of existential quantification. Analogously, a clause for `greatgrandfather` would have a level of 3, ...

Definition 4.4 *Series 2*

$$\forall i, k \geq 1 : L_i^k = \{c \mid (1)\ head(c) = p(x_1, ..., x_n) \text{ and } body(c) \subseteq B_i^k(x_1, ..., x_n) \text{ and } (2)\ c \text{ is linked and range restricted and } (3)\ \text{all } x_r \text{ are different variables and } (4)\ \text{all variables in } c \text{ that have } k \text{ as level of existential quantification have only one occurrence in } c\}$$

where

$$\text{for } k = 1 : B_i^1(x_1, ..., x_n) = B_i(x_1, ..., x_n)$$

$$\text{for } k > 1 : B_i^k(x_1, ..., x_n) = \{q(y_1, ..., y_m) \in B_i(z_1, ..., z_l) \mid q \text{ is a predicate occurring in the knowledge base and } \{z_1, ..., z_l\} \text{ is the set of all variables occurring in } B_i^{k-1}(x_1, ..., x_n)\}$$

Series 2 is illustrated in example 4.4. Using definition 4.4, many interesting series of language biases can be derived, e.g. $L_1^1, L_2^1, L_2^2, ...$ or $L_1^1, L_1^2, L_1^3, ...$ Although the first series specifies a means to introduce new existentially quantified variables in clauses, the first series does not allow relations among these existentially quantified variables (the `Zi` in example 4.3). Series 2 specifies how relations among existentially quantified variables, defined at a deeper level, can be introduced. The following two operations: introducing new existentially quantified variables and introducing relations between them, are the basic operations needed to define series of language biases for Horn-clauses.

Definition 4.5 *Series 3*

$$\forall i_1, ..., i_n \geq 0 : L_{i_1,...,i_n} = \{c \mid (1)\ head(c) = p(x_1, ..., x_k) \text{and } (2)\ body(c) \subseteq B_{i_1,...,i_n}(x_1, ..., x_k) \text{ and } (3)\ c \text{ is linked and range restricted}\}$$

where

$$\forall i_1, ..., i_n \geq 0 : B_{i_1,...,i_n}(x_1, ..., x_k) = \{q(y_1, ..., y_m) \in B_{i_n}(z_1, ..., z_l) \mid q \text{ is a predicate occurring in the knowledge base and } \{z_1, ..., z_l\} \text{ are all the variables occurring in } B_{i_1,...,i_{n-1}}(x_1, ..., x_k)\}$$

As before, all x_r, y_s, z_t are variables. The series is initialized with L_{i_1}. The series $L_0, L_{0,1}, L_{0,1,0}, ...$ will be illustrated in example 4.6.

Example 4.4: Illustrating language series 2

Suppose the knowledge base of example 3.3 is used and the aim is to learn a predicate p(X). Then the languages are: $L_1^1 = L_1$ and L_1^2 is based on $B_1^2(X)$.

B_1^1 (X) = B_1 (X) = { male(X), female(X), parent(X,Y), parent(W,X) }

$B_1^2(X) = \{q(y_1,...,y_m) \mid q(y_1,...,y_m) \in B_1(X, Y, W)\} =$
{ male(X), male(Y), male(W), female(X), female(Y), female(W),
parent(X,Y), parent(Y,X), parent(X,W), parent(W,X), parent(Y,W),
parent(W,Y), parent(X,Z1), parent(Z2,X), parent(Y,Z3), parent(Z4,Y),
parent(W,Z5), parent(Z6,W) }

So, in language L_1^2 the following clause can be expressed:

grandfather(X) ← parent(X, Y) ∧ parent(Y, Z3) ∧ male(X)

Properties of the languages.

The languages defined in the previous subsection satisfy the languages assumptions 3.1 of chapter 3 by definition.

Lemma 4.1 *All languages L belonging to series 1, 2 or 3 w.r.t. all knowledge bases satisfy the language assumptions 3.1.*

The lemma implies that the set of justifications for a positive example p, $\mathcal{JUS}(KB, L, p)$ is always finite. This property allows to detect when the language bias is too restricted (cf. subsection 4.3.2). Note that the bias can be too restricted as the language of examples is not a subset of the language of concepts. The property also implies that each language can only express a certain class of concepts. Indeed, for each language, one can formulate a concept-description which is outside the language. The language-series split an infinite language (a subset of Prolog) into an infinite number of finite sublanguages.

Observe also that as stated in chapter 3, the operator $\mathcal{L}_g$ (see definition 3.33) is sufficient to find all generalizations of clauses in the languages of series 1, 2 and 3. The reason is the condition *body* $\subseteq B$ in the definitions of the languages.

An interesting theoretical question w.r.t. the language definitions, is whether the disjunctive or conjunctive nature of a concept is preserved when shifting from one language to another in the given series. First, if a concept is conjunctive w.r.t. a language L_i, it will also be conjunctive w.r.t. languages L_j with $j \geq i$. The reason is that the considered language series have the following property:

Lemma 4.2 $\forall i : L_i \subseteq L_{i+1}$. *This holds for series 1, 2 and 3 defined above.*

However, if a concept is disjunctive in language L_i then it may be conjunctive in language L_{i+1}. This is illustrated in example 4.5.

Another question is how to extend the language definitions when functors and constants are allowed. If functors and constants are to be used, then the concept-learning

Example 4.5: The disjunctive nature of a concept is not preserved.

Suppose the knowledge base contains the following facts:

```
aa(a).  bb(b).  r(a).  r(b).  r(c).  p(a,d).  p(b,d).
```

Suppose there are two positive examples: `q(a)` and `q(b)`, and one negative example `q(c)`. Then the predicate `q` is disjunctive in L_0: `q(X)` $\leftarrow$ `aa(X)` and `q(X)` $\leftarrow$ `bb(X)` but conjunctive in L_1: `q(X)` $\leftarrow$ `p(X,Y)`.

task becomes one of synthesising programs. This – in general – requires an oracle for the halting problem (cf. [Biermann 86, Shapiro 83] and chapter 3). This is the underlying reason why functors and constants have not been considered in the above defined languages. Nevertheless, we indicate how the given series can be modified to cope with functors and constants. There are at least two possibilities to treat functors and constants.

- A first approach does not modify the languages. Indeed, [Rouveirol and Puget 89] show that any clause can be transformed into an equivalent clause without functors and constants if certain facts are added to the knowledge base. To transform a clause, one has to introduce for each functor f/n a predicate $F/n+1$ that is defined by the fact $F(x_1,...,x_n,f(x_1,...,x_n)) \leftarrow$ where the x_i are different variables, and for each constant c, a predicate $C/1$ defined by the fact $C(c) \leftarrow$. The clause `length([X], 1)` $\leftarrow$ can be rewritten as `length(Y,Z)` $\leftarrow$ `list(X,W,Y)` $\wedge$ `nil(W)` $\wedge$ `one(Z)` if the facts `list(X,Y,[X|Y])` $\leftarrow$, `nil([])` $\leftarrow$ and `one(1)` $\leftarrow$ are added to the knowledge base.

  As a consequence, if the necessary predicate definitions for functors and constants are added to the knowledge base, the language series specify logic programs.

- Secondly, it is also possible (and probably more feasible) to define a third basic operation and use it to redefine the language series. This third basic operation, allows the instantiation of a variable by a term $f(x_1,...,x_n)$ where f/n is a functor and the x_i are variables not occurring anywhere else.

There is also the important issue whether the series of languages defined above are good ones, and if so whether they are to be used under all circumstances. In this respect, it is important to see that the series of languages used in a system, *biases* that system. Therefore, the suitability of a language series depends on the application. The language bias together with the knowledge base and its structure influences the search space of the concept-learning process. As a consequence, the structure of the knowledge base determines the suitability of a language series. As an illustration, consider the family-domain and imagine that there are many exotic predicates like `grandfather_in_law`, `uncle_of_my_father`, `nephew_in_the_third_degree` in the knowledge base. Then each person in the domain will have many relations (w.r.t. various predicates) to other persons. If the considered language allows for many existentially quantified variables, one

will obtain large justifications, containing many literals. Clearly, this will slow down the learning process. One might want to try other language biases, specifying different restrictions, as e.g. clauses in L_i allow for at most i occurrences of an existentially quantified variable. Although this solves the problems in our family domain example, we believe (and our experience with Clint shows) that this is not always a good idea. The reason is that by doing so, there is a potential danger of obtaining a smaller but more complicated search space. Let us illustrate this on the family domain. Assume that the versionspace method of [Mitchell 77, Mitchell 82] is being used. Given the above exotic situation, and the considered language series, one would usually obtain a small S-set. Initially the S-set is a singleton; the clause containing all literals expressible in the language is its single member. This clause is of the form $p(x_1,...,x_n) \leftarrow B_i(x_1,...,x_n)$ for the language L_i. However, if the number of occurrences of existentially quantified variables would be used as a criterion, there would be many elements in the initial S-set, as there would be one most specific clause for each literal that is allowed to contain an existentially quantified variable. Let me illustrate this point on a small example. Assume the knowledge base contains only the predicate p of arity 2 and that the aim is to learn the predicate q of arity 1. Then the S-set w.r.t. language L_1 is S={q(X) ← p(X,Y) ∧ p(W,X)}. If the number of occurrences of existentially quantified variables is limited to 1 and the number of such variables to 1, S={ q(X) ← p(X,Y) ; q(X) ← p(Y,X) }. For n predicates of arity 2 in the knowledge base, the S-set for this example would contain $2 \times n$ clauses. This is a general phenomenon when limiting the number of existentially quantified variables. So, it appears that, although the given language series may not always be optimal, there are good reasons for using them. A second point to make is that the basic operations, which are necessary to define different language series, have been defined. These basic operations do not change from one application to another. It is only the way in which they are applied that might be modified.

4.3.2 When and how to shift the bias ?

By now, it is easy to characterize an incorrect language bias. For the moment, we will assume that the used knowledge base is definitely correct. This assumption allows us to decide whether the reason for an anomaly (cf. chapter 3) is due to an incorrect bias or to an incorrect or incomplete knowledge base.

A bias L is considered to be incorrect for learning a conjunctive predicate from examples P and N, whenever $\mathcal{VS}(KB,L,P,N)=\emptyset$. A bias L is considered to be incorrect for learning a disjunctive predicate with examples P and N, whenever $\mathcal{DVS}(KB,L,P,N)=\emptyset$. Notice that the conditions $\mathcal{VS}(KB,L,P,N)=\emptyset$ and $\mathcal{DVS}(KB,L,P,N)=\emptyset$ arise in Clint when there is an anomaly! In the context of Clint, there were two possible causes or an anomaly: either the language bias is incorrect or the knowledge base does not contain enough knowledge. In the previous chapter, the second cause was remedied by postponing the learning of certain examples. Here, we will remedy the first cause. These issues will be reconsidered while incorporating the technique to shift the bias in Clint in section 4.4. Until then we will assume that if an anomaly arises, or – equivalently – if the versionspace becomes empty, this is due to the language bias.

When the language bias is incorrect, it can be shifted. A high-level algorithm to learn concepts and shift the bias is shown in algorithm 4.1. The algorithm assumes a (possibly infinite) series of languages $L_1,...,L_n,...$ to describe the concept. Whenever the

algorithm discovers that the current language does not allow to describe a concept, the next language in the series is considered.

Algorithm 4.1: Two-layered concept-learning.

```
procedure two layered concept-learning
    i := 1
    P := ∅
    N := ∅
    while there is no solution do
        Select an example e
        Add e to P or N
        Compute or adjust the versionspace ((D)VS(KB, L_i, P, N))
        if the versionspace is empty ((D)VS(KB, L_i, P, N) = ∅)
        then i :=i+1
        endif
    endwhile
endproc
```

Note that algorithm 4.1 does not require that the (inefficient) algorithms of $\mathcal{VS}$ or $\mathcal{DVS}$ are used. Indeed, any sound concept-learner that computes elements of $\mathcal{VS}$ or $\mathcal{DVS}$ may be used. Sound concept-learners consider all concept-descriptions expressible in the concept description language. Their aim is to find one or more complete and consistent concept-descriptions. They are able to detect the conditions $\mathcal{VS}(KB, L, P, N) = \emptyset$ and/or $\mathcal{DVS}(KB, L, P, N) = \emptyset$. By using $\mathcal{VS}$ and $\mathcal{DVS}$ in algorithm 4.1, abstraction is made of specific concept-learners and the technique is explained in general. Examples of sound concept-learners are the candidate elimination approach of [Mitchell 77, Mitchell 82, Hirsh 90], Clint (see the previous chapter), The Model Inference System [Shapiro 83] and learning by enumeration [Gold 67, Biermann 86, Angluin and Smith 83].

During the learning process for L_i it is not necessary to consider the elements of $L_i \cap \{\cup_{j<i} L_j\}$. This knowledge may be used to prune the search.

There is only one problem with algorithm 4.1: if there is no correct bias in the series $L_1, ..., L_n, ...$, then it will keep on shifting the bias and enter an infinite loop. Such infinite loops indicate that there is vital information, needed for learning the concept, missing. This may be suggested to the user after language L_N is attempted for some given N.

The following theorem holds for algorithm 4.1.

Theorem 4.1 *(from [De Raedt and Bruynooghe 90a])*
Algorithm 4.1 error-free identifies (resp. with postponing) any concept c that is expressible in one of the languages $L_1, ..., L_n, ...$ and the knowledge base KB in the limit if a sound concept-learner A is used such that A error-free identifies (resp. with postponing) any concept d expressible in L_j when A's language is L_j and A's knowledge base is KB $\forall j = 1, ..., n, ...$

Proof : If the bias under consideration is incorrect, a sound concept-learner will eventually detect that the versionspace becomes empty and then the two-layered learning

procedure will consider the next language. Therefore, we have that if any of the biases in the series is correct, the two-layered learning procedure will eventually find a correct bias (if it exists). Given the correct bias, a sound concept-learner will identify the concept. □

Algorithm 4.1 is related to the technique of [Grosof and Russell 90] to shift the bias. They declaratively represent language biases as defaults. If there is evidence indicating that a certain default is incorrect (the versionspace corresponding to the language bias is empty), another default is tried. In their approach, it is also possible to specify preferences for certain biases. Algorithm 4.1 could be rephrased in terms of the one of Grosof and Russell by representing the languages $L_1, ..., L_n, ...$ as defaults and specifying that L_i is to be preferred over L_{i+1}.

A reformulated condition for simplified disjunctive concept-learners

As stated in section 3.3, some concept-learners such as Clint use the simplified model of $\mathcal{DVS}$ to learn disjunctive concepts. Those concept-learners employ rule (4) of theorem 3.2 instead of rule (2). For incremental concept-learners of this type, the conditions to shift the language bias can be rephrased as follows:

Lemma 4.3 *The reformulated condition.*

For incremental concept-learners using the simplified model of $\mathcal{DVS}$ the condition $\mathcal{DVS}(KB, L, P, N) \neq \emptyset \wedge \mathcal{DVS}(KB, L, P \cup \{p\}, N) = \emptyset$ is equivalent to $\forall KB' = KB \cup D$ where $D \in \mathcal{DVS}(KB, L, P, N) : \forall c \in \mathcal{JUS}(KB', L, p)$: c covers a negative example belonging to N.

This result is stated without proof as it is trivial. Lemma 4.3 can be used to test more efficiently whether the disjunctive versionspace is empty. The new test is more operational as only the elements $c \in \mathcal{JUS}(KB', L, p)$ must be considered for all possible KB'. When KB' is unique (this is the case when the system works with a current knowledge base or when the concept being learned may not be recursive since we then can take KB' to be KB), this test can be used to realize a more efficient search strategy. If the disjunctive versionspace becomes empty in algorithm 4.1, then all already computed results are thrown away as a new disjunctive versionspace for the next language is computed. In the situation considered by the lemma, the reason for the versionspace becoming empty is that there are no consistent justifications for the new positive example. This means that using the normal approach, much work is wasted as the correct parts of the versionspace are thrown away together with the incorrect parts. Using lemma 4.3, a different strategy can be followed: First select a solution S_1 to the old problem $\mathcal{DVS}(KB, L_i, P, N)$; then compute a solution S_2 to the problem $\mathcal{DVS}(KB \cup S_1, L_{i+1}, \{p\}, N)$. A solution to the original problem is then a disjunction of two parts $S_1 \vee S_2$. If $\forall i : L_i \subseteq L_{i+1}$ then this strategy will always yield a correct solution. However, not all complete and consistent solutions in L_{i+1} will be obtained. If one is only interested in one solution as in Clint, this forms no problem.

4.3.3 Using knowledge to reduce the search-space.

The presented language definitions are purely syntactical and may therefore result in search-spaces that are larger than necessary. The search-space can be pruned by using knowledge. This section indicates how this can be realized.

In the literature [Subramanian and Genesereth 87, Buntine 87, Russell and Grosof 90, Subramanian 90], there are some approaches that augment the learning process with knowledge about the concept-description languages. [Russell and Grosof 90] use determinations, which can be used to declaratively represent concept-description languages. [Buntine 87] introduces the notion of irrelevance constraints, which specify that certain descriptions should not be taken into account during the learning process. For example, when learning Prolog programs the constraint that the body of clauses should contain only lists of a smaller size than those in the head of the clause, can be used. [Subramanian and Genesereth 87] consider different logical definitions of irrelevance and show how they can be used to reduce the search-space of concept-learning systems. Roughly speaking, they specify irrelevance facts of the form $I(f,g,M)$, which means that the fact f is irrelevant to the fact g in theory M.

These forms of knowledge can also be used in our approach. Consider for instance the irrelevance claims of [Subramanian and Genesereth 87]. If it were known that $I(q(x_1,...,x_n),p(y_1,...,y_m),M)$ is true, then the predicate q may be deleted from the versionspaces (and the languages) used for learning p.

Although the mentioned approaches also specify properties about the language bias and versionspace to be used by the learner, our approach is complementary to the other ones. The reason is that they declaratively specify semantic constraints (knowledge) while we specify syntactic constraints. Syntactic constraints are also necessary (cfr. example 4.2 and section 3.4) to reduce the search-space when there is not enough knowledge available. The mentioned approaches address how to introduce knowledge but they do not clearly say how this knowledge can be acquired. It is usually assumed that the user has to provide this knowledge. In the next section, it is shown how the user can assist the concept-learner to reduce the search-space by answering questions about the (ir)relevance of certain relations in the justification of examples.

Reducing the search-space by answering questions.

In section 3.4, the set of justifications of an example p w.r.t. a language L, $\mathcal{JUS}(KB,L,p)$, was introduced. The elements of $\mathcal{JUS}(KB,L,p)$ were called justifications because they indicate why p belongs to the concept. Justifications can be used to formulate interesting questions about the (ir)relevance of certain relations in a justification. In example 4.6, some justifications of the example `same_color(t2,b4)` are shown.

The justifications may be used to reduce the search-space. The approach to reduce the search-space is based on a technique, applied in Disciple [Kodratoff and Tecuci 87, Tecuci and Kodratoff 90, Tecuci 88]. Justifications may contain irrelevant relations. Such relations increase the search-space. Hence, they slow down the learning process. In practice this often implies that more examples are needed before a solution is found. In many situations, the user can identify some of these unwanted relations. This allows the system to optimize the search. Suppose there is a justification (a clause) c for an example and it is known to contain an irrelevant relation s occurring as a condition in c. Irrelevant conditions may be discarded from the justification, which reduces the search-space. The

Example 4.6: Some justifications

Suppose the knowledge base consists of predicate definitions for `block`, `table`, `object`, `color`, `weight`, `composition`, `=`, `less`, ... Part of the knowledge base is:

```
place(p1) ←              composition(p1,concrete) ←    color(p1,white) ←
place(p2) ←              composition(p2,sand) ←        color(p2,blue) ←

table(t1) ←              composition(t1,metal) ←
table(t2) ←              composition(t2,wood) ←

block(b1) ←              composition(b1,wood) ←
block(b2) ←              composition(b2,plastic) ←
block(b3) ←              composition(b3,metal) ←
block(b4) ←              composition(b4,stone) ←
block(b5) ←              composition(b5,glass) ←

color(t1,red) ←          weight(t1,50) ←
color(t2,grey) ←         weight(t2,50) ←

color(b1,red) ←          weight(b1,20) ←
color(b2,blue) ←         weight(b2,40) ←
color(b3,red) ←          weight(b3,30) ←
color(b4,grey) ←         weight(b4,120) ←
color(b5,white) ←        weight(b5,40) ←

object(X) ← table(X)     object(X) ← block(X)
```

$\mathcal{JUS}(KB, L_{0,}$ `same_color(t2,b4)` $) =$

$\{$`same_color(X,Y) ← block(Y) ∧ table(X) ∧ object(X) ∧ object(Y)`$\}$

$\mathcal{JUS}(KB, L_{0,1},$ `same_color(t2,b4)`$) =$

$\{$`same_color(X,Y) ← block(Y)∧ table(X)∧ object(X) ∧ object(Y) ∧`
`weight(X,XW) ∧ weight(Y,YW) ∧ color(X,XC) ∧ color(Y,YC) ∧`
`composition(X,CX) ∧ composition(Y,CY)`$\}$

$\mathcal{JUS}(KB, L_{0,1,0},$ `same_color(t2,b4)`$) =$

$\{$`same_color(X,Y) ← block(Y)∧ table(X)∧ object(X)∧ object(Y)∧`
`weight(X,XW) ∧ weight(Y,YW) ∧ color(X,XC) ∧ color(Y,YC) ∧`
`composition(X,CX) ∧ composition(Y,CY) ∧ eq(XC,YC) ∧`
`less(XW,YW)`$\}$

All conditions in the body of a justification are true for a certain ground substitution θ which includes `X=t2` and `Y=b4`. These properties of the example together justify `same_color(t2,b4)`.

structure of the language may be used to prune the search further. For example, if it is known that for the example `same_color(t2,b4)`, the relations `weight(X,XW)` and `weight(Y,YW)` are irrelevant then one can infer that `less(XW,YW)` is also irrelevant in $L_{0,1,0}$. Keeping `less(XW,YW)` in the justification would make the justification unlinked. For this example, the irrelevance information reduces the search-space considered by Clint significantly. Instead of requiring 8 examples in order to converge to a solution, Clint needs only 5 of them.

Information about the (ir)relevance of a certain relation in a justification, can be obtained by querying the user. This can be realized by presenting the relations in a justification to the user and asking the user to classify them as relevant or irrelevant. In many situations, it is not only interesting to present these relations to the user but also the substitutions which are true in the example. So instead of merely presenting `color(X,XC)` one can present the instantiation `color(t2,grey)`.

The irrelevance information obtained in this manner is very much related to the information discussed by [Subramanian and Genesereth 87, Subramanian 90].

A second, related form of obtaining irrelevance information from the user queries for the (ir)relevance of certain arguments. This is illustrated in example 4.7.

Example 4.7: Reducing the search space

Assume the application domain is family relations and the aim is to learn a definition for the predicate `grandfather`. Assume also that there are many family-relations as `uncle`, `aunt`, `niece`, `nephew`, and `mother_in_law`, in the knowledge base and assume there is no known `ancestor` or `grandparent` relation. Then the predicate `grandfather` should be expressed as `grandfather(X,Y)` $\leftarrow$ `father(X,Z)` $\wedge$ `parent(Z,Y)` in e.g. $L_{0,1,0}$. Without (ir)relevance information one would obtain a large versionspace when moving to $L_{0,1}$ for the example `grandfather(tom, jeffrey)` because there are many relations that could be included in justifications for the example. This follows from the fact that there are many relations from `tom` and `jeffrey` to other persons, which results in many candidate literals containing existentially quantified variables, e.g. `uncle(T,P)` because of `uncle(tom, paul)` and `niece(M,J)` because of `niece(mary, jeffrey)`, ... In order to reduce the number of these relations to be considered, the system may query the user for the *relevant arguments*. The relevant arguments are those arguments that occur at the first level in a correct proof for the predicate being learned; arguments that occur in the example itself such as `tom` and `jeffrey` are always relevant. In the `grandfather` example, the relevant arguments are the arguments bound to `X`, `Y` and `Z` in the proof of `grandfather`. Given the example `grandfather(tom, jeffrey)` it would be `tom`, `jeffrey` and the `parent` of `jeffrey` which is the `child` of `tom`. Suppose this is `kate`. Using this information the system can reduce its search-space by deleting all relations from the justification which do not have a true instance containing only relevant arguments (`jeffrey`, `tom`, `kate`). In the `grandfather` example, the relations `niece(M,J)` and `uncle(T,P)` would be omitted from the justifications and the versionspaces.

4.3.4 Dynamically computing the next language bias.

As indicated in section 4.3.1, the choice of the language-series biases the system. Therefore it may be useful to compute the next language-bias in the series dynamically rather than determining it statically. It is easy to devise an algorithm to realize such dynamic choices. The next language has to be determined each time that the versionspace becomes empty in algorithm 4.1. The next language can be determined using algorithm 4.2.

Algorithm 4.2: Choosing the next bias dynamically

```
procedure find_bias(L: language, p : positive example) returns language
    Q := successors(L)
    repeat
        if Q contains a language L' such that (D)VS(KB, L, {p}, N) ≠∅
        then return the best of these languages L'
        else delete the best language L'' from Q
             Q :=Q ∪ successors(L'')
        endif
    until a language L' has been returned
endproc
```

Imagine that in such a modified algorithm there are different languages to consider as the next language bias. The space of language biases can be searched in a best-first manner by using algorithm 4.2. Algorithm 4.2 keeps track of a list Q of candidate-biases. Q is initialized to the successors of the incorrect language bias. If the list contains a language that solves the current problem, the algorithm will return such a language. Otherwise it will delete the best bias from the list and compute its successor(s). The best bias is the one with the smallest versionspace.

4.4 Applying the method in practice.

This section describes how our method to shift the bias can be integrated in existing concept-learners. Two concept-learners are considered: the versionspace approach of [Mitchell 77, Mitchell 82, Hirsh 90] and the system Clint as presented in the previous chapter.

4.4.1 Versionspaces.

The generality of our method is demonstrated by showing how to integrate it in versionspaces [Mitchell 77, Mitchell 82, Hirsh 90]. The framework of versionspaces was chosen because it forms a general, well-known formal model of concept-learning (cf. also chapter 2). In order to apply our technique to versionspaces one has to:

- initialize the S-set of the versionspace with the set of justifications for the first positive example p in the first language L_1; $S = \mathcal{JUS}(KB, L_1, p)$.

- initialize the G-set of the versionspace with the following set: $G = \{c \in L_1 \mid c$ is maximally general and c is more general than an element in $\mathcal{JUS}(KB, L_1, p)\}$
- Updating the versionspace w.r.t. other examples can be done using any of the methods described by [Mitchell 77, Subramanian and Feigenbaum 86, Hirsh 90].
- Algorithm 4.1 is then used in a straightforward manner to detect when the bias is incorrect and to shift it when necessary. Of course, if the bias has to be shifted, the versionspace for L_{i+1} has to be initialized. The initialization of the versionspace for L_{i+1} is done as for L_1.

4.4.2 Integrating the method in Clint.

Integrating the method to shift the bias in Clint is rather easy. The only difficulty lies in the closed-loop nature of Clint. As indicated in section 3.6, closed-loop learners are able to learn a set of concepts. Because the learnability of one concept may depend on the presence of other concepts and on the order in which the concepts are learned (see example 4.2 and section 3.6), there are two possible causes for an empty versionspace or anomaly in closed-loop learners:

- The language bias may be insufficient for learning the concept from the starting knowledge base. This means that the predicate cannot be expressed in the language, even when additional predicates would be learned. Such a situation is sketched in example 4.1.
- The knowledge base does not yet contain enough knowledge to describe the concept. Situations of this kind can be handled by postponing the learning of certain clauses for the concept (see also section 3.6, where it was shown how Clint postpones the treatment of certain positive examples).

In the integration of Clint and the method to shift the bias, two possible actions can be performed when an anomaly arises (the versionspace $\mathcal{VS}(KB, L_i, \{p\}, N)$ or $\mathcal{DVS}(KB, L_i, \{p\}, N)$ is empty): the problem can be solved by postponing the learning of p or by shifting the bias to L_{i+1}. Generally speaking, it seems more desirable to shift the bias to a more expressive language than to postpone the treatment of certain examples. The reason is that shifting the bias may give rise to concept-descriptions that satisfy all known examples, whereas postponing does not. However, if the bias shift is preferred, then one should not use an infinite series of languages. Such a series of languages may cause severe problems if the anomaly would be due to an incomplete knowledge base. In that case, no language in the series would be correct for learning the predicate and the system would go on shifting its bias forever. From a practical point of view, this forms – however – no problem as infinite series contain arbitrary large Horn-clauses and it is neither desirable to add such clauses to the knowledge base nor to attempt to learn them as this requires too much computation. Therefore we argue that only clauses of a certain complexity, i.e. clauses belonging to a finite series of languages $L_1, ..., L_M$, should be considered. Given a finite series, the above sketched difficulty can be solved as follows:

- If the versionspace for L_i, with $i < M$ is empty, then shift the bias

- If the versionspace for $\forall i \leq M : L_i$ is empty, then add the responsible positive example(s) to the list postpone.

This solution gives rise to the modification of the basic algorithm of Clint shown in algorithm 4.3. It is only necessary to modify the procedure handle_example. By now, the required modifications should be straightforward.

Algorithm 4.3: Clint and shifting the bias.

```
procedure handle_example( p(t_1,...,t_n): example )
    if τ_2(p(t_1,...,t_n)) ≠ φ_2(KB, p(t_1,...,t_n))
    then if τ_2(p(t_1,...,t_n)) = false
            then call handle_covered( p(t_1,...,t_n) )
            else call handle_not_covered( p(t_1,...,t_n) )
            endif
            for all predicates q that depend on p do
                for all examples e of q do
                    call handle_example(e)
                endfor
            endfor
    endif
    if ∃e' ∈ postpone such that KB was modified
      since the previous attempt to call handle_uncovered(e', L_j)
    then delete e' from postpone
            call handle_example(e')
    endif
endproc

procedure handle_not_covered(e : uncovered positive example)
    i := 0; i is the bias index
    repeat
        i := i + 1
        call handle_uncovered(e, L_i)
    until i = M or handle_uncovered(e, L_i) did not signal an anomaly
    if handle_uncovered(e, L_i) signaled an anomaly
    then add e to postpone
    endif
endproc
```

Notice that this integration of Clint with the method to shift the bias, shifts the language bias at the clause level. Indeed, the procedure inspired by lemma 4.3 is used. This forms no problem as Clint only searches for one solution.

The extended algorithm of Clint retains its identification in the limit behavior (note that a perfect query answering procedure is assumed):

Theorem 4.2 *Any non-recursive predicate q will be error-free identified using Clint from any presentation of q if q is expressible in one of the languages $L_1, ..., L_M$ (such that $\forall i : L_i \subseteq L_{i+1}$) and the knowledge base is definitely correct for the predicates used in a complete and consistent concept-description of q in $L_1, ..., L_M$.*

Proof : The theorem directly follows from theorems 3.3 and 4.1. □

Theorem 4.3 *Any set of concepts C_i expressible in one of the languages $L_1, ..., L_M$ (such that $\forall i : L_i \subseteq L_{i+1}$) and KB_i will be postponed identified from a definitely correct starting knowledge base KB and from any presentation P of the concepts C.*

Proof : The theorem directly follows from theorems 3.4 and 4.1. □

Notice that these theorems imply that there is identification in the limit w.r.t. the language $L = \cup_{i=1}^{i=N} L_i$. This reveals that the method to shift the bias is mainly a clever *divide and conquer* algorithm. Indeed, it divides one language (L) into several sublanguages L_i. These sublanguages result in a smaller search space to consider at a time. This is computationally advantageous, provided that one does not need to shift the bias too much.

A session with the modified algorithm of Clint is shown in example 4.8.

Example 4.8: An example session with Clint

Suppose Clint starts learning from the following knowledge base:

```
female(alice).  male(luke).    parent(alice, kate).  parent(steve, kate).
female(kate).   male(morris).  parent(alice, luke).  parent(steve, luke).
female(rose).   male(jeffrey). parent(rose, steve).  parent(morris, steve).

has_children(H) ← parent(H,X)
```

Suppose also that Clint uses as language-series: $L_0, L_{0,1}, L_{0,1,0}, \ldots$ What follows is a session with Clint: user input is in **bold** and system output is typewritten.

```
?-learn.
Next example ?  @ grandparent(rose, luke).
Classification ?  @ true.
Starting Clause for grandparent(rose, luke) in L0:
```

In L_0, no existentially quantified variables are allowed

```
grandparent(G,C) ← female(G) ∧ male(C) ∧ has_children(G)

Is grandparent(steve,luke) true ?  { female(G) } deleted @ no.
Is grandparent(kate,luke) true ?  {has_children(G)} deleted @ no.
No more valid generalizations.  Asserted clause:

grandparent(G,C) ← female(G) ∧ male(C) ∧ has_children(G)

?-learn.
```

The user signals an error in the knowledge base.

```
Next example ?  @ grandparent(alice, luke).
Classification ?  @ false.
Is female(alice) true ?  @ yes.
Is male(luke) true ?  @ yes.
Is has_children(alice) true ?  @ yes.
```

The system locates the incorrect clause.

```
Retracting incorrect clause:

grandparent(G,C) ← female(G) ∧ male(C) ∧ has_children(G)

True fact grandparent(rose, luke) is not covered ...
Starting Clause for grandparent(rose, luke) in L0:

grandparent(G,C) ← female(G) ∧ male(C) ∧ has_children(G)
```

```
Starting Clause covers false fact grandparent(alice, luke);
Shifting bias to L0,1
Starting Clause for grandparent(rose, luke) in L0,1
```

In $L_{0,1}$, one unique existentially quantified variable is allowed to occur in each condition; however, no relations between these variables are allowed.

```
grandparent(G,C) ← female(G) ∧ male(C) ∧ has_children(G)
      ∧ parent(X,C) ∧ parent(G,Y)
```

Although Clint knows about the relation between parent and has_children it keeps both conditions explicit to facilitate the generalization.

```
Starting Clause covers false fact grandparent(alice, luke);
Shifting bias to L0,1,0
```

In $L_{0,1,0}$, relations between the existentially quantified variables introduced in the starting clause of $L_{0,1}$ are allowed. This explains why X occurs twice.

```
grandparent(G,C) ← female(G) ∧ male(C) ∧ has_children(G) ∧
      parent(X,C) ∧ parent(G,X) ∧ parent(Y,C) ∧ female(Y) ∧
      has_children(X) ∧ has_children(Y)

Is grandparent(morris,luke) true ? { female(G) } deleted @ yes.
Generalized clause:

grandparent(G,C) ← male(C) ∧ has_children(G) ∧
      parent(X,C) ∧ parent(G,X) ∧ parent(Y,C) ∧ female(Y) ∧
      has_children(X) ∧ has_children(Y)

I know grandparent(rose,luke) is true, so I generalize. {female(Y)} deleted
Generalized clause:

grandparent(G,C) ← male(C) ∧ has_children(G) ∧
      parent(X,C) ∧ parent(G,X) ∧ parent(Y,C) ∧
      has_children(X) ∧ has_children(Y)

Is grandparent(rose,kate) true ? { male(C) } deleted @ yes.
Generalized clause:

grandparent(G,C) ← has_children(G) ∧ parent(X,C) ∧ parent(G,X)
      ∧ parent(Y,C) ∧ has_children(X) ∧ has_children(Y)

Is grandparent(rose,steve) true ? { parent(X,C) } deleted @ no.
No more valid generalizations ; Asserted clause:


grandparent(G,C) ← parent(X,C) ∧ parent(G,X)
```

Knowledge about the relation between different predicates is used to simplify the clause.

4.5 Related Work.

In this chapter I reported on work that is also contained in [De Raedt and Bruynooghe 90a]. An earlier version of this paper appeared in [De Raedt and Bruynooghe 89d] and the integration of the presented technique with Clint was presented in [De Raedt and Bruynooghe 89e].

Our approach to cope with indirect relevance is related to explanation based learning [Mitchell, Keller and Kedar-Cabelli 86, De Jong and Mooney 86], to other approaches that address bias in concept-learning [Mitchell 80, Utgoff and Mitchell 82, Utgoff 86, Grosof and Russell 90, Russell and Grosof 90, Morik 89, Wrobel 89], to classical concept-learning systems like versionspaces [Mitchell 77, Mitchell 82, Hirsh 90, Subramanian and Feigenbaum 86, Murray 87], AQ [Michalski 83], Marvin [Sammut and Banerji 86] and to work on indirect relevance [Buntine 87].

A justification can be considered as a weak form of an explanation: it contains the features which *may* indicate why the example belongs to the concept. Given this point of view, Clint's knowledge intensive induction technique can be interpreted as an approach to the imperfect theory problem of explanation based learning [Mitchell, Keller and Kedar-Cabelli 86, De Jong and Mooney 86].

The techniques presented in this chapter are also related to classical concept-learning approaches as [Michalski 83, Sammut and Banerji 86, Hirsh 90] where the examples specify some relevant relations, and where the system extends the descriptions of the example using predicates in the knowledge base. However, in these approaches, the problem is simplified as the system starts from a justification and generalizes [Sammut and Banerji 86] or extends it [Michalski 83], while our technique is not given any justification at all. In this respect, our work resembles Buntine's Plausible Generalization Algorithm. The Plausible Generalization Algorithm copes with indirect relevance within a predefined language in a similar way as the early version of Clint, presented in chapter 3. So, it does not shift its bias.

The presented technique is also applicable to versionspaces [Mitchell 77] and its extensions [Subramanian and Feigenbaum 86, Hirsh 90, Murray 87]. Indeed, in section 4.4.1, it was shown how our technique can be used to determine the initial versionspace and to shift the language bias when the versionspace becomes empty. As this are the only required changes to the versionspace mechanism, it is straightforward to apply the resulting algorithm in the contexts of [Subramanian and Feigenbaum 86, Hirsh 90, Murray 87]. A particularly relevant extension, would be the one obtained by using Hirsh's technique. Hirsh describes an integration of explanation based learning with versionspaces that copes with the imperfect theory problem. However, he also assumes that the language bias is given at the start. Using our technique this assumption would not be necessary.

Finally, this chapter is also related to the work of [Utgoff and Mitchell 82, Utgoff 86, Grosof and Russell 90, Russell and Grosof 90, Russell and Subramanian 90] on declarative bias and shifting the concept-description language bias. Early work [Utgoff and Mitchell 82, Utgoff 86] on bias has concentrated on propositional calculus and is therefore rather limited. One of the main contributions of this chapter is the introduction of series of languages that are subsets of first order logic and that can be used for shifting the language bias. Hence, the work of Utgoff is extended here. Algorithm 4.1 is very similar to the nonmonotonic reasoning approach of Grosof and Russell. In their work, language biases are considered to be defaults, and the bias is shifted whenever there is evidence that

the default does not hold. Also, preferences of certain biases can be declared. [Russell and Grosof 90] discuss how biases can be represented in a declarative way, while [Russell and Subramanian 90] show how (ir)relevance can be used to reformulate a given bias. However, none of the mentioned approaches provide clear indications of how bias can be acquired. Rather, they assume it to be given (by the user ?). In contrast, our approach shows how domain independent abstract language definitions, can be used to compute specific language biases.

Our work on bias is also related to the notion of bias in the system Blip [Morik 89, Wrobel 89]. Blip is a model-driven learner that learns by instantiating and testing schemata. Each schema can be viewed as a particular language bias. Because the order of instantiating the schemata is based on complexity of the schemata, this aspect is quite similar to our shift of bias. Schemata (see also chapter 8) differ however from our languages.

4.6 Summary.

A general and original approach to cope with indirect relevance in inductive concept-learning was presented. This technique uses facts instead of ground clauses to represent examples. The examples and abstract language definitions are used to compute a specific language bias, which determines the versionspace of concept-descriptions. The version-space consistent with one positive example is determined by the set of justifications. These justifications contain the relevant relations holding for the example. Because the description language of concepts may depend on the particular learning task, a domain-independent technique must be able to recover from an incorrect bias. As the language biases are computed from the knowledge base, the examples and a series of abstract languages, a particular bias may not allow to find a correct concept-description. Therefore, a technique that shifts the language bias was developed. An operational criterion to detect when the bias is incorrect and a method to shift it were presented. Using this framework, learning occurs at two different levels: one to determine the correct bias and another one to learn the concept given the correct bias. The method was presented in general and it was claimed that it applies to concept-learning systems in general. To support this claim, it was shown how it can be integrated in versionspaces and the version of Clint presented in chapter 3. The integration of the method with Clint leads to an interesting system as it requires only little information from the user. This contributes to the applicability of Clint in learning apprentice systems operating in a weak, incomplete or imperfect domain-theory [Rajamoney and De Jong 87, Tecuci and Kodratoff 90, Kodratoff and Tecuci 87].

Chapter 5

Meta-Knowledge and Constraints.

5.1 Introduction.

This chapter addresses the use of integrity constraints in concept-learning. A method is presented which makes it possible to augment the induction process with knowledge about what has to be learned. The user can specify properties that must be satisfied by the learned knowledge base. For example, the user can tell the system that a `parent` should be a `father` or a `mother`. Such properties are stated as general first order logic clauses such as `parent(X,Y)` $\rightarrow$ `father(X,Y)` $\vee$ `mother(X,Y)`. As the properties play a similar role as integrity constraints in database technology [Ullmann 89], they are termed constraints. The learner takes care that the specified constraints are satisfied by the learned knowledge base. When a constraint is violated, the learner generates questions to the oracle. These questions correspond to examples and allow to recover from violations. Because clauses are used to represent constraints, the approach is very general. It augments the induction process, which used to be knowledge poor, with very rich knowledge-sources. We argue that this is a major extension of inductive techniques. The extension is knowledge intensive and can therefore be viewed as an attempt to bridge the gap between similarity based learning and explanation based learning. In particular the issues concerned with the imperfect theory problem are addressed. It is shown how Clint can be extended with constraint handling abilities. The resulting properties of Clint are also discussed.

From the point of view of meta-knowledge, there is – to the best of my knowledge – no approach that is able to cope with meta-knowledge in our sense. There is in fact only one system that uses meta-knowledge while learning, i.e. the system Blip [Morik 89, Wrobel 89], but Blip's notion of meta-knowledge differs from ours (see also section 5.6).

This chapter is organized as follows : in section 5.2, a framework for learning from constraints is presented; in section 5.3, some principles to learn from constraints are derived; in section 5.4 a lazy method to handle constraints is integrated in Clint; in the next section, an eager approach to constraint handling is integrated in Clint; finally, we discuss some related work and draw some conclusions.

5.2 Constraints in interactive concept-learning.

5.2.1 Integrity constraints

We first define the issues involving constraints and show then how to redefine the problem statement of Clint to cope with constraints.

The key notion in this chapter is:

Definition 5.1 *A constraint is a range restricted first order logic clause [Genesereth and Nilsson 87]. A constraint* $\neg p_1 \vee \neg p_2 \vee ... \vee \neg p_n \vee q_1 \vee ... \vee q_k$ *is represented by the implication* $p_1 \wedge p_2 \wedge ... \wedge p_n \rightarrow q_1 \vee ... \vee q_k$. $p_1 \wedge p_2 \wedge ... \wedge p_n$ *is the body of the constraint and* $q_1 \vee ... \vee q_k$ *the head.*

Throughout this thesis it will be assumed that constraints are range-restricted. Remember from definition 3.26 that range-restriction means that all variables occurring in the head of a clause also occur in the body of a clause. The reason for this restriction is to assure that constraints can be used for generating questions. This restriction is often imposed in database systems.

Notice that constraints are very powerful as they allow to specify very general knowledge. In terms of clauses, it is easy to specify properties such as transitivity, symmetry, reflectivity, number restrictions such as in KL-one [Brachman and Schmolze 85] and Blip [Morik 89, Thieme 89]. Constraints are a generalization of examples. Indeed, each positive example p can be represented by the constraint (`true` $\rightarrow p$) and each negative example n by the constraint ($n \rightarrow$ `false`). The constraint theory is the set of all known constraints. From now on we will assume that:

> *All gathered examples, constraints and answers to queries are added as constraints to the constraint theory.*

This will allow to treat constraints and examples uniformly.

A second way in which constraints are related to examples is that examples and constraints both provide information about the intended interpretation. Indeed, all constraints are assumed to be true in the intended interpretation. Therefore, they also have to be satisfied by the learned knowledge base.

Definition 5.2 *A constraint I is satisfied by a knowledge base KB of concept-descriptions iff* $KB \models I$. *Otherwise I is violated.*

Given a constraint $p_1 \wedge p_2 \wedge ... \wedge p_k \rightarrow q_1 \vee ... \vee q_m$ and a knowledge base KB, the constraint is satisfied by KB iff the query $\leftarrow p_1 \wedge p_2 ... \wedge p_k \wedge \neg q_1 \wedge ... \wedge \neg q_m$[1] fails. Otherwise, the constraint is violated. The substitutions that make the query true indicate the source of the violations.

The modified problem-specification for Clint now becomes:

[1]The form of this query differs slightly from the one assumed by a perfect query answering mechanism. However, the query can be rewritten as $\leftarrow p_1 \wedge p_2 ... \wedge p_k \wedge \neg q$ while asserting $q \leftarrow q_i$ for all i. So, if a perfect query answering mechanism is used, one can safely verify the satisfaction of constraints.

Problem Statement 5.1 *Constraint Specification*

- **Given:**
  - a set of constraints C, $\forall i \in C : \tau_2(i) = \text{true}$
  - a knowledge base KB of learned and basic predicates
  - a series of languages $L_1, ..., L_M$ defining Horn-clauses
  - a new constraint c (possibly an example)
  - an oracle willing to answer membership questions about τ
- **Find:** a knowledge base KB' obtained by retracting and asserting clauses belonging to L_j for some j such that KB' satisfies all constraints in $C \cup \{c\}$.

Notice that the modified problem specification talks about constraints instead of examples. The reason is that examples are merely a special case of constraints. This problem statement is illustrated in example 5.1.

Example 5.1: Illustrating the notion of constraints.

Reconsider example 3.2. Assume that the starting knowledge base is used and that the constraint `is_allowed_to_drive(X,Y) ∧ isa(Y,car) → license(X,car)` is supplied to the system. The constraint is violated for $\theta = \{$ `X=luc, Y=dyane` $\}$. This implies that one of the predicates occurring in the constraint is incorrect. The knowledge base can be made consistent by replacing the current definition of `is_allowed_to_drive` by the one given at the end of example 3.2.

The reasons for introducing constraints can be summarized as follows (see also the literature on databases, e.g. [Reiter 90]) :

- The expressiveness of the used knowledge formalism does not permit to add constraints to the knowledge base and use them for deduction.
- When adding constraints to the knowledge base would be allowed, one would obtain a much more complicated knowledge base with many redundancies. This would decrease the efficiency of deductions.
- Certain constraints should in no case be used as inference rules. Think for examples about the following rule : `human(X) ∧ age(X,Y) → Y < 150`. One does not want to use such rules to derive facts for the predicate $<$.

For these reasons we adopt the view of [Reiter 90] that *integrity constraints are statements about the contents of the knowledge base whereas truths about the world belong in the database*. For a further discussion of these issues I refer to the database literature.

5.2.2 The relation between integrity constraints and meta-knowledge

Integrity constraints are normally not considered as meta-knowledge according to logic programming principles [2](see e.g. [Kowalski 79]). The reason why we will nevertheless sometimes refer to integrity constraints as meta-knowledge is that some forms of meta-knowledge can be easily translated into integrity constraints (see example 5.2). The example reveals that the difference has more to do with the syntax than with the semantics. Also, the meta-level can often be implemented by using integrity constraints at the object-level and translating meta-level statements into object-level ones. For more details about such translations and a system that works this way, see the literature on Blip [Morik 89, Thieme 89, Wrobel 89, Emde 89].

Example 5.2: Translating meta-level into object-level

When we would specify at the meta-level that the predicate `isa` is transitive, we would obtain a statement as `transitive(isa)` ← . Furthermore, there would be a definition for `transitive` such as the following one `transitive(P)` ↔ `(P(X,Y)` ∧ `P(Y,Z)` → `P(X,Y))`. The meta-level statement `transitive(isa)` ← can now be translated into the object level statement `isa(X,Y)` ∧ `isa(Y,Z)` → `isa(X,Z)`. The statement at the object level is an integrity constraint.

5.3 Principles for learning from constraints.

In the next few sections, methods to extend Clint with constraint handling mechanisms will be presented. We will present two techniques: a *lazy* technique, which is rather straightforward and which uses the constraints mainly for verifying the results of the learning; and an *eager* approach, which uses the constraints in an active way. We will also discuss the correctness and convergence of the resulting algorithms. However, before presenting the techniques, some principles of learning from constraints are presented.

The general principle for using constraints for concept-learning is rather simple. If a constraint $p_1 \wedge ... \wedge p_k \rightarrow q_1 \vee ... \vee q_m$ is violated for a substitution θ we have that:

- one of the predicates p_i in the body of the constraint is too general. In that case $\varphi_2(KB, p_i\theta) =$ true but should be false.
- Or, one of the predicates q_j in the head of the constraint is too specific. In that case $\varphi_2(KB, q_j\theta) =$ false but should be true.

This observation reveals that violated constraints can be used to generate examples that are not satisfied by the knowledge base. If there is only one literal in a constraint, it is clear that the violated instance of the literal itself is an example that is not satisfied by the knowledge base. However, if there is more than one literal in a violated constraint, the responsible examples have to be identified by performing credit-assignment on a violated

[2]This was noticed by Daniel De Schreye.

instance of a constraint. In interactive learners, the credit-assignment procedure queries the oracle for the truthvalue in the intended interpretation of the instances of the literals in the violated constraint. This is illustrated in example 5.3.

Example 5.3: Credit-assignment.

Consider the following constraint :

`is_allowed_to_drive(X,Y)` $\wedge$ `isa(Y,car)` $\rightarrow$ `license(X,car)`

Let us assume that the constraint is violated for $\theta =$ { `X=luc, Y=dyane` }. This means that $\varphi_2(KB,$ `is_allowed_to_drive(luc,dyane)`$) =$ true, $\varphi_2(KB,$ `isa(dyane,car)` $) =$ true and $\varphi_2(KB,$ `license(luc,car)`$) =$ false. The source of the violation can be located by asking the oracle for the intended interpretation of the corresponding facts. The error is then the fact f for which $\varphi_2(KB, f) \neq \tau_2(f)$. So, we have that `is_allowed_to_drive` or `isa` is too general or `license` is too specific.

5.4 The lazy method.

The examples and the discussion in the previous section motivate the following straightforward *lazy* method to cope with constraints in the context of Clint.

When a constraint is violated, credit is assigned to the literals in a violated instance of the constraint. This allows to derive examples that are not satisfied by the given knowledge base. These examples are passed to Clint, which treats them as usual. Once Clint finishes learning from these examples, it is verified whether the constraints are satisfied by the resulting knowledge base. Violations at that point are treated in the same manner. This process of verifying constraints, deriving examples from violated instances of constraints, learning from examples by Clint, ... is repeated until the knowledge base satisfies all constraints. Then, the user is queried for additional constraints (or examples).

All this would be fine, if it were not the case that Clint may run into anomalies. As usual, special attention has to be paid to anomalies. When an anomaly arises for a predicate p, one should not expect the constraints mentioning the predicate p to be satisfied. Also, the predicates (and the constraints) that depend on the anomalous predicate, may be affected.

Definition 5.3 *A constraint c depends on a predicate p if*

1. *there is a literal in c of the predicate p, or*
2. *there is literal in c of a predicate q that depends on p or*

Therefore the affected constraints will be deactivated. Deactivating constraints is similar to postponing examples as in chapter 3. When all anomalies of predicates on which a certain constraint depends are solved, an attempt will be made to reactivate the constraint. This will involve testing whether the constraint is satisfied, and when necessary assigning credit and derive examples.

5.4.1 The lazy method: algorithm.

Let us first define the notion of an activated constraint, which is used by the modified algorithm of Clint.

Definition 5.4 *An anomaly for a constraint c is an example $p(t_1,...,t_n)$ such that $p(t_1,...,t_n) \in$ postpone and c depends on p.*

Definition 5.5 *A constraint c is activated iff*

1. *c corresponds to an example and $c \notin$ postpone or*
2. *there is no anomaly for the constraint c.*

Clearly, if a constraint is not activated, it is *deactivated.*

Notice that the class of activated constraints evolves when anomalies are added or removed from postpone. Examples are a special case of constraints. Examples are always activated unless they are on postpone. Remember that algorithm 5.1 assumes that all constraints, examples and answers to oracle queries are all added to the constraint theory as constraints! As in previous chapters, this is not written explicitly in the algorithms.

Let us now explain algorithm 5.1. Observe that examples and constraints are now treated uniformly at the top-level by Clint. At certain points however examples are handled in a different way than ohter constraints.

The procedure handle_constraint only takes into account activated constraints. The class of activated constraints evolves because postpone may be modified by the procedure handle_not_covered and the last if-test of the procedure handle_constraint. Given a violated activated constraint, the procedure handle_constraint performs credit assignment in order to find an example that is not satisfied by the current knowledge base. This is realized by asking the oracle for the truthvalue of literals in a violated instance[3] of the constraint. Note that if the violated constraint contains only one literal (as in the case of examples), its truthvalue is known. Then the oracle need not be bothered with questions. In any case, the while loop allows to identify a violated example. This example is then passed to the basic procedures of Clint presented in chapters 3 and 4. Because the knowledge base will be modified by Clint in order to satisfy the example, the predicates depending on the example may also be affected. Therefore, the system verifies whether the affected examples and constraints are satisfied by the modified knowledge base. In general, it is a good heuristic to test examples before general constraints as this usually requires less questions to the oracle. After verifying the possibly affected constraints, the system attempts to recover from some anomalies as in previous versions of the system. Notice that the deletion of a postponed example from postpone activates the corresponding constraint!

Because the procedure handle_covered calls the procedure handle_example, this last procedure is generalized in a straightforward way.

This algorithm is illustrated in example 5.4.

[3]This instance can be made ground using a similar argument as the one for obtaining a ground proof tree while locating an incorrect clause in section 3.5.

Algorithm 5.1: The lazy constraint algorithm

```
procedure meta_clint
        postpone := ∅
        while the oracle wants to input constraints do
                read the next constraint c and add it to the constraint theory
                call handle_constraint(c)
        endwhile
endproc

procedure handle_constraint(c : constraint)
        if c is activated
        then if c is violated for a substitution θ
                then while there is no literal l ∈ cθ such that τ2(l) ≠ φ2(KB,l) do
                                select a literal l' ∈ cθ such that τ2(l') is not known
                                ask the oracle for τ(l')
                        endwhile
                        select an example p(t1,...,tn) ∈ cθ
                          such that τ2(p(t1,...,tn)) ≠ φ2(KB;p(t1,...,tn))
                        if τ2(p(t1,...,tn)) = false
                        then call handle_covered(p(t1,...,tn))
                        else call handle_not_covered(p(t1,...,tn))
                        endif
                        for all constraints c' that depend on p do
                                call handle_constraint(c')
                        endfor
                        if ∃e ∈ postpone such that KB was modified
                          since the previous call to handle_uncovered(e,Lj)
                        then delete e from postpone; this activates e
                                call handle_constraint(e)
                        endif
                endif
        endif
endproc

procedure handle_example( e : example )
        call handle_constraint( e )
endproc
```

Example 5.4: An example with the lazy algorithm

Suppose the knowledge base contains the following definitions:

```
female(alice) ←                    male(luke) ←
female(kate) ←                     male(morris) ←
female(rose) ←                     male(jeffrey) ←
female(mary) ←                     male(steve) ←

parent(jeffrey, kate) ←            parent(alice, kate) ←
parent(alice, luke) ←              parent(mary, luke) ←
parent(rose, mary) ←               parent(morris, jeffrey) ←
parent(alice, steve) ←

father(X,Y) ← parent(X,Y) ∧ female(Y)
```

Notice that this knowledge base contains an erroneous definition : the clause for **father**. Assume that it was input by the user but not as a basic predicate. Assume also that **parent**, **male** and **female** are basic predicates. What follows is a session with the lazy method:

```
?-learn.
Next constraint ? @ mother(X,Y) ∧ father(X,Y) → false.
Verifying constraint mother(X,Y) ∧ father(X,Y) → false ...
Constraint satisfied
Next constraint ? @ parent(X,Y) → father(X,Y) ∨ mother(X,Y).
Verifying constraint parent(X,Y) → father(X,Y) ∨ mother(X,Y) ...
Constraint violated :
parent(mary, luke) → father(mary, luke) ∨ mother(mary, luke)
I already know parent(mary, luke) is true
```

This is because **parent** is a basic predicate.

```
Is father(mary, luke) true ? @ no.
Then mother(mary, luke) must be true ...
Starting clause for mother(mary, luke) in L0 :

      mother(X,Y) ← female(X) ∧ male(Y) ∧ parent(X,Y)

Is mother(rose, mary) true ? { male(Y) } deleted @ yes.
Generalized clause :

      mother(X,Y) ← female(X) ∧ parent(X,Y)

Is mother(morris, jeffrey) true ? { female(Y) } deleted @ no.
No more valid generalizations. Asserted clause :

      mother(X,Y) ← female(X) ∧ parent(X,Y)
```

```
Verifying constraint mother(X,Y) ∧ father(X,Y) → false
Constraint violated :
mother(rose, mary) ∧ father(rose, mary) → false
I already know that mother(rose, mary) is true
Then father(rose, mary) must be false
Retracting incorrect clause :

      father(X,Y) ← parent(X,Y) ∧ female(Y)

Verifying constraint mother(X,Y) ∧ father(X,Y) → false
Constraint satisfied
Verifying constraint parent(X,Y) → father(X,Y) ∨ mother(X,Y) ...
Constraint violated :
parent(morris,jeffrey) → father(morris,jeffrey) ∨ mother(morris,jeffrey)
I already know parent(morris, jeffrey) is true
Is father(morris, jeffrey) true ? yes.
Starting clause for father(morris, jeffrey) in L_D :

      father(X,Y) ← male(X) ∧ male(Y) ∧ parent(X,Y)

Is father(jeffrey, kate) true ? { male(Y) } deleted @ yes.
Generalized clause :

      father(X,Y) ← male(X) ∧ parent(X,Y)

I already know that father(rose, mary) is false
No more valid generalizations. Asserted clause :

      father(X,Y) ← male(X) ∧ parent(X,Y)
```

5.4.2 The lazy method: properties.

Because of the modified problem statement, it is necessary to adapt our notions of convergence:

Definition 5.6 *A constraint presentation P of a set of concepts C w.r.t. a knowledge base KB is a possibly infinite sequence of constraints such that*

1. *all constraints in P are true in the intended interpretation τ_2*
2. *all constraints in P mention only predicates in C or KB*
3. *all examples of predicates in C occur as a constraint in P.*

Definition 5.7 *An algorithm identifies a set of concepts C error-free in the limit from a starting knowledge base KB and a constraint presentation P of w.r.t. KB if*

1. *for all constraints c in P, after the user inputs c, the algorithm always outputs a knowledge base that satisfies c*

2. *there is a finite time after which the learned knowledge base is definitely correct for the predicates in C and is not modified afterwards.*

Definition 5.8 *An algorithm identifies a set of concepts C with postponing in the limit from a starting knowledge base KB and a constraint presentation P of w.r.t. KB if there is a finite time after which the learned knowledge base is definitely correct for the predicates in C and is not modified afterwards.*

The following properties hold for the lazy method:

Theorem 5.1 *Any non-recursive predicate q will be error-free identified using the lazy method from a definitely correct knowledge base KB and any constraint presentation of q w.r.t. KB if q is expressible in terms of the languages $L_1, ..., L_M$ and KB.*

Proof : This follows directly from theorem 4.2 because the same procedure as in theorem 4.2 is used to handle examples. Constraints are only used to generate examples. □

Theorem 5.2 *Any set of concepts C, expressible in one of the languages $L_1, ..., L_M$ and KB, will be postponed identified using the lazy method from a definitely correct starting knowledge base KB and any constraint presentation of C w.r.t. KB.*

Proof : This follows directly from theorem 4.3 because the same procedure as in theorem 4.3 is used to handle examples. Constraints are only used to generate examples. □

5.5 The eager method.

The disadvantage of the lazy method is that the constraints are only used for verification. The lazy method implements a kind of generate and test strategy to cope with constraints. Indeed, from the lazy procedure's point of view, the constraints are used to generate knowledge bases, which are tested against the constraint theory. The constraints generate knowledge bases since when they are violated they allow to derive examples which result in a modified knowledge base. It is generally known that generate and test approaches can be improved by pruning failing paths as soon as possible. In the eager method, the verification of the constraints is integrated with the learning in order to realize this.

5.5.1 Eager method: algorithm

In concept-learning there are two basic operations: specialization and generalization (see chapter 2). Specialization often occurs when a negative example is covered by the current concept-description whereas generalization is performed when the current concept-description does not cover a positive example. In section 5.3, it was shown how a similar property can be used for dealing with constraints. This motivates the following definitions:

Definition 5.9 *A constraint c is generative for a predicate p iff there is an occurrence of p in head(c) and no occurrence of p in body(c).*

Definition 5.10 *A constraint c is restrictive for a predicate p iff there is an occurrence of p in body(c) and no occurrence of p in the head(c).*

The idea here is that, while generalizing a predicate p, one should take into account the restrictive constraints for p whereas the generative constraints for p should be taken into account while specializing. The use of restrictive and generative constraints is illustrated in example 5.5.

Example 5.5: Deriving examples from constraints.

Consider the following constraint :

`parent(X,Y)` $\rightarrow$ `mother(X,Y)` $\vee$ `father(X,Y)`

Assume first that:

- The definition of `parent` is being learned; this implies that the constraint is restrictive.
- the definitions of `mother` and `father` are correct
- the current hypothesis for `parent` violates the constraint for the fact `parent(rose, luke)`; this means that $\varphi_2(KB,$ `mother(rose, luke)`$)=$ false, $\varphi_2(KB,$ `father(rose, luke)`$)=$ false, but $\varphi_2(KB,$ `parent(rose, luke)`$)=$ true.

Under these assumptions, the reason for the violation must be that the current hypothesis for `parent` is incorrect (too general). This also implies that $\tau_2($`parent(rose, luke)`$)=$ false.
If we assume the following :

- The definition of `mother` is being learned; this implies that the constraint is generative.
- the definitions of `parent` and `father` are correct
- the current hypothesis for `mother` violates the constraint for the fact `mother(rose, luke)`; this means that $\varphi_2(KB,$ `parent(rose, luke)`$)=$ true, $\varphi_2(KB,$ `mother(rose, luke)`$)=$ false and $\varphi_2(KB,$ `father(rose, luke)`$)=$ false.

then the reason for the violation must be that the current hypothesis for `mother` is too restricted (too specific). This also implies that $\tau_2($`mother(rose, luke)`$)=$ true.

Example 5.3 also illustrates that — under certain conditions — it is possible to actively use the constraints while learning. The conditions state that the predicate being learned should be responsible for a violated constraint. This is the case when the violated constraint does not depend on predicates that are known to be incomplete or inconsistent, with the exception of the predicate being learned. This observation leads to the notion of applicable constraints:

Definition 5.11 *A constraint c is applicable to a predicate p iff*

1. *c is an example or*
2. *p occurs in c and c does not depend on a predicate $q \neq p$ that is incomplete or inconsistent with the known examples.*

If there is a violation in an applicable constraint, it is very likely that the violation is due to the predicate p to which the constraint is applicable.

Specializing and generalizing one predicate may affect other predicates and also constraints that do not mention the modified predicate. In order to minimize the interactions between the learning of different predicates, the constraints that are not applicable to the current predicate will be treated as in the lazy approach.

In the context of Clint, actively using the constraints while learning, means verifying the restrictive constraints during generalization. This will be realized by modifying the procedures handle_uncovered and generalize_clause. If during generalization violations occur, the system will first suspect the predicate being generalized. This means that it will first generate questions to the oracle about the literals concerning the predicate that is currently being generalized. Only in case these literals are not violated by the current definition of the predicate, will the system ask questions about the other predicates.

As Clint uses a rather radical refinement transformer (removing clauses from the knowledge base), we will not use the generative constraints while specializing but verify them at the end of the procedure handle_constraint as in the lazy approach.

These observations lead to the modified algorithm shown in algorithm 5.2 and 5.3. The procedure handle_constraint of the lazy strategy is retained in order to handle the constraints that are not applicable to the current predicate. It is merely the procedures handle_uncovered and generalize_clause used by handle_not_covered that change. These modified procedures are summarized in algorithms 5.2 and 5.3.

Let us now explain these modifications. As a matter of fact, there is only one change: the while-loop at level 2 in the procedure generalize_clause and the outer while-loop in the procedure handle_uncovered. These two while-loops are nearly identical. Therefore they will be discussed together. The aim of the while-loops is to guarantee that the candidates (i.e. c, the starting clause or justification in the procedure handle_uncovered and c', the candidate generalizations in generalize_clause) satisfy the restrictive constraints for the predicate. In order to do so it is verified whether asserting the candidates in the knowledge base would give rise to any violations. If there would be any violations, credit-assignment is performed on a violated instance of the constraint. As indicated above, the algorithm should first ask for the truthvalue of literals of the predicate p currently being learned. In any case, the credit assignment procedure results in an example that is violated by the knowlege base. There are three cases to consider:

- The violated example is a negative example $p(u_1, ..., u_n)$ for the predicate p. Then $p(u_1, ..., u_n)$ must occur in $body(r)$. In this case, the candidate covers a negative example and has to be rejected. This is done in the same manner as in previous versions of the algorithm because the while-loop terminates.
- The violated example is an example l for a different predicate and l occurs in $body(r)$. Then the algorithm will first handle the error in the other predicate. In the call to handle_constraint(l) the constraint for $p(t_1, ..., t_n)$ will be called a second time after

Algorithm 5.2: The eager constraint algorithm

```
procedure generalize_clause( c : clause, p(t_1,...,t_n) : positive example)
        : returns maximally general clause
    Q := λ_g(c)
    while Q ≠ ∅ do
        delete c' from Q
        while KB ∪ {c'} violates an applicable restrictive constraint r for p
              and c' does not cover negative examples do
            select an applicable restrictive constraint r that is violated
              by KB ∪ {c'} for a substitution θ
            while there is no known literal l ∈ rθ
                      such that τ_2(l) ≠ φ_2(KB ∪ {c'}, l) do
                select a literal l' ∈ rθ such that τ_2(l') is not known
                ask the oracle for τ(l')
            endwhile
            if p(u_1,...,u_n) ∈ body(rθ) and τ_2(p(u_1,...,u_n)) = false and
              τ_2(p(u_1,...,u_n)) ≠ φ_2(KB ∪ {c'}, p(u_1,...,u_n))
            then void c' covers a negative example
            else if there is a literal l ∈ body(rθ)
                      such that τ_2(l) ≠ φ_2(KB, l)
                 then call handle_constraint(l)
                 else void r is no longer applicable
                 endif
            endif
        endwhile
        if c' does not cover negative examples
        then it does not violate applicable restrictive constraints
            if c' covers an example e that is not covered by c
            then ask the oracle for τ(e)
                 if τ_2(e) = true
                 then return generalize_clause(c', p(t_1,...,t_n))
                 endif
            else Q := Q ∪ λ_g(c')
            endif
        endif
    endwhile
    return c
endproc
```

Algorithm 5.3: The eager constraint algorithm (continued)

```
procedure handle_uncovered( p(t_1,...,t_n) : uncovered positive example, L : language bias )
    S := JUS(KB, L, p(t_1,...,t_n))
    c' := void
    repeat
        delete a clause c from S
        while KB ∪ {c} violates an applicable restrictive constraint r for p
              and c does not cover negative examples do
            select an applicable restrictive constraint r that is violated
              by KB ∪ {c} for a substitution θ
            while there is no literal l ∈ rθ such that τ_2(l) ≠ φ_2(KB ∪ {c}, l) do
                select a literal l' ∈ rθ such that τ_2(l') is not known
                ask the oracle for τ(l')
            endwhile
            if p(u_1,...,u_n) ∈ body(rθ) and τ_2(p(u_1,...,u_n)) = false and
              τ_2(p(u_1,...,u_n)) ≠ φ_2(KB ∪ {c}, p(u_1,...,u_n))
            then void c covers a negative example
            else if there is a literal l ∈ body(rθ)
                      such that τ_2(l) ≠ φ_2(KB, l)
                 then call handle_constraint(l)
                 else void r is no longer applicable
                 endif
            endif
        endwhile
        if c does not cover negative examples
        then it does not violate applicable restrictive constraints
            c' := generalize_clause( c, p(t_1,...,t_n) )
        endif
    until c' ≠ void or S = ∅
    if c' ≠ void
    then assert c' in KB
    else signal anomaly
    endif
endproc
```

recovering from the error in l. Recovery will involve the retraction of an incorrect clause.

- The violated example is an example l for a different predicate and l occurs in $head(r)$. In this case there is nothing wrong with the current predicate p. Therefore the constraint becomes automatically not applicable and will be reconsidered later. So, the constraint becomes not applicable and the system will continue generalizing the current predicate p.

A session with this algorithm is shown in example 5.6.

Example 5.6: A session with the eager method

Assume the same knowledge base as in example 5.4 is used. What follows is a session with the eager method:

```
?-learn.
Next constraint ?  @ mother(X,Y) ∧ father(X,Y) → false.
Verifying constraint mother(X,Y) ∧ father(X,Y) → false ...
Constraint satisfied
Next constraint ?  @ parent(X,Y) → father(X,Y) ∨ mother(X,Y).
Verifying constraint parent(X,Y) → father(X,Y) ∨ mother(X,Y) ...
Constraint violated :
parent(mary, luke) → father(mary, luke) ∨ mother(mary, luke)
I already know parent(mary, luke) is true
Is father(mary, luke) true ?  @ no.
Then mother(mary, luke) must be true ...
Starting clause for mother(mary, luke) in L0 :

      mother(X,Y) ← female(X) ∧ male(Y) ∧ parent(X,Y)

Is mother(rose, mary) true ?  { male(Y) } deleted
This instance violates father(X,Y) ∧ mother(X,Y) → false @ yes.
Then father(rose, mary) must be false
Retracting incorrect clause :

      father(X,Y) ← parent(X,Y) ∧ female(Y)

Verifying constraint mother(X,Y) ∧ father(X,Y) → false ...
Constraint satisfied
Verifying constraint parent(X,Y) → father(X,Y) ∨ mother(X,Y) ...
Constraint violated :
parent(mary, luke) → father(mary, luke) ∨ mother(mary, luke)
I already know parent(mary, luke) is true
I already know father(mary, luke) is false
I already know mother(mary, luke) is true
Starting clause for mother(mary, luke) in L0 :

      mother(X,Y) ← female(X) ∧ male(Y) ∧ parent(X,Y)
```

```
I already know mother(rose, mary) is true
Generalized clause :

      mother(X,Y) ← female(X) ∧  parent(X,Y)

Is mother(morris, jeffrey) true ?  { female(X) } deleted @ no.
No more valid generalizations.  Asserted clause :

      mother(X,Y) ← female(X) ∧  parent(X,Y)

Verifying constraint parent(X,Y) → father(X,Y) ∨  mother(X,Y)
Constraint violated :
parent(morris,jeffrey) → father(morris,jeffrey) ∨  mother(morris,jeffrey)
I already know parent(morris, jeffrey) is true
Is father(morris, jeffrey) true ?  yes.
Starting clause for father(morris, jeffrey) in E_D :

      father(X,Y) ← male(X) ∧  male(Y) ∧  parent(X,Y)

Is father(jeffrey, kate) true ?  { male(Y) } deleted @ yes.
Generalized clause :

      father(X,Y) ← male(X) ∧  parent(X,Y)

I already know that father(mary, luke) is false
No more valid generalizations.  Asserted clause :

      father(X,Y) ← male(X) ∧  parent(X,Y)
```

Although the examples 5.4 and 5.6 yield the same result, they show a different way of obtaining this result. Furthermore, example 5.6 reveals that the eager method usually detects and corrects errors faster than the lazy method. Although in the given (small) examples this was not really important, the eager method can reduce the amount of questions and work significantly. Such situations arise when the predicate p being learned depends on an incorrect predicate q. The sooner the error in q will be corrected, the sooner that the algorithm will converge on a good solution for p. Indeed, assume that the lazy method is used and that p is learned. However, after the learning of p, the constraints are verified and it is detected that q is incorrect. In that case, part of the learning for p has to be redone.

5.5.2 Eager method: properties

The theoretical properties of the eager algorithm follow from the ones of the lazy approach. There is however one aspect that deserves special attention: termination.

Lemma 5.1 *The eager method always terminates when equipped with a perfect query answering mechanism.*

Proof : There are two possible reasons why the algorithm (say: the while-loops) would not terminate: either (1) handle_constraint could call itself an infinite number of times or (2) the conditions of the inserted while-loop could be true an infinite number of times.

(1): Handle_constraint cannot call itself an infinite number of times because each call gives rise to the retraction (and detection) of an incorrect clause. The reason is that Handle_constraint is always called with a covered negative example (a literal of the body) from our while-loops. Furthermore, Clint considers only a finite number of clauses because there are only a finite number of predicates and for each predicate a finite number of clauses. Also, clauses that are known to be incorrect are never added to the knowledge base.

(2): each pass through the while-loop results in either :

- leaving the while-loop in the case that handle_constraint is called or a negative example $p(u_1, ..., u_n)$ for p is acquired that is covered by the candidate clause.
- Or, an applicable constraint becomes not applicable. Because there are only a finite number of constraints, the while loop cannot loop infinitely.

□

The lemma allows to prove:

Theorem 5.3 *Any non-recursive predicate q will be error-free identified using the eager method from a definitely correct knowledge base KB and any constraint presentation of q w.r.t. KB if q is expressible in terms of the languages $L_1, ..., L_M$ and KB.*

Proof : This follows directly from theorem 5.1 and lemma 5.1. □

Theorem 5.4 *Any set of concepts C, expressible in one of the languages $L_1, ..., L_M$ and KB, will be postponed identified using the eager method from a definitely correct starting knowledge base KB and any constraint presentation of C w.r.t. KB.*

Proof : This follows directly from theorem 5.2 and lemma 5.1. □

5.6 Conclusions and Related Work.

Regarding related work, we can be short. To the best of my knowledge the use of constraints – in our sense – is new. There are however two approaches that are somehow related :

- the approaches to intentional knowledge base updating also use constraints to control the facts that are added to a knowledge base when effecting the addition or retraction of a fact from a knowledge base (see e.g. [Guessoum and Lloyd 90, Bry 90]). The relation between intentional knowledge base updating and incremental concept-learning has been further investigated in [De Raedt and Bruynooghe 90d], where we have also argued in favour of using constraints while learning concepts.

- The model-driven learning system Blip [Morik 89, Wrobel 89, Thieme 89, Emde 89] employs meta-knowledge and metameta-knowledge while learning.

Blip is a knowledge acquisition system that employs a model-driven learner to model the domain. It uses a special knowledge representation scheme that is based on a higher order logic. In Blip's framework one can represent meta-knowledge and metameta-knowledge. Blip learns by deriving rules. These rules are similar to definitions learned by Clint. In Blip they are represented by meta-facts. However, during the learning process, Blip uses also knowledge about the types of rules and about the relations among different types of rules. Knowledge about types of rules is represented at the meta-level and knowledge about the relation among them at the metameta-level by metameta-facts. For instance, a type of rule could be an isa relation, which can be implemented by `p(X)` ← `q(X)`. The knowledge about the relations between types of rules could e.g. be a property such as transitivity of the isa relation. This corresponds to a logical formula such as `(p(X) -> q(X))` ∧ `(q(X) -> r(X)) -> (p(X) -> r(X))`, where `p`, `q`, and `r` have to be considered as predicate variables. Blip uses such knowledge to avoid generating redundant rules (e.g. `p(X) -> r(X)` when the other two rules are already in the knowledge base). Furthermore, it employs other kinds of meta-knowledge to avoid generating inconsistent rules such as `not(p(X))` ← `q(X)` and `p(X)` ← `q(X)`.

As one can see, Blip uses meta-knowledge. However, the nature of the meta-knowledge and the way in which it is used, is different than in Clint. Blip's notion of meta-knowledge, i.e. knowledge about types of rules, corresponds to our notion of language-bias. With regard to metameta-knowledge about the relations among types of rules, there is however, no corresponding notion in Clint. Indeed, Clint does not consider relations among types of rules but rather relations among predicates. In this way, Blip's metameta-knowledge can be seen as encoding procedural knowledge and our notion of meta-knowledge as a form of declarative knowledge. This is also revealed by the way Blip uses its metameta-knowledge to avoid generating redundant, undesirable or inconsistent rules. In our context issues like inconsistency and redundancy are realized without (meta)meta-knowledge because of the logical framework. Clint will for instance never allow to generate inconsistent knowledge bases. Therefore, a large part of the metameta-knowledge (the ones that encode tautologies) in Blip is not needed by Clint. The remaining part cannot be described in Clint. However, it is also not clear whether Clint's meta-knowledge can be used in Blip. While it is clear that certain constraints such as `father(X,Y)` ∧ `mother(X,Y)` → `false` can be handled by Blip by adding them to the knowledge base as inference rules (`father(X,Y)` → `not(mother(X,Y))` and `mother(X,Y)` → `not(father(X,Y))`); it is not clear how to treat constraints such as `human(X)` ∧ `age(X,Y)` → `Y < 150` in Blip. The reason for this is that it is not appropriate to add such rules (or rewritings of such rules) to the knowlege base because this would result in many redundancies in the knowledge base and a knowledge base that is very hard to analyse and understand.

An additional difference is that the metameta-level of Blip is hard to understand for a user (cf. [Thieme 89]), whereas constraints in Clint are quite easy to grasp. So, even if constraints could be used in our way by Blip, it would be much harder to specify them or to understand their meaning.

As far as related work is concerned, we can conclude by saying that the meta-knowledge employed by Blip and Clint is quite different.

The use of constraints while learning concepts is a very interesting contribution to the field of concept-learning because it allows to augment the induction process with

knowledge about what has to be learned. Furthermore, the representation of constraints as clauses is easy to understand and to formulate. This contributes to the user-friendliness of the Clint system, as the user is provided with a powerful tool to supply the available (incomplete) knowledge to the learner. The learner will then take care that its knowledge base is consistent with this knowledge.

Chapter 6

Knowledge Acquisition.

6.1 Introduction.

An important class of knowledge in learning apprentices and knowledge based systems is the fact knowledge concerning objects. In terms of first order logic, the fact knowledge about an object is the set of all facts in the knowledge base that mention the object. In previous chapters, it was shown how general rules could be induced from examples by Clint. However, using those techniques alone, Clint, as presented in previous chapters, may run into problems when attempting to gather fact-knowledge. This is a weakness, which may sometimes lead to severe problems. Indeed, assume Clint does not know anything about the object `gunther`. Given a positive example `father(gunther, jef)` it cannot derive any justification in one of the languages presented in chapter 4, as this would require the facts `parent(gunther,jef)` and `male(gunther)` to be derivable from the knowledge base. Because fact knowledge is a vital part of the knowledge contained in knowledge bases, it is desirable to augment Clint with a component to acquire such knowledge. Such a component is the subject of this chapter.

The acquisition of fact knowledge is an important subtask in general knowledge acquisition tools as well as in learning systems. In learning systems fact knowledge is often a prerequisite for success. Indeed, many systems require that the domains of variables are completely known [Michalski 83, Quinlan 86]. Other approaches, like [De Raedt and Bruynooghe 89e, Wirth 88, Quinlan 90] cannot learn if they are not provided with fact knowledge. In learning apprentice systems such as Disciple [Kodratoff and Tecuci 87, Tecuci and Kodratoff 90], some knowledge about objects is acquired as a side-effect of the learning, while in other systems as Blip [Morik 89, Wrobel 88], knowledge about objects is derived from facts that are input by the user.

In this chapter a different approach is taken. When the user inputs a true fact containing an unknown object, an attempt is made to gather as much knowledge about the object as possible. To this purpose, the knowledge base is examined and interesting membership and existential queries [Angluin 87, Shapiro 83] are formulated to the user. The answers to these questions allow to update the knowledge base. Two main methods are employed: analysing existing definitions (Horn-clauses) for the predicate and using types or sorts of predicates. The methods are completely implemented in Prolog (see [Feyaerts 89]). They can easily be integrated in other systems like e.g. Disciple [Kodratoff and Tecuci 87, Tecuci and Kodratoff 90], Clint ([De Raedt and Bruynooghe 89e] and chapters 3 and 4), LFP [Wirth 88], Cigol [Muggleton and Buntine 88] and Blip [Morik 89, Wrobel

88]. It is feasible to regard the approach as a form of learning by being told, and in this sense, there is some resemblance with e.g. [Haas and Hendrix 83].

This chapter is organized as follows: in section 6.2, we define the basic notions and provide a problem specification; in section 6.3, the algorithms to acquire fact knowledge are presented while in section 6.4, an example of the algorithms at work is shown; in section 6.5, it is indicated how these algorithms can be integrated in the Clint system and finally, in section 6.6 we conclude and touch briefly upon related work. Before going through sections 6.2 and/or 6.3, it may be useful to have a look at the example session in section 6.4.

6.2 Problem-specification.

First, we will see why Clint may run into problems when attempting to acquire fact knowledge and then we will present a problem-specification.

6.2.1 Why Clint might run into difficulties.

In the previous chapters, we have always assumed that the clauses in L are linked and range-restricted. Furthermore, in chapter 4, we defined several language-series, which normally do not allow for constants or functors. This means that when Clint receives a positive example f containing an unknown object o, it will have difficulties in finding justifications. The reason is that it will look for the true properties in which o is involved. Under the sketched circumstances it can only use the facts that are in the knowledge base and contain variables (e.g. p(X,a) ←). Generally speaking, this may not be sufficient to find a justification (and generalizations of that justification) that fulfills the range-restriction and linkedness requirements. Imagine for instance that all facts in the knowledge base are ground. In such cases Clint will never be able to learn a correct clause for the fact f. This problem is due to the range-restriction, the linkedness condition and the forbidding of the constants. Indeed, without these assumptions, the fact f would itself be a clause that is expressible in the language. This clause would always be consistent if the example-set is. One may want to add the fact f or a generalization of the fact f to the knowledge base. This would be possible from a theoretical point of view if we are prepared to use other languages and to lift the range-restriction and linkedness conditions. Using other languages in the framework of Clint is possible (c.f. chapter 4) and we also stated in section 3.4 that the range-restriction and linkedness condition are strictly speaking not necessary. One solution would e.g. be to add a new language L_{M+1} to our series, in which there are only facts. However, these modifications would interfere with the other techniques. E.g. adding language L_{M+1} to our series would imply that we never postpone the learning of examples as the examples itself could be added to the knowledge base. Yet the postponing of learning is in many situations very useful because it results in the learning of general rules instead of asserting individual facts. As a consequence, the approach – although theoretically possible – is not desirable in practice. There is also a second reason why the modified approach should not be followed. Usually, the fact-knowledge about an object consists of more than one fact. Following the L_{M+1} approach, the user would have to supply one example to learn one fact in the fact-knowledge. This is neither opportunistic nor user-friendly. Upon the acquisition of a fact containing a new

object, it is much more desirable to help the user identify the other facts mentioning the object. This is precisely what we shall do in the rest of this chapter. The integration of the techniques described in this chapter with Clint will be discussed in section 6.5. It will be shown that the techniques do not interfere with the postponing mechanism.

6.2.2 Problem-specification.

In this chapter, the Datalog framework introduced in section 3.7 will be used. No functors will be considered. It has already been argued that this framework is in most cases sufficient for the learning apprentice context. Recall from chapter 3, section 3.2 that there is a distinction between basic and learned predicates. Here, a further distinction is made between *fact*, *rule* and *mixed* predicates.

Definition 6.1 *A fact predicate is defined by a set of ground facts.*

Definition 6.2 *A rule predicate is defined by a set of non-unit Horn-Clauses.*

Definition 6.3 *A mixed predicate is a predicate that is neither a fact nor a rule predicate.*

It will be assumed that the system has access to an oracle that is willing to answer membership (see definition 3.22) and existential questions.

Definition 6.4 *(see [Shapiro 83, Angluin 87])*
An existential question $p(x_1, ..., x_n)$ asks the oracle about the truthvalue of a non-ground atom $p(x_1, ..., x_n)$ in the intended interpretation τ, where p is a predicate and at least one of the x_i is not a variable. The oracle must return some substitutions θ such that $\tau(p(x_1, ..., x_n)\theta) = t$, where t is a truthvalue.

Furthermore, all answers to queries, all examples and all constraints will be added to the constraint theory as before.

By now, the problem can be specified more formally:

Problem Statement 6.1 *The update problem.*

- **Given:**
  - a knowledge base KB of learned and basic predicate definitions (see definitions 3.20 and 3.21).
  - a set of constraints C, $\forall i \in C : \tau_2(i) = \text{true}$
  - a ground fact $p(o_1, ..., o_n)$ such that $\varphi_2(KB, p(o_1, ..., o_n)) = \text{false}$ and at least one of the o_i is a constant, not occurring in the knowledge base KB, i.e. o_i does not occur in the Herbrand-domain of KB.
  - an oracle willing to answer existential and membership questions.
- **Find:** a set S of facts, true in the intended interpretation τ_2, such that $\varphi_2(KB \cup S, p(o_1, ..., o_n)) = \text{true}$ and $KB \cup S$ satisfies all constraints in C; the set S will be called the update.

In this problem-specification, there is a fact $p(o_1, ..., o_n)$ that is not implied by the knowledge base KB, but which should. Furthermore, there is a set of constraints, true in the intended interpretation. These constraints include the facts that have been obtained from previous questions to the oracle. Indeed each answer to a membership- or existential query corresponds to a constraint of the form $(q \rightarrow \texttt{false})$ or $(\texttt{true} \rightarrow q)$ in the two-valued framework. (In the multi-valued framework this will become a tuple of constraints, see chapter 7.) While presenting the procedures to find updates, it will be assumed that only constraints of this form are used. Later, in section 6.5, when integrating the techniques in Clint, the framework of chapter 5 will be used. This will result in the correct treatment of more general constraints.

The aim of the update problem is to assert a set of true facts S in the knowledge base such that $p(o_1, ..., o_n)$ is implied by the knowledge base $KB \cup S$ and $KB \cup S$ satisfies the integrity constraint theory. This problem will be referred to as the *update* problem; in particular, if the predicate p is a rule predicate it will be referred to as the *rule update* problem; if p is a fact predicate, the problem becomes the *fact update* problem. These notions are illustrated in example 6.1.

In example 6.1, it is rather clear which questions should be posed to the oracle as `drives` is a rule predicate. However, suppose a new type of `car` (a fact predicate) has to be added to the knowledge base. Then it would be harder to decide which questions to generate. Of course, one could assert the fact itself. However, this is not desirable as the amount of knowledge acquired in this way is minimal. Two approaches that generate *interesting* questions to the oracle, will be presented. In these approaches, an attempt is made to maximize the acquired knowledge while keeping the number of questions to the user acceptable.

Note that the above sketched problem is a *machine learning* problem. It is a form of learning by being told [Carbonell, Michalski and Mitchell 83], where new information is to be transformed before it is added to the knowledge base. It is also a form of induction: in example 6.1, one could imagine that the definition of `vehicle` was inductively learned by a concept-learner as e.g. Clint. By asserting `car(citroen)` it becomes possible to induce `vehicle(citroen)`.[1] Furthermore, it is essential that inductive concept-learners address the sketched problem. Without addressing it, they are often unable to learn. Consider e.g. the concept-learner Clint: as outlined in section 6.2.1 it may run into problems when attempting to process the true fact (positive example) `drives(luc,citroen)`. Indeed, using its normal languages it will fail to learn a correct clause for `drives(luc,citroen)` because it will find no justification for `drives` by looking for the relations that hold among the objects (`luc` and `citroen`). In this situation, Clint cannot make the necessary modifications to its knowledge base to make the fact `drives(luc,citroen)` implied. In section 6.5, we will see how Clint can be extended to cope with such situations. Similar situations arise in other concept-learners, as e.g. LFP [Wirth 88], Blip [Morik 89, Wrobel 88] and Disciple [Kodratoff and Tecuci 87, Tecuci and Kodratoff 90]. LFP suffers from the same problem as Clint, whereas Blip and Disciple take the minimal approach, i.e. they would assert the facts containing the new objects. However, in both cases, an opportunity to gather additional knowledge about the involved objects is missed.

[1] One might argue that the inference of `vehicle(citroen)` is deduction rather than induction because the fact follows from the clauses for `vehicle` and the fact `car(citroen)`. This argument is false, because the clauses for `vehicle` are derived by induction. As a consequence, the inference of `vehicle(citroen)` is an inductive (falsifiable) conclusion.

Example 6.1: Illustrating the problem-specification.

Assume the knowledge base contains the following definitions:

```
owns(yves, peugeot) ←
owns(katharina, mercedes) ←
owns(maurice, renault) ←
owns(luc, plume) ←

car(peugeot) ←
car(mercedes) ←
car(renault) ←

bicycle(plume) ←

vehicle(X) ← car(X)
vehicle(X) ← bicycle(X)

drives(X,Y) ← owns(X,Y) ∧ car(Y) ∧ license(X)
drives(X,Y) ← owns(X,Y) ∧ bicycle(Y)

license(katharina) ←
license(yves) ←
license(maurice) ←
```

In this knowledge base, the predicates `drives` and `vehicle` are rule predicates, the others are fact predicates. Assume that all predicates are learned and that the aim is to make the fact `drives(luc, citroen)` true. There are several possibilities to realize this:

S_1 = { `drives(luc, citroen)` }
S_2 = { `owns(luc, citroen), license(luc), car(citroen)` }
S_3 = { `owns(luc, citroen), bicycle(citroen)` }

Given the available information, it is only known that S_2 is correct in the intended interpretation. However, if one of the other possible updates is correct, it is not interesting to assert S_1 in the knowledge base. The reason is very obvious: by asserting S_1, much potential information would be lost. Indeed, it would no longer be possible to derive e.g. `vehicle(citroen)` from the knowledge base. Furthermore, the structure of the knowledge base would become more clumsy: the rule predicate `drives` would become mixed. In order to be sure that one of the other possible updates is correct, the oracle has to be queried for the truth-value of the facts in the updates S_2 and S_3.

6.3 Updating the knowledge base.

In this section, two algorithms to handle updates are presented. First, rule updates are treated and then fact updates.

6.3.1 Updating rule predicates.

The algorithm to handle rule updates relies on the following notion of dependency:

Definition 6.5 *A literal l in a goal g of a learned predicate (cf. problem statement 6.1), depends on a constant c iff*

1. *c occurs in l or*
2. *there is a variable v occurring in l, that also occurs in a literal l' of g, such that l' depends on the constant c*

Clearly, if l does not depend on a constant c, it is independent of c.

Definition 6.6 *The set of subgoals of a goal g that are independent of new objects in the goal is denoted as $I(g)$, whereas the set of subgoals that depend on new objects is $D(g)$. A new object is an object that does not yet occur in the knowledge base, or one that is currently being added to the knowledge base.*

This notion of dependency is essential for the approach because our technique for rule updates is based on the following assumption:

Assumption 6.1 *Algorithm 6.1 can be used to determine whether a query q fails or succeeds. To determine the success or the failure call* solve(q) .

This assumption means that for the independent literals, the knowledge base is supposed to be correct. In other words, for these literals, the intended interpretation is the same as the one according to the knowledge base. Therefore independent subgoals are solved by querying the knowledge base and dependent ones by querying the user. The algorithm solve works as follows : it adds the original goal to its list of goals; then it repeatedly deletes a goal from the list and attempts to solve a subgoal of the goal. In order to solve a goal, the algorithm splits the goal in a dependent and an independent part. The independent part is run on the knowledge base. The remaining subgoal is then added to the list of goals. If the independent part of the goal is empty, then the algorithm will query the oracle for the truthvalue of a literal containing a new object. The remaining subgoal is — of course — also added to the list of goals. The process continues until the list of goals contains the empty goal and returns success or until the list of goals is empty and returns failure. Example 6.2 makes the involved notions clearer.

Algorithm 6.2 summarizes how to realize rule updates under assumption 6.1. The algorithm starts from a ground fact $p(o_1, ..., o_n)$ for a rule predicate p that contains a new object and that is not implied by the knowledge base. The algorithm will assert facts making $p(o_1, ..., o_n)$ implied. During the session, the system may also acquire knowledge about other objects and predicates. Non-ground facts will be handled by the procedure handle_fact (see below).

Algorithm 6.1: Solving a goal.

```
procedure solve( ← g1 ∧ ... ∧ gn: goal ): returns succes or failure
    L = { ← g1 ∧ ... ∧ gn }
    repeat
        delete a goal g from L
        if I(g) ≠ ∅
        then for all θ such that φ2(KB, I(g)θ) = true do
                add D(g)θ to L
             endfor
        else if D(g) ≠ ∅
             then delete a literal l from g such that l ∈ D(g) and
                      l contains a new object
                  ask τ(l) to the oracle
                  for all θ returned by the oracle such that τ2(lθ) = true do
                      add gθ to L
                  endfor
             endif
        endif
    until L contains the empty goal or L = ∅
    if L = ∅
    then return failure
    else return success
    endif
endproc
```

Example 6.2: Illustrating dependency.

Suppose the goal is ← `person(jeff)` ∧ `has_license(jeff)` ∧ `works_for(jeff,C)` ∧ `air_line_company(C)` ∧ `owner(C,concorde-7)` ∧ `plane(concorde-7)` where `concorde-7` is a new object and all predicates are learned. The truthvalue of the goal can be verified by first checking whether the independent literals are implied by the knowledge base: `person(jeff)` and `license(jeff)`; if they are, the existential question corresponding to `owner(C,concorde-7)` can be posed to the oracle; in case the oracle returns a true substitution θ = { `C = air_france` } , where `air_france` is not a new object, then the answer to the query follows from the query ← `air_line_company(air_france)` ∧ `works_for(jeff,air_france)` to the knowledge base, because `air_line_company(air_france)` and `works_for(jeff, air_france)` are independent.

In algorithm 6.2, L contains the list of goals g fulfilling the property that if g is true in the intended interpretation ($\tau_2(g) = \text{true}$), there is a clause in the knowledge base that correctly covers the fact $p(o_1, ..., o_n)$.

Definition 6.7 *A clause c correctly covers a fact f if there is a substitution θ such that $head(c)\theta = f$ and $\tau_2(body(c)\theta) = true$.*

Notice the difference between *correctly covers* and *covers* (definition 3.13). Covers means implication in the knowledge base; whereas correctly covers means implication in the intended interpretation !

The procedure handle_rule first initializes the list L with the goals corresponding to the instantiated bodies of clauses in the knowledge base.

In the repeat-loop, the most interesting goal (see section 6.3.3 and below) is deleted from L. The inner part of the repeat loop verifies whether the goal contributes to the aim of correctly covering the fact. Clearly, if one of the literals in such a goal is false in the intended interpretation (this can be derived from constraints of the form $p \rightarrow$ `false`), then the goal cannot contribute to this aim. As in example 6.2 and algorithm 6.1, to determine the truthvalue of the goal, the independent part of the goal is solved by querying the knowledge base. If there are no independent subgoals in the goal, a dependent literal that contains a new object, is used to generate a question to the oracle. If there are no independent subgoals in a goal and the goal is not the empty goal, there is always a literal containing a new object. The solutions θ making the subgoals true are propagated to the rest of the original goal, which is added to L. (Notice that in case there is more than one true substitution for the selected subgoal(s), more than one goal will be added to L.)

The heuristic to delete the most interesting goal from L will always select goals according to the clauses from which they are derived. In this way, the heuristic considers the goal corresponding to the body of one clause, and continues to work on subgoals of this clause until a solution has been found or it is certain that the clause does not cover the fact. At that point, a new clause will be selected according to the heuristic which will be presented in section 6.3.3 . So, within one clause, a depth-first strategy is realized and in between different clauses, a best-first strategy.

Of course, if the repeat-loop stops on the condition $L = \emptyset$, it means that there is no clause in the knowledge base that correctly covers the fact. Therefore, the fact has to be asserted in the knowledge base. This is realized by invoking handle_fact (algorithm 6.3, see section 6.3.2) on this fact. On the other hand, if the loop terminates because the empty goal is found, care must be taken that the literals contributing to the coverage of the fact are implied by the knowledge base. This explains why the procedure handle_fact or handle_rule is called for these facts.

Non-ground true facts will be treated by the procedure handle_fact. The reason is that those facts have to be valid for all instances of the fact. When using the procedure handle_rule, one would make such facts valid for some instances, but not necessarily for all instances. Therefore non-ground facts have to be asserted in the knowledge base. This is realized by calling handle_fact, which will also generate other interesting questions.

Algorithm 6.2: Realizing rule updates.

```
procedure handle_rule( p(o_1,...,o_n): ground true fact containing a new object o_i)
    L = { body(c)θ | c is a clause for p in KB and
          most general unifier(head(c), p(o_1,...,o_n)) = θ}
    repeat
        delete the most interesting goal g from L
        if there is no literal t in g for which there is a constraint (f → false)∈ C
          and a substitution ρ such that t = fρ
        then if I(g) ≠ ∅
             then for all θ such that φ_2(KB, I(g)θ) = true do
                      add D(g)θ to L
                  endfor
             else if D(g) ≠ ∅
                  then delete a literal l from g such that l ∈ D(g)
                                and l contains a new object
                       ask τ(l) to the oracle
                       for all θ returned by the oracle such that τ_2(lθ) = true do
                           add gθ to L
                       endfor
                  endif
             endif
        endif
    until L = ∅ or L contains the empty goal
    if L =∅
    then call handle_fact(p(o_1,...,o_n))
    endif
    for all oracle questions τ(q) in the repeat loop such that τ_2(qθ) = true do
        if φ_2(KB, qθ) = false
        then if q is a rule or mixed predicate and qθ is ground
             then call handle_rule( qθ )
             else call handle_fact( qθ )
             endif
        endif
    endfor
endproc
```

6.3.2 Updating Fact Predicates.

First, the very important notion of a type is introduced. Types are treated as in Blip [Kietz 88, Morik 89, Wrobel 89].

Definition 6.8 *The type T_p^i, the type corresponding to the i-th argument position of the predicate p is defined as follows:*
$T_p^i = \{ o \text{ is an object} \mid \exists x_1, ..., x_{i-1}, x_{i+1}, ..., x_n \text{ for which } \varphi_2(KB, p(x_1, ..., x_{i-1}, o, x_{i+1}, ..., x_n)) = true \}$.

So, the type T_p^i is the set of all objects occurring at the i-th position in a fact f for p that is implied by the knowledge base. Because Datalog is used, the following assumptions holds :

Assumption 6.2 *Type assumption.*
All types are finite !

Although Datalog knowledge bases may contain universally quantified facts such as `is_allowed_to_drive(X,plume)`, one can always compute a finite type for such predicates. The reason is that there is always only a finite set of objects $T_{universal}$ in the knowledge base. If such universally quantified variables are encountered, one can assign the set $T_{universal}$ to the type, e.g. $T^1_{is_allowed_to_drive} = T_{universal}$.

Since types are sets of objects, they are partially ordered by the subset relation. Hence, they can be represented in a typegraph with at the top the universal type, containing all possible objects and at the bottom the empty type, containing no objects. In the algorithms given below, it will be assumed that types are represented in such a graphstructure. This assumption is also made in Blip. The explicit representation of types from a knowledge base can be automatically computed using the techniques of [Kietz 88]. Furthermore, as in Blip's approach, the non-empty intersection types are explicitly represented in the graph. A typegraph is shown in figure 6.1.

Definition 6.9 *An intersection type T is a type such that $T = T_1 \cap T_2$ where T_1 and T_2 are either intersection types or types corresponding to an argument position of a predicate.*

Definition 6.10 *The typegraph of a knowledge base represents all types corresponding to an argument position of a predicate, the empty type, the universal type and all resulting intersection types.*

The advantage of explicitly representing intersection types is that it is easy to determine the supertype(s) and subtype(s) of a certain type and to decide whether two types are equal, disjoint or overlapping.

Definition 6.11 *A supertype S of a type T w.r.t. a certain typegraph G is a type occurring in G such that S is a proper superset of T.*

Definition 6.12 *A subtype S of a type T w.r.t. a certain typegraph G is a type occurring in G such that S is a proper subset of T.*

Definition 6.13 *A type T corresponds to a predicate q if $\exists i : T = T_q^i$.*

Example 6.3: Illustrating types.

The types for the knowledge base of example 6.1 are the following:

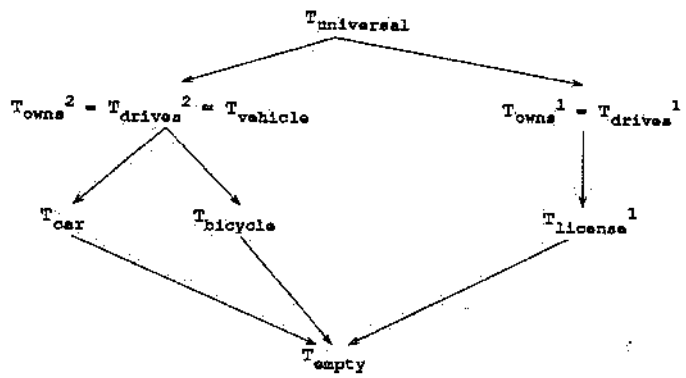
T_{owns}^{1} = { yves, katharina, maurice, luc }
T_{owns}^{2} = { peugeot, mercedes, renault, plume }
T_{car}^{1} = { peugeot, mercedes, renault }
$T_{bicycle}^{1}$ = { plume }
$T_{vehicle}^{1} = T_{owns}^{2}$
$T_{drives}^{1} = T_{owns}^{1}$
$T_{drives}^{2} = T_{owns}^{2}$
$T_{license}^{1}$ = { yves, katharina, maurice }

Furthermore, there is a universal and an empty type:

$T_{universal}$ = { yves, katharina, maurice, luc, peugeot, mercedes, renault, plume }
T_{empty} = {}

There are also some intersection types. However, these do not give rise to additional types. Indeed, all intersection types as e.g. { plume } already occur as a regular type. The resulting typegraph is shown in figure 6.1.

Figure 6.1: The typegraph of the drives knowledge base.



In example 6.3, types are illustrated
The main assumption underlying algorithm 6.3 is the following one :

Assumption 6.3 *If a new object o is inserted in type T of the typegraph, there is a high probability that it should also be inserted in the supertypes of T.*

Notice that this assumption is a plausible one. It will be satisfied when the typegraph is stable enough to represent the true relations among the types of the domain. Clearly, the more stable the typegraph, the more clever questions algorithm 6.3 will generate. The assumption explains why algorithm 6.3 queries the oracle for the truthvalue of $q(x_1,...,x_{j-1},o,x_{j+1},...,x_n)$ if o has to be inserted in a type $T \subset T_q^j$. Algorithm 6.3 is more complicated than this because different kinds of predicates and types have to be taken into account.

Algorithm 6.3: Realizing Fact Updates.

procedure handle_fact($p(o_1,...,o_n)$: true fact containing a new object o_j)
  assert $p(o_1,...,o_n)$
  for all positions i for which o_i is a new object in $p(o_1,...,o_n)$ **do**
    call handle_supertypes(T_p^i, o_i)
    $L := \{\ T \mid T$ is a maximal subtype of T_p^i in the typegraph $\}$
    $Fa := T_p^i$; *Fa always refers to the father type of the types in L and $o_i \in Fa$*
    repeat
      delete the most interesting type T from L
      case $T = T_q^j$ for a learned predicate q **do**
        call insert_in_type(q,j,o_i)
      case T is an intersection type and there is a parent type T_q^j of T
        such that q is a learned predicate and $T_q^j \cap Fa = T$ **do**
        call insert_in_type(q,j,o_i)
      case otherwise **do**
        $L := \{\ T' \mid T'$ is a maximal subtype of T in the typegraph $\}$
        $Fa := T$
      endcase
    until $L = \emptyset$
    Fa is the smallest type such that $o_i \in Fa$
    for all types $T_q^j \supset Fa$, for which q is learned and (mixed or rule) **do**
      if $\forall\theta : \varphi_2(KB, q(x_1,...,x_{j-1},o_i,x_{j+1},x_k)\theta) =$ false
      then call query($q(x_1,...,x_{j-1},o_i,x_{j+1},x_k)$)
      endif
    endfor
  endfor
  Reconfigure the typegraph
endproc

Algorithm 6.4: Procedures used by handle_fact.

```
procedure handle_supertypes( T: type, o: new object)
      for all supertypes T_q^j of T that correspond to a learned fact predicate q
          and that not have been handled yet do
            call query(q(x_1,...,x_{j-1},o_i,x_{j+1},x_k))
      endfor
endproc

procedure query( q(x_1,...,x_{j-1},o_i,x_{j+1},x_k))
      ask τ(q(x_1,...,x_{j-1},o_i,x_{j+1},x_k)) to the oracle
      for all substitutions θ, returned by the oracle,
          such that τ_2(q(x_1,...,x_{j-1},o_i,x_{j+1},x_k)θ) = true do
            if q is a rule predicate and q(x_1,...,x_{j-1},o_i,x_{j+1},x_k)θ is ground
            then call handle_rule( q(x_1,...,x_{j-1},o_i,x_{j+1},x_k)θ)
            else assert q(x_1,...,x_{j-1},o_i,x_{j+1},x_k)θ
            endif
      endfor
endproc

procedure insert_in_type(q: learned predicate, j: argument position, o_i: object)
      if ( ¬∃θ : φ_2(KB, q(x_1,...,x_{j-1},o_i,x_{j+1},x_k)θ) = true ) and there is no constraint
        (f → false) ∈ C such that ∃ρ : fρ = q(x_1,...,x_{j-1},o_i,x_{j+1},x_k)ρ
      then call query( q(x_1,...,x_{j-1},o_i,x_{j+1},x_k))
      endif
      if ∃θ : φ_2(KB, q(x_1,...,x_{j-1},o_i,x_{j+1},x_k)θ) = true
      then call handle_supertypes(T_q^j, o_i)
            L := { T | T is a maximal subtype of T_q^j in the typegraph}
            Fa := T_q^j
      endif
endproc
```

Algorithm 6.3 works as follows: for each new object o_i occurring in the given fact, the type where the object has to be inserted is located. Following the assumption, the algorithm verifies whether the object also has to be inserted in its supertypes. In order to do so, the procedure handle_supertypes (see algorithm 6.4) is invoked. This procedure generates a question for all supertypes that contain a fact predicate as to decide whether the object should be inserted in the supertype or not. The questions are handled by the procedure query, which calls the procedure handle_rule or asserts the facts if the answer to the question is positive. The procedure handle_supertypes does not ask questions about mixed or rule predicates. The reason is very simple: by asserting facts for fact predicates in the knowledge base, the mixed and rule predicates may also be affected. Therefore, algorithm 6.3 and procedure handle_supertypes first modify the fact predicates and will only later verify whether questions about rule and/or mixed predicates are needed. This verification takes place in the inner for-loop of algorithm 6.3.

After treating the supertypes of the type where the object has to be inserted, algorithm 6.3 verifies whether the object has to be inserted in one of the subtypes of the type. Testing whether an object has to be inserted in a subtype is done by calling the procedure insert_in_type. This procedure first tries to deduce whether the object belongs to the type or not. If it cannot deduce this, it will generate a query to the oracle in order to find out the answer anyway. When the object has to be inserted in a certain subtype, the procedure insert_in_type will also attempt to insert the object in the supertypes of the subtype and will also consider the subtypes of the current type. This is realized by calling handle_supertypes and by modifying the current list of subtypes L.

In order to determine whether an object has to be inserted in one of the subtypes of a current type, there are three different cases to consider (the three cases in the repeat-loop of algorithm 6.3).

- the subtype corresponds to the argument position of a learned predicate : in this case, the system will verify whether the object should be inserted in the subtype by calling insert_in_type
- the subtype is an intersection type : the system will verify whether there is a parent type of the intersection type that corresponds to a learned predicate and that is — together with the parent type already considered — responsible for the intersection type. If there is such a parent type, the system will verify whether the object has to be inserted in the parent type. If the object has to be inserted in the parent type, it will also belong to the intersection type.
- in the other case : the system is unable to generate relevant questions about the currently considered subtype. Therefore it will go one step deeper into the type-graph.

When the object cannot be inserted in any of the maximal subtypes of the current type Fa, the repeat loop terminates and the algorithm examines the relevant learned rule predicates.

6.3.3 A heuristic to select the most interesting candidates first.

To be able to select the most interesting candidates first in the above algorithms, the system keeps track of a set $analog(n)$ of analogous objects o for each new object n.

Furthermore, for each object $o \in analog(n)$ it computes a score, consisting of two numbers: the number of tests t on analogy in which the analogous object was involved and the number of times st a test has been successful. For each answer to an oracle question (the initial new term is treated as an answer to an oracle question) $q(x_1, ..., x_{j-1}, n, x_{j+1}, ..., x_k)$ involving the new object n, the set $analog(n)$ is modified as follows:

If the answer was 'yes' for a substitution θ, then (1) the set $analog(n)$ is augmented with all objects o, that did not yet occur on the set and for which $\varphi_2(KB, q(x_1, ..., x_{j-1}, o, x_{j+1}, ..., x_k)\theta) = true$ holds. The number of tests t for these objects o is initialized to 0. (2) The number of successful tests st of all objects n on the new set satisfying $\varphi_2(KB, q(x_1, ..., x_{j-1}, n, x_{j+1}, ..., x_k)\theta) = \text{true}$, is increased by 1. (3) The number of tests t of all objects on the set is increased by 1.

If the oracle's answer was 'no', then the number of tests t of all members of the set is increased by 1 and the number of successful tests st of all objects o for which there is no θ such that $\varphi_2(KB, q(x_1, ..., x_{j-1}, o, x_{j+1}, ..., x_k)\theta) = \text{true}$, is increased by 1.

By taking subsets of the set of analogous objects, it becomes possible to detect the most interesting candidate. In order to do so, one takes a subset corresponding to each candidate and calculates the average ratio st/t for each considered candidate. The most interesting candidate is then the one whose average ratio is maximal.

In algorithm 6.2, the heuristic can be used to choose among the clauses. This is done by associating to each clause the subset of objects from $analog(n)$ that are provable using that clause.

For the selection of the most interesting subtype in algorithm 6.3, the same method is used. The subset is then the intersection of the set $analog(n)$ with the type itself.

The heuristic works very well in practice. Its only drawback is that when there are many elements in the set, it requires much computation. Of course, the number of elements in the set could be kept within certain bounds, by storing only a representative sample of the analogous objects.

6.3.4 Properties of the algorithms.

The procedures handle_fact and handle_rule have one important property:

Theorem 6.1 *After calling* handle_fact(f) *or* handle_rule(f) *the knowledge base implies* f, *i.e.* $\varphi_2(KB, f) = true$.

Proof : For handle_fact, the proof is trivial. For handle_rule, the procedure derives in fact a proof for the fact f, and takes care that all facts that occur as a leaf in the proof will be asserted. This will be realized by calling handle_fact. If the leaves are derivable, we also have that the original fact f is provable, as all clauses needed to prove it are in the knowledge base. □

6.4 An example session with the system.

The algorithms specified in section 6.3, are completely implemented in Prolog. Below, a trace of a session with the implemented system is shown. Part of the knowledge base used in the session is shown in example 6.4 and part of the typegraph is shown in figure 6.2. The session is shown in example 6.5.

Example 6.4: Part of the knowledge base used in the session.

```
is_allowed_to_fly(X,Y) ← person(X) ∧ owner(X,Y) ∧ has_license(X)
      ∧ plane(Y)

is_allowed_to_fly(X,Y) ← person(X) ∧ has_license(X) ∧
      works_for(X,C) ∧ air_line_company(C) ∧
      owner(C,Y) ∧ plane(Y)

is_allowed_to_fly(X,Y) ← person(X) ∧ works_for(X,C) ∧
      air_force(C) ∧ owner(C,Y) ∧ military_plane(Y)

person(X) ← man(X)

person(X) ← woman(X)

plane(X) ← passengers_plane(X)

plane(X) ← military_plane(X)

plane(X) ← carrier(X)
```

The rest of the knowledge is specified by ground facts.

Example 6.5: An example session with the knowledge acquisition component

User input is in **bold** and systems output is typewritten.

```
?-learn.
Next example ?  @ is_allowed_to_fly(john,concorde).
Classification ?  @ true.
```

The user inputs a new fact, it contains two unknown objects: john and concorde. Since, is_allowed_to_fly is a rule predicate, procedure handle_rule is invoked. The analogy heuristic identifies the second clause as the most interesting one, and the system proceeds.

```
Is person(john) true ?@ yes.
```

As person is a rule predicate, it will be examined later

```
Is has_license(john) true ?  @ yes.
has_license(john) asserted ...
has_license is a fact predicate
Is plane(concorde) true ?  @ yes.
```

As plane is a rule predicate, it will be examined later

```
Is works_for(john,X) true for some X ?@ X = air_france.
air_france is a known object.
works_for(john, air_france) asserted ...
works_for is a fact predicate
Are there other X for which works_for(john,X) holds ?  @ no.
Is owner(air_france,concorde) true ?  @ yes.
owner(air_france,concorde) asserted ...
owner is a fact predicate
```

At this point, the system only needs to determine the truthvalue of air_line_company(air_france) in order to derive that the second clause correctly covers the original fact. However, as air_france is a known object, the oracle is not questioned but the knowledge base is queried. Fortunately, the fact is implied by the knowledge base, so the clause correctly covers the original fact. Now, the system proceeds with the new facts for person and plane. These two predicates are rule predicates, whereas man and passengers_plane are fact predicates.

```
   Is man(john) true ?  @ yes.
man(john) asserted ...
```

Because man is a fact predicate, algorithm 6.3 is invoked on man(john). However, as john belongs to all supertypes of $T_{\mathtt{man}}$ ($T_{\mathtt{person}}$) and to all other possible intersection

types in the typegraph ($T^1_{\texttt{is_allowed_to_fly}}$, $T^1_{\texttt{has_license}}$ and $T^1_{\texttt{works_for}}$), no further questions about john are generated. The system continues with `plane(concorde)`

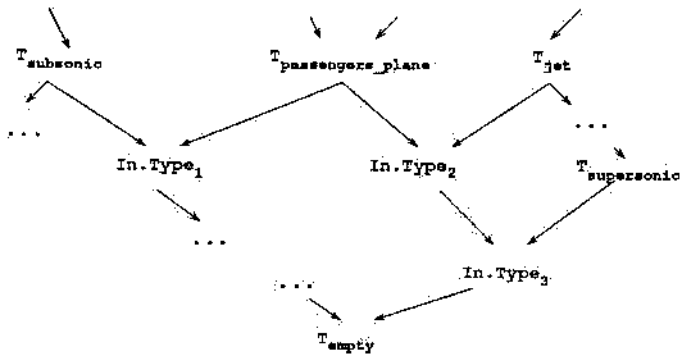
```
Is passenger_plane(concorde) true ?  @ yes.
passenger_plane(concorde) asserted ...
```

As `passenger_plane` has an intersection type with `subsonic`, the system asks:

```
Is subsonic(concorde) true ?
@ no.
As passengers_plane has also an intersection type with jet,it asks:
Is jet(concorde) true ?  @ yes.
jet(concorde) asserted ...
Because supersonic is a subtype of jet:
Is supersonic(concorde) true ? @ yes.
supersonic(concorde) asserted ...
```

Now, the last step of the algorithm is executed: the system checks that `concorde` belongs to all relevant supertypes. Since this is the case, no further questions are generated to the user.

Figure 6.2: The typegraph of the planes knowledge base.



6.5 Integrating the methods in Clint.

In this section, it is shown how the knowledge acquisition procedures can be integrated in Clint. Of course, it is also possible to integrate them into other systems such as LFP [Wirth 88], Disciple [Tecuci and Kodratoff 90, Kodratoff and Tecuci 87] and Blip [Morik 89, Wrobel 88].

The integration of the methods in Clint is rather straightforward. One only has to replace the call to handle_not_covered in the procedure handle_constraint (algorithm 5.1) to the call to handle_ka_not_covered. This procedure then calls handle_not_covered when the incoming constraint is an uncovered fact that does not contain new objects. The resulting procedures are shown in algorithms 6.5 and 6.6.

There is only one design decision that has to be made for the integrated system: when is the incoming fact treated by Clint and when by the algorithms specified in this chapter. The most plausible approach is to let the knowledge acquisition procedures handle the fact when the fact contains new objects and to let Clint handle the other facts. This decision also assures that the use of postponing will not interfere with the knowledge acquisition component. When the incoming example does not contain new objects, it is treated by Clint. Hence, this may result in postponing its treatment.

If the knowledge base contains many non-ground facts such as `is_allowed_to_drive(X, eddy_merckx_bike)`, it is not always necessary to invoke the knowledge acquisition procedure (see also section 6.2.1). A strategy that might be followed then is to invoke the knowledge acquisition procedures only when the incoming fact concerns a fact predicate. However, I believe that this should usually not be done as most applications do not require many non-ground facts.

Obviously the integrated algorithm retains its theoretical properties. Furthermore, the integration algorithm also takes care of the influences that modifications of one predicate may have on other predicates and constraints. These effects are treated as before (see

chapters 5 and 3).

6.6 Conclusions and Related Work.

The work presented in this chapter was done in collaboration with Johan Feyaerts [Feyaerts 89]. It contains an adapted version of [De Raedt, Feyaerts and Bruynooghe 91].

The presented work is related to work in learning apprentices such as Disciple [Tecuci and Kodratoff 90, Kodratoff and Tecuci 87] and Blip [Morik 89, Wrobel 89], learning by being told [Haas and Hendrix 83] and knowledge base revision (e.g. [Tomasovic 88]).

The learning apprentices Disciple and Blip provide some facilities to the user to help her to input new facts. Basically these facilities are such that the system shows the user some situation and that the user is required to enter some new facts concerning this situation. In Disciple this occurs when the system is in its weak theory mode and part of an explanation is needed by the system. If the system is unable to generate this explanation the user is supposed to help the system by inputting some facts. In Blip the user inputs facts and gets immediate feedback about the consequences of the input facts. If the user notices that not all desired facts concerning a certain object are consequences she can enter additional facts.

The type formalism used in this paper is the same as in Blip [Kietz 88]. In Blip, the type graph is automatically modified when the user enters new facts. Also, if the type structure should be modified, the user is told so. In this way the user is provided with additional hints about possibly missing knowledge [Kietz 88].

[Haas and Hendrix 83] present an approach to interactively acquire knowledge. Their approach gathers knowledge in a very similar way to ours. However, there are also some important differences: they use a natural language interface, determine the structure of the type graph by asking questions to the user and use a rather specific knowledge representation formalism. It is not clear how to integrate their approach in concept-learning systems because of the use of a special formalism.

The approaches to knowledge base updating such as [Tomasovic 88] attempt to automatically find the transactions needed to accommodate new facts in the data base. However, they do not try to obtain as much knowledge as possible about the new objects.

As a conclusion, the presented technique is different from the above mentioned ones in the sense that it attempts to acquire in a systematic way as much knowledge as possible about new objects. Therefore it seems also very useful in the learning apprentice context. In this respect, it goes further than the existing approaches to knowledge base updating and learning apprentices.

Algorithm 6.5: Integration in Clint

```
procedure ka_clint
        postpone := ∅
        while the oracle wants to input constraints do
                read the next constraint c and add it to the constraint theory
                call handle_constraint(c)
                Reconfigure the typegraph
        endwhile
endproc

procedure handle_constraint(c : constraint)
        if c is activated
        then if ∃θ such that cθ is violated
                then while there is no literal l ∈ cθ such that τ2(l) ≠ φ2(KB,l) do
                                select a literal l' ∈ cθ such that τ2(l') is not known
                                ask the oracle for τ(l')
                        endwhile
                        select an example p(t1,...,tn) ∈ cθ
                          such that τ2(p(t1,...,tn)) ≠ φ2(KB,p(t1,...,tn))
                        if τ2(p(t1,...,tn)) = false
                        then call handle_covered(p(t1,...,tn))
                        else call handle_ka_not_covered(p(t1,...,tn))
                                this is the only change w.r.t. algorithm 5.1
                        endif
                        for all constraints c' that depend on p do
                                call handle_constraint(c')
                        endfor
                        if ∃e ∈ postpone such that KB was modified
                          since the previous call to handle_uncovered(e,Lj)
                        then delete e from postpone; this activates e
                                call handle_constraint(e)
                        endif
                endif
        endif
endproc
```

Algorithm 6.6: Integration in Clint (continued)

```
procedure handle_ka_not_covered(p(o_1, ..., o_n) : true uncovered fact)
    if φ_2(KB, p(o_1, ..., o_n)) = false and τ_2(p(o_1, ..., o_n)) = true
      and one of the o_i is a new object
    then if p is a rule predicate and p(o_1, ..., o_n) is ground
           then call handle_rule(p(o_1, ..., o_n))
           else call handle_fact(p(o_1, ..., o_n))
           endif
    else call handle_not_covered(p(o_1, ..., o_n))
    endif
endproc
```

Chapter 7

Negation and Multi-valued Logic.

7.1 Introduction.

Many machine learning techniques (e.g. explanation-based learning [Mitchell, Keller and Kedar-Cabelli 86], [De Jong and Mooney 86], Marvin [Sammut and Banerji 86] and more recently Cigol [Muggleton and Buntine 88] and also Clint [De Raedt and Bruynooghe 90c]) have difficulties in handling negative and uncertain information. Several conceptual and pragmatic problems are raised by negation and uncertainty in concept-learning. Conceptually there is a major problem, due to the use of negation as failure as a basis for negation. For concept-learning, negation as failure (and the underlying closed world assumption) is unacceptable because it acts as if everything is known. Clearly, in learning this is not the case, since otherwise nothing ought to be learned. Pragmatic problems with negation and uncertainty are due to the increase in the search space when negation is allowed and to the implementation of negation. Negation increases the search space since the expressivity of the concept-description language grows. Also, allowing negation often amounts to allowing disjunction since conjunctions with negation can be rewritten as disjunctions. That disjunction is a difficult problem for concept-learning is well-known (see e.g. [Cohen and Feigenbaum 81, Bundy, Silver and Plummer 85, De Raedt and Bruynooghe 88] and chapter 2). The implementation of negation may also raise problems since negation as failure is not constructive. Consider for example Prolog, where a query containing negative literals is only handled correctly when the negative literals are ground at call time (cf. also section 3.7). Research concerning constructive negation in logic programs has been initiated by [Chan 88] and is being continued by [Puget 89] amongst others.

In this chapter, a multi-valued logic with an explicit definition of the negated concept is proposed to alleviate the above sketched problems. The proposed technique is integrated in the system Clint and is shown to be sound. Although the technique is presented in the context of Clint, it also applies to other learning systems that use a first order logic framework. The technique can be regarded as a generalization of Mitchell's versionspace method [Mitchell 77], where new examples can be classified in three ways: positive, negative and unknown. The expressiveness of the concept-description language, which is severely limited for versionspaces, is significantly extended here.

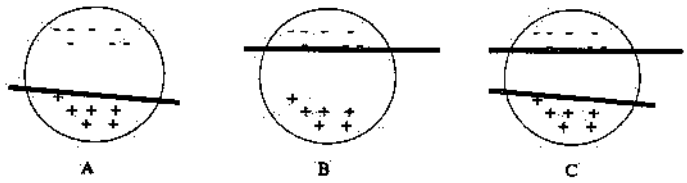
I organized this chapter as follows: in section 7.2 we analyze the problems with negation and discuss why classical solutions are not acceptable; in section 7.3 the problem is solved at an abstract level; in section 7.4 we define a practical multi-valued logic based on logic programming concepts; this logic is used in section 7.6 by the modified algorithms of

Clint and the knowledge acquisition procedures of chapter 6; in section 7.7 the properties of the new version of Clint are analyzed and; finally, in section 7.8 we conclude and discuss some related work.

7.2 The problems with negation.

The main conceptual problem with most approaches to negation — is that they assume everything to be known. This offers great computational advantages as one can use negation as failure to handle negative knowledge. However, it is well-known that negation as failure is only valid when the closed world assumption holds. From the viewpoint of learning this leads to the following paradox: if everything is known (which is the intuition behind the closed world assumption), why should we still learn something ? As a result, it is either necessary to learn, in which case the closed world assumption and negation as failure are not valid, or the closed world assumption and negation as failure apply, and nothing ought to be learned. Of course, it may be argued that one should learn in the case of the closed world assumption and negation as failure when there is an inconsistency in the knowledge base. Although this represents a plausible view, there is in general no other means than the external world to signal such inconsistencies. This leads to a second and related problem. The result of various forms of negation is that unseen examples are classified on an all or nothing basis: either they are classified as positive or negative (but not anything else). This may result in problematic situations for learners. Consider for instance an autonomous agent that selects its own experiences (e.g. when to apply a certain action). Using a cautious approach (and learning in a specific to general manner), the agent will only know what is likely to be allowed. So, before the search converges, it cannot distinguish between what is likely to be false (forbidden) and what is not yet known. Clearly, such a situation can be quite dangerous. This is illustrated in figure 7.1.a, where everything above the line cannot be distinguished. When the system searches general to specific, it does not know the difference between what is likely to be true (allowed) and what it does not know yet (see figure 7.1.b). If the system alternates between searching from general to specific and from specific to general, it knows even less. So, the problem is that the learned knowledge cannot be used in practice before the search has converged. A more natural situation arises in the candidate elimination approach [Mitchell 77, Mitchell 82], where three values are distinguished: true, false and unknown (figure 7.1.c). This is much more interesting since one can now reason about what is unknown (and ought to be learned). Although versionspaces present a nice theoretical framework, there are some major problems when they are applied in practice: they have difficulties in handling disjunction and therefore also often with negation (cf. [Murray 87]). If disjunction is allowed then the S-set becomes the disjunction of all positive examples, while if negation and conjunction are allowed the G-set becomes the conjunction of the negation of all negative examples. So, the expressiveness of concepts for versionspaces is severely limited. Another problem with negation in a two-valued logic setting is that for most predicates, there are much more false facts than true ones [Reiter 87]. Strict negation allows the derivation of irrelevant facts such as `not married(snake-1,chair-7)`. If this kind of negation is used in interactive systems such as Clint and Marvin [Sammut and Banerji 86] irrelevant hypotheses will be investigated and irrelevant questions will be asked. So, it is important and natural that certain facts are irrelevant (and neither

Figure 7.1: Two-valued and three-valued classification.



provable false nor true) for this reason.

Further problems with negation are of a more pragmatic nature: the use of negation significantly increases the searchspace and there are often implementation difficulties because of the non-constructive nature of negation as failure (cf. Prolog [Sterling and Shapiro 86]). The non-constructive nature of negation as failure leads to severe problems. Remember section 3.7 where a perfect query answering procedure was discussed.

In this chapter, a multi-valued logic is used. Examples can be classified in basically four different ways: positive, negative, unknown and inconsistent. Also, an explicit definition of the negated concept is learned. The advantage is that it solves all the above mentioned problems with negation and that inconsistencies can be detected by the system, rather than by an external agent.

7.3 An abstract approach.

In this section, two abstract approaches to cope with the problem of negation and a notion of the unknown are outlined. First, the approach taken in the versionspace paradigm of [Mitchell 77, Mitchell 82] is briefly discussed. It is shown that the approach does not solve all of the outlined problems. This motivates the second approach, which uses the abstract concept-learners $\mathcal{VS}$ and $\mathcal{DVS}$.

7.3.1 Mitchell's versionspace approach.

Mitchell's versionspace approach solves the problems with the representation of the unknown. Indeed, [Mitchell 82] shows that the versionspace approach actually represents three values:

- If an example is covered by all elements in the versionspace $(\mathcal{D})\mathcal{VS}$, it is assigned the value `true`.
- If an example is covered by no element in the versionspace $(\mathcal{D})\mathcal{VS}$, it is assigned the value `false`.

- Otherwise, it is assigned the valued unknown.

Using Mitchell's versionspace approach, the first case is equivalent to the example being covered by all elements in the S-set and the second case to the example not being covered by any element in the G-set. These tests are clearly more efficient.

Although Mitchell's approach allows to introduce a three-valued logic in an elegant way, it cannot be easily extended towards other approaches such as Clint. The reason is that the versionspace approach requires that one takes into account *all* complete and consistent concept-descriptions. In the local approaches (see section 3.5.1) this is impossible. Furthermore Mitchell's approach does not solve the problems with negation. In a logical framework, it is desirable to have an explicit definition of the negated concept in order to alleviate the non-constructive nature of negation as failure. In Mitchell's approach there is no such explicit definition. A slightly different approach that avoids these problems is outlined in the next section.

7.3.2 A general approach: explicit representation of the negated concept.

To solve the problems with the non-constructive nature of negation as failure, an explicit representation of the negated concept is needed. Let us therefore learn two concepts: a positive definition $p \in (\mathcal{D})\mathcal{VS}(KB, L, P, N)$ and a negative definition $n \in (\mathcal{D})\mathcal{VS}(KB, L, N, P)$. To learn an explicit definition of the negated concept one only has to reverse the examples: all examples that are positive for the positive concept are negative for the negated concept and all examples that are negative for the positive definition are positive for the negative definition.

Given such a positive definition $p \in (\mathcal{D})\mathcal{VS}(KB, L, P, N)$ and a negative definition $n \in (\mathcal{D})\mathcal{VS}(KB, L, N, P)$, an example e can be classified in four different ways:

- if e is covered by p and not covered by n then e is `true`
- if e is covered by p and covered by n then e is `inconsistent`
- if e is not covered by p and covered by n then e is `false`
- if e is not covered by p and not covered by n then e is `unknown`

Because examples can now be classified in 4 different way, we also modify our notion of the intended interpretation to cope with this new situation:

Definition 7.1 *The multi-valued intended interpretation τ_M, can yield 4 different truthvalues when applied on an example e. $\tau_M(e) = t$, where t = true, false, unknown or inconsistent.*

The value unknown can be used to keep uninteresting atoms such as `not married(snake-12,chair-89)` unknown in the intended interpretation. The value `inconsistent` can for instance be used when the intended interpretation of an example is determined by consulting two experts and they disagree on the truthvalue of the example.

Using this approach there is a fourth truthvalue: `inconsistent`. Having this fourth truthvalue is not always attractive unless the training set itself contains inconsistent information. In interactive concept-learners, this fourth truthvalue can easily be excluded, when desired. If at a certain point in time the negative definition n and the positive definition p overlap, i.e. they both cover an example e, the learner can ask the oracle to classify such an example e according to τ_M. The answer to the question makes it possible to modify the responsible definition(s) :

- if the example e is `true`, then the negative definition should be modified as it covers an example that is negative to it.
- if the example e is `false`, then the positive definition should be modified as it covers an example that is negative to it.
- if the example e is `unknown`, then the positive and the negative definitions should both be modified as they both cover an example that is negative to it
- the example e should not be classified as `inconsistent` because otherwise the fourth truthvalue is not excluded

Using this approach, all the problems sketched in section 7.2 are solved as there is a (constructive) explicit definition of the negated concept in the knowledge base and a possibility to reason about the unknown (and the inconsistent).

7.4 A multi-valued logic for concept-learners.

In this section[1], the ideas concerning the multi-valued logic are integrated in logic programming. This will result in a modified logical framework for Clint. In section 7.6, the algorithms of Clint will be modified in order to cope with this extended logical framework.

The multi-valued logic is based on logic programming [Lloyd 87] concepts. The basic idea behind the approach is very simple: in order to reason about the (un)known and the (in)consistent it is necessary to have knowledge about truth and falsity; therefore negation has to be represented explicitly; however, because there are some difficulties with deduction mechanisms when negation is explicitly represented (e.g. in logic programming [Lloyd 87]) these problems are circumvented by considering occurrences of a negated predicate $\neg p$ as the occurrences of a different predicate $\overline{p}$. So, p and $\overline{p}$ are considered to be two different predicates, with the property that $\overline{\overline{p}} = p$. To explicitly represent the concepts p and $\overline{p}$, one has to provide Horn clause definitions for $\overline{p}$ and p and also to replace all occurrences of $\neg p$ in the bodies of clauses by $\overline{p}$. In the examples we will use the notation `not_p` for the negated predicate `p`. So, we have that $\overline{\texttt{not_p}} = \texttt{p}$ and $\overline{\texttt{p}} = \texttt{not_p}$.

Because of the multi-valued framework, we now have two different levels : the object-level which contains the translation of the meta-level. An example e in the multi-valued framework always corresponds to two facts e and $\overline{e}$ at the two-valued object-level.

Definition 7.2 *The dual fact of a fact* $f = p(t_1, ..., t_n)$ *is* $\overline{f} = \overline{p}(t_1, ..., t_n)$.

[1]The semantics of the logic presented in this section are – in part – due to Bern Martens and Wlodek Drabent.

The interpretation $\varphi_M(KB,e)$ of elements e in the Herbrand base of a knowledge base KB, is defined below. The M in φ_M denotes that the multi-valued interpretation is used, φ_2 denotes the two-valued interpretation ($\models$), while φ denotes any of these interpretations (see section 3.2).

Definition 7.3 *The truthvalue $\varphi_M(KB,e)$ of an element $e \in$ Herbrand base of KB is:*

- $\varphi_M(KB,e) =$ `true` *iff* $\varphi_2(KB,e) =$ *true and* $\varphi_2(KB,\overline{e}) =$ *false.*
- $\varphi_M(KB,e) =$ `false` *iff* $\varphi_2(KB,e) =$ *false and* $\varphi_2(KB,\overline{e}) =$ *true*
- $\varphi_M(KB,e) =$ `unknown` *iff* $\varphi_2(KB,e) =$ *false and* $\varphi_2(KB,\overline{e}) =$ *false*
- $\varphi_M(KB,e) =$ `inconsistent` *iff* $\varphi_2(KB,e) =$ *true and* $\varphi_2(KB,\overline{e}) =$ *true*

This definition specifies a *partial* semantics because only the truth-value of elements in the Herbrand base is defined. It does not specify the semantics of *formulas* as is done in e.g. [Turner 84]. It is possible to avoid the specification of the truth-values of formulas as a consequence of the reduction to two-valued logic. Also, in practical applications like concept-learning, people are often only interested in truth-values of elements of the Herbrand base and not of whole formulas. The given semantics directly yields an implementation as a meta-interpreter in Prolog (cf. [De Raedt, Martens and Bruynooghe 90]).

Lemma 7.1 *We have that :*

- $\varphi_M(KB,e) =$ `true` iff $\varphi_M(KB,\overline{e}) =$ `false`.
- $\varphi_M(KB,e) =$ `false` iff $\varphi_M(KB,\overline{e}) =$ `true`.
- $\varphi_M(KB,e) =$ `unknown` iff $\varphi_M(KB,\overline{e}) =$ `unknown`.
- $\varphi_M(KB,e) =$ `inconsistent` iff $\varphi_M(KB,\overline{e}) =$ `inconsistent`.

We now have two different levels : the meta-level which corresponds to the multi-valued semantics and interpretation and the object-level which corresponds to the translation of the meta-level into a two-valued semantics and interpretation. Definition 7.3 states how the semantics of the meta-level are translated into the semantics of the object-level. For the multi-valued interpretation τ_M, there is a similar translation into the corresponding two-valued interpretation τ_2 :

Definition 7.4 *Translating τ_M into τ_2 :*

- $\tau_M(e) =$ `true` iff $\tau_2(e) =$ true and $\tau_2(\overline{e}) =$ false.
- $\tau_M(e) =$ `false` iff $\tau_2(e) =$ false and $\tau_2(\overline{e}) =$ true.
- $\tau_M(e) =$ `unknown` iff $\tau_2(e) =$ false and $\tau_2(\overline{e}) =$ false.
- $\tau_M(e) =$ `inconsistent` iff $\tau_2(e) =$ true and $\tau_2(\overline{e}) =$ true.

This defines τ_2.

The translation from τ_M to τ_2 will be used to obtain a knowledge base KB that satisfies all examples at the meta-level. In order to satisfy all examples at the meta-level (τ_M), one only has to assure that all translated examples at the object-level (τ_2) are satisfied :

Lemma 7.2 *The knowledge base KB satisfies the example e at the meta-level, i.e.* $\varphi_M(KB, e) = \tau_M(e)$ *iff the knowledge base KB satisfies the example e at the object-level, i.e.* $\varphi_2(KB, e) = \tau_2(e)$ *and* $\varphi_2(KB, \overline{e}) = \tau_2(\overline{e})$.

Proof : This follows directly from definitions 7.3 and 7.4. □

It is also necessary to modify the definition of a concept-description for the multi-valued framework:

Definition 7.5 *A multi-valued concept-description of a predicate* p *is a set of Horn-clauses for the predicate* p *and possibly its dual* not_p. *The set of Horn-clauses defining* p *is the positive definition, the corresponding set for* not_p *is the negative definition.*

Whereas examples are now interpreted at two different levels (φ_2 and φ_M), we will retain the two-valued interpretation for constraints ! This means that – as before – a constraint c is satisfied iff $KB \models c$.

The main difference between the two-valued and the multi-valued framework is that in the two-valued framework, the negative definition does not exist and the above sketched translation from meta- to object-level does not take place!

Example 7.1 illustrates the multi-valued framework.

7.5 Special cases of the multi-valued logic.

When certain constraints hold for a knowledge base, we obtain special cases of the multi-valued logic. We can obtain two different kinds of three-valued logics and a (classical) two-valued logic with explicit negation.

7.5.1 A three-valued logic : true, false and unknown.

If for each predicate p and its dual $\overline{p}$ it is the case that $KB \models (p \wedge \overline{p} \rightarrow$ `false`$)$, one obtains a three-valued logic without the truth-value `inconsistent`. No example will be classified as `inconsistent` because of these constraints.

This logic corresponds closely to the logic used in versionspaces (see also figure 7.1 and section 7.3.1). I believe that this logic is the most useful one for concept-learning because it solves the sketched problems with negation and the unknown (see also [De Raedt and Bruynooghe 90c]). Secondly, it can be argued that an intelligent concept-learner should not derive a knowledge base that classifies examples as inconsistent (φ_M), unless the examples are classified as inconsistent by the user (τ_M). Thirdly, the value unknown can be used by the user to keep examples such as `married(snake-87, chair-2)` unknown. The proposed three-valued logic satisfies these demands.

In general, the following lemma holds for constraints in this three-valued logic :

Lemma 7.3 *The constraint* $p_1 \wedge ... \wedge p_n \rightarrow q_1 \vee ... \vee q_k$ *logically implies* $p_1 \wedge ... \wedge \overline{q_i} \wedge ... \wedge p_n \rightarrow q_1 \vee ... \vee q_{i-1} \vee q_{i+1} ... \vee q_k$ *in the logic with values true, false and unknown.*

Example 7.1: Concept-Learning in multi valued logic

Suppose the knowledge base consists of the following definitions:

```
ostrich(oliver)  ←
swallow(woody)  ←
penguin(iglo) ←
parrot(kiki) ←
sparrow(sperry) ←

bird(B)  ← ostrich(B)
bird(B) ←  parrot(B)
bird(B)  ←  swallow(B)
bird(B)  ←  penguin(B)

normal_bird(B)  ← parrot(B)
normal_bird(B)  ← ostrich(B)
normal_bird(B)  ← swallow(B)
not_normal_bird(B)  ← ostrich(B)
```

Assume also the following examples: $\tau_M(\texttt{flies(iglo)}) = \text{false}$, $\tau_M(\texttt{flies(woody)}) = \text{true}$, $\tau_M(\texttt{flies(frank)}) = \text{unknown}$ and $\tau_M(\texttt{flies(kiki)}) = \text{true}$.
Assume that the language of completely bound Horn-clauses L_0 (see chapter 4 and example 3.3) is being used. Then the following concept-description satisfies these examples.

```
flies(F)  ←  normal_bird(F)
not_flies(F)  ←  pinguin(F)
```

Notice that certain facts are unknown (e.g. `normal_bird(iglo)`) or inconsistent (e.g. `normal_bird(oliver)`). An important property of the framework is that it is not allowed to rewrite clauses by applying two valued equivalence preserving transformations. E.g. the clause `flies(F)` $\wedge$ `not_flies(F)` $\rightarrow$ `false` is equivalent to $\rightarrow$ $\neg$`flies(F)` $\vee$ $\neg$`not_flies(F)` but not to $\rightarrow$ `not_flies(F)` $\vee$ `flies(F)`. The reason is $\neg p \neq \overline{p}$. See also section 7.5 for a discussion of these issues.

Proof : We have the following :

$$p_1 \wedge ... \wedge p_n \rightarrow q_1 \vee ... \vee q_k$$

implies

$$p_1 \wedge ... \wedge p_n \rightarrow q_1 \vee ... \vee q_{i-1} \vee \neg\overline{q}_i \vee q_{i+1} \vee ... \vee q_k$$

because

$$q_i \wedge \overline{q}_i \rightarrow \texttt{false} \text{ is equivalen to } q_i \rightarrow \neg\overline{q}_i$$

The second formula is then equivalent to :

$$p_1 \wedge ... \wedge \overline{q}_i \wedge ... \wedge p_n \rightarrow q_1 \vee ... \vee q_{i-1} \vee q_{i+1} ... \vee q_k$$

□

We have that `bird(X)` $\rightarrow$ `flies(X)` $\vee$ `not_normal_bird(X)` implies `bird(X)` $\wedge$ `normal_bird(X)` $\wedge$ `not_flies(X)` $\rightarrow$ `false`. The reverse of the lemma does not hold; this is shown in example 7.2. So, it is allowed to replace a literal p in the head of the constraint by the literal $\overline{p}$ in the body, but not vice versa !

Example 7.2: Illustrating constraints in 3-valued logic

Suppose we have the following knowledge base :

`q` $\leftarrow$

This knowledge base satisfies the three-valued constraints `not_p` $\wedge$ `p` $\rightarrow$ `false` and `not_q` $\wedge$ `q` $\rightarrow$ `false` and also the constraint `p` $\wedge$ `q` $\rightarrow$ `false`. However, it does not satisfy `q` $\rightarrow$ `not_p`.

7.5.2 A three-valued logic : true, false and inconsistent.

The dual three-valued logic of the one discussed above is obtained when for all predicates p and their dual $\overline{p}$, $KB \models (\texttt{true} \rightarrow p \vee \overline{p})$. This logic excludes the value unknown.

For this logic, the dual property holds :

Lemma 7.4 *The constraint* $p_1 \wedge ... \wedge p_n \wedge \overline{q}_i \rightarrow q_1 \vee ... \vee q_{i-1} \vee q_{i+1} ... \vee q_k$ *logically implies* $p_1 \wedge ... \wedge p_n \rightarrow q_1 \vee ... \vee q_k$ *in the logic with values true, false and inconsistent.*

Proof : We have the following :

$$p_1 \wedge ... \wedge p_n \wedge \overline{q}_i \rightarrow q_1 \vee ... \vee q_{i-1} \vee q_{i+1} ... \vee q_k$$

is equivalent to

$$p_1 \wedge ... \wedge p_n \rightarrow q_1 \vee ... \vee q_{i-1} \vee \neg\overline{q}_i \vee q_{i+1} \vee ... \vee q_k$$

Then because

$$\texttt{true} \rightarrow q_i \vee \overline{q}_i \text{ is equivalent to } \neg\overline{q}_i \rightarrow q_i$$

the second formula implies :

$$p_1 \wedge ... \wedge p_n \rightarrow q_1 \vee ... \vee q_k$$

□

7.5.3 A classical two-valued logic : true and false.

When for all predicates p and their dual $\overline{p}$, we have that $KB \models (\texttt{true} \rightarrow p \vee \overline{p})$ and $KB \models (p \wedge \overline{p} \rightarrow \texttt{false})$, we obtain a classical two-valued logic with explicit negation.

In this logic the values unknown and inconsistent are excluded. Furthermore, we obtain a classical two-valued logic. This is revealed by the following lemma.

Lemma 7.5 *In the logic with values true and false we have that* $\neg p \leftrightarrow \overline{p}$.

Proof : The constraint $p \wedge \overline{p} \rightarrow \texttt{false}$ is equivalent to $\overline{p} \rightarrow \neg p$. The constraint $\texttt{true} \rightarrow p \vee \overline{p}$ is equivalent to $\neg p \rightarrow \overline{p}$. □

7.6 Adapting Clint.

This section shows how Clint can be adapted to cope with the multi-valued framework.

7.6.1 The modified problem statement of Clint.

The modified problem statement for Clint is now (cf. problem statement 3.1 and chapter 4):

Problem Statement 7.1 *Multi-valued problem statement of Clint*

- **Given:**
  - a set of examples E
  - a set of constraints C; for all constraints $c \in C : \tau_2(c) = \text{true}$.
  - a knowledge base KB of learned and basic multi-valued concept-descriptions;
  - a series of languages $L_1, ..., L_M$ defining Horn-clauses
  - a new constraint c or a new example e;
  - an oracle willing to answer membership questions about τ_M
- **Find:** a knowledge base KB', obtained by retracting and asserting clauses belonging to L_i for some i from KB, such that KB' satisfies all constraints in $C \cup \{c\}$ and all examples in $E \cup \{e\}$;

This problem statement differs from the problems attacked in chapters 3 and 4 by the use of multi-valued predicate definitions and the multi-valued intended interpretation.

7.6.2 Modifying Clint.

Let us now show how the algorithms of Clint as presented in previous chapters can be adapted. First we show how to modify the algorithms of Clint and then how to modify the knowledge acquisition procedure of chapter 6. We also discuss how to obtain the special logics presented in section 7.5.

Clint

In order to cope with the modified problem statement, one must only handle meta-level examples twice at the object-level as indicated in definition 7.4 and lemma 7.2. Incorporating this in the algorithms, presented in the previous chapters, is trivial : one only has to interpret the τ in these algorithms as τ_M and translate τ_M into the corresponding τ_2 following definition 7.4. This can easily be realised using assumption 7.1.

Assumption 7.1 *Multi-valued Clint.*

- *When an example e is acquired such that $\tau_2(e) = true$ (according to τ_M), add the constraint* (`true` $\rightarrow$ e) *to the constraint theory.*
- *When an example e is acquired such that $\tau_2(e) = false$, add the constraint* ($e \rightarrow$ `false`).
- *For the eager method it is desirable (but not necessary) to also slightly modify the definition of an* acceptable *constraint to the one in definition 7.6.*

Definition 7.6 *A constraint c is applicable to a predicate p iff*

1. *c is an example or*
2. *p or $\overline{p}$ occurs in c and c does not depend on a predicate q or $\overline{q}$ ($q \neq p$ and $q \neq \overline{p}$) that does not satisfy all examples*

The reason why it is desirable to modify the definition of an applicable constraint is that definition 7.6 will also take into account the constraints that mention the dual predicate. This is certainly important in the context of the constraints used to obtain the special cases of the multi-valued logic. Allowing those constraints to be applicable cannot give any problems because all the facts that are examples for the predicate p are also examples for $\overline{p}$. So, querying the oracle about p is also querying it about $\overline{p}$, and vice versa.

When assumption 7.1 is satisfied, the versions of Clint presented in chapter 5 will automatically take care that the knowledge base satisfies these examples at the meta- and object-level. Obviously the outlined approach to cope with a multi-valued logic retains its identification in the limit behaviour. This directly follows from the theorems in chapter 5 and lemma 7.2.

Theorem 7.1 *Any multi-valued predicate q will be error-free identified using assumption 7.1 and the eager or the lazy constraint algorithm, from a definitely correct knowledge base KB and any constraint presentation of q w.r.t. KB if q and $\overline{q}$ are expressible without using recursion in terms of the languages $L_1, ..., L_M$ and KB.*

Theorem 7.2 *A set of multi-valued concepts C, such that all predicates p and $\overline{p}$ in C are expressible in one of the languages $L_1, ..., L_M$ and KB, will be postponed identified using assumption 7.1 and the eager or the lazy constraint algorithm from a definitely correct starting knowledge base KB and any constraint presentation of C w.r.t. KB.*

Observe that these theorems mean that the system will converge to a knowledge base in which τ_M and φ_M fall together.

Knowledge Acquisition

It is also easy to modify the knowledge acquisition procedures of chaper 6 to cope with the multi-valued logic. One only has to use the algorithms of chapter 6 with the following convention:

Assumption 7.2 *Multi-valued Knowledge Acquisition*

- *When an example e is acquired such that $\tau_2(e)$ = true (according to τ_M), add the constraint* (`true` $\rightarrow e$) *to the constraint theory.*
- *When an example e is acquired such that $\tau_2(e)$ = false (according to τ_M), add the constraint* ($e \rightarrow$ `false`).
- *When an existential question q is formulated to the oracle, and the oracle returns a substitution θ such that $q\theta$ is not ground, and $\tau_2(q\theta)$ = true (according to τ_M), then assert the fact $q\theta$ in the knowledge base.*
- *When an existential question q is formulated to the oracle, and the oracle returns a substitution θ such that $q\theta$ is not ground, and $\tau_2(q\theta)$ = false (according to τ_M), add the constraint* ($q\theta \rightarrow$ `false`) *to the constraint theory.*
- *For the eager method it is desirable (but not necessary) to also slightly modify the definition of an* acceptable *constraint to the one in definition 7.6.*

The only further required modification is that we should also treat answers to existential questions. When the answer to such a question q states e.g. that $\tau_M(q\theta)$ = true and $q\theta$ is not ground, it means that $\tau_2(q\theta\rho)$ is true and $\tau_2(\overline{q}\theta\rho)$ is false for all substitutions ρ. We can assure that $\varphi_2(KB, \overline{q}\theta\rho)$ is false for all substitutions ρ by adding the constraint ($\overline{q}\theta \rightarrow$ `false`) to the constraint theory. To make $\tau_2(q\theta\rho)$ true for all substitutions ρ, we could add the constraint (`true` $\rightarrow q\theta$) to the knowledge base. This is – however – not allowed in the formalism of Clint because it is assumed that all constraints are range-restricted (see chapter 5). Furthermore, if we would merely add the constraint to the knowledge base, the knowledge base may become violated many times. The reason is that the fact $q\theta$ contains (universally quantified) variables. To make such a constraint consistent, many modifications may have to be effected on the knowledge base. Therefore it is more appropriate to add the fact to the knowledge base. Then the constraint will automatically be satisfied. Moreover, the fact will never be retracted from the knowledge base as it is correct in the intended interpretation (τ_2). So, the constraint (`true` $\rightarrow q\theta$) can be forgotten and need not be tested for violations.

The special logics.

Realizing the special logics presented in section 7.5 is quite easy : one only as to add certain constraints to the constraint theory. Because Clint will take care that these constraints are always satisfied, this is sufficient for realizing the logics.

Formulated more precisely, this means that :

- to obtain the three-valued logic with values true, false and unknown, one has to assert for each predicate p and its dual $\overline{p}$, the constraint $p \wedge \overline{p} \rightarrow$ `false`.
- to obtain the three-valued logic with values true, false and inconsistent, one has to assert for each predicate p and its dual $\overline{p}$, the constraint `true` $\rightarrow p \vee \overline{p}$.
- to obtain the classical two-valued logic with an explicit negation and the values true and false, one has to assert for each predicate p and its dual $\overline{p}$, the constraints `true` $\rightarrow p \vee \overline{p}$ and $p \wedge \overline{p} \rightarrow$ `false`.

In [De Raedt and Bruynooghe 90c], we have presented a special version of Clint to learn from examples in the three-valued framework with values true, false and unknown. This version corresponds to a special case of the algorithm obtained by using assumption 7.1 and the eager method. The difference between the algorithm in [De Raedt and Bruynooghe 90c] and the one presented here is that the one of [De Raedt and Bruynooghe 90c] only allows to use examples and clauses of the form $p \wedge \overline{p} \rightarrow$ `false` as constraints.

An example session with this three-valued (true, false and unknown) algorithm that uses assumption 7.1 and the eager constraint mechanism is shown in example 7.3.

Example 7.3: A session with the three-valued Clint.

Suppose we have the following definitions in our knowledge base a.o.:

```
parrot(kiki) ←                    bird(X) ← parrot(X)
pinguin(lulu) ←                   bird(X) ← pinguin(X)
ostrich(oli) ←                    bird(X) ← ostrich(X)
swallow(tweety) ←                 bird(X) ← swallow(X)
swallow(woody) ←
human(jeffrey) ←                  not_human(clint) ←

not_parrot(X) ← swallow(X)        not_ostrich(X) ← swallow(X)
not_parrot(X) ← ostrich(X)        not_ostrich(X) ← pinguin(X)
not_parrot(X) ← pinguin(X)        not_ostrich(X) ← parrot(X)

not_pinguin(X) ← parrot(X)        not_swallow(X) ← ostrich(X)
not_pinguin(X) ← swallow(X)       not_swallow(X) ← parrot(X)
not_pinguin(X) ← ostrich(X)       not_swallow(X) ← pinguin(X)
```

A typical run with the system goes as follows (user input is in **bold**, system output is typewritten):

```
?-learn.
Next example ?  @ flies(kiki).
Classification ?  @ true.
Starting clause for flies(kiki) in L0:

      flies(X) ← parrot(X) ∧  not_swallow(X) ∧
            not_ostrich(X) ∧  not_pinguin(X) ∧  bird(X)

Is flies(oli) true ?  { parrot(X), not_ostrich(X) } deleted @ no.
Is flies(lulu) true ?  { parrot(X), not_pinguin(X) } deleted @ no.
Is flies(tweety) true ?  { parrot(X), not_swallow(X) } deleted @ yes.
Generalized clause:

      flies(X) ← not_ostrich(X) ∧  not_pinguin(X) ∧  bird(X)

No more valid generalizations.  Asserted clause:

      flies(X) ← not_ostrich(X) ∧  not_pinguin(X)

not_flies(oli) not covered, learning starts ...
Starting clause for not_flies(oli) in L0:

      not_flies(X) ← ostrich(X) ∧  not_swallow(X) ∧
            not_parrot(X) ∧  not_pinguin(X) ∧  bird(X)

I know not_flies(lulu) is positive, so I may generalize !
Generalized clause:

      not_flies(X) ←  not_swallow(X) ∧  not_parrot(X) ∧  bird(X)

No more valid generalizations.  Asserted clause:

      not_flies(X) ←  not_swallow(X) ∧  not_parrot(X)
```

Notice that the system deals successfully with negation and also that irrelevant facts as `not_flies(clint)` remain unknown.

7.7 Properties of the three-valued Clint.

First of all, the new version of our algorithms solves all the above sketched problems with negation:

- Since the negative definition is in the form of a set of Horn-Clauses for the predicate `not_p`, it is constructive. In this way it is feasible to learn definitions with negative literals in the body (e.g. `normal_bird(X) ← bird(X) ∧ not_pinguin(X) ∧ not_ostrich(X)`). This was not possible with the previous versions of Clint and is not treated in related systems such as the Model Inference System [Shapiro 83], Cigol [Muggleton and Buntine 88] and Marvin [Sammut and Banerji 86].

- Since the system represents what it does not know and what is inconsistent, it is possible to learn a new concept even if the other concepts are still partly unknown or inconsistent. This is much more natural than requiring that a predicate is completely learned before starting to learn another one.

- The negative and the positive definitions learned by Clint will only cover examples that are relevant for the concept. If the user always classifies irrelevant examples such as `flies(clint)` as unknown, the system will never generate questions related to such objects and the corresponding definitions will never cover such examples. So, these irrelevant facts will always remain unknown. This would also be achieved by using a typed or sorted logic as e.g. done by Blip [Morik 89, Wrobel 88].

- The algorithm takes a much more greedy approach to learning than the earlier versions. In the previous versions, the system learned only one clause from a positive example. Now, it can derive several clauses because during the generalization process negative examples may be gathered, which results in learning clauses for the corresponding dual concept and this may again give rise to learn clauses for the dual dual concept (i.e. the original one), etc ... So, on the one hand, given a user-supplied example, the new version will pose more questions, but on the other hand it will learn more from it.

- A further extension of Clint, would be to use the representation in multi-valued logic to generate interesting further examples. This can easily be done by taking unknown combinations of arguments such that the arguments occur in a true or false statement at the same position. Suppose that the positive definition covers only `p(a,b)` and the negative one only `not_p(c,d)`. Then it would be interesting to ask for the classification of `p(a,d)` and `p(c,b)`. This realizes a similar effect as using an order sorted logic.

- Previously, Clint could only learn starting from a positive fact. Now, it can also start from a negative fact. The reason is that positive and negative examples are treated in the same way but with the dual definitions.

- In this chapter, it has been assumed that all concepts are defined by two dual predicates. For some concepts this is more useful than for others.[2] Indeed, for certain concepts such as `flies` this is natural. However, for others such as `not_ostrich` this is less evident. Therefore, it may be desirable to define only certain concepts by using two dual definitions and to treat the other ones as in the previous chapters. This would be a straightforward extension of the technique: the user should only tell Clint which predicates should be treated in multi-valued logic and which ones in two-valued logic.

- Of course, all other capabilities of Clint are retained in the new version: the generation of the examples, shifting the bias, using knowledge, recovering from errors, ...

[2]This observation was made by Pavel Brazdil at the European Conference on Artificial Intelligence, Stockholm, 1990.

7.8 Conclusions and Related Work.

This chapter contains an adapted version of [De Raedt and Bruynooghe 90c]. The difference of [De Raedt and Bruynooghe 90c] with the technique presented here, is that in [De Raedt and Bruynooghe 90c] we presented a specific algorithm to handle the three-valued framework with values true, false and unknown, whereas in this chapter we presented a more general algorithm and framework. This is revealed by the fact that the algorithms given in this chapter to cope were obtained as a special case of the constraint handling mechanism instead of the special purpose algorithm given in [De Raedt and Bruynooghe 90c]. Secondly, I also showed how to obtain some other logics.

The idea of explicitly representing the negated concept is not really new. It is also possible to learn the negated concept in the AQ program of [Michalski 83], Blip [Morik 89] and Predictor [Gordon and Perlis 89]. Also relevant is the versionspace approach of [Mitchell 77] and Disciple where one also achieves a three valued logic. However, AQ is not incremental and Predictor and the versionspace approach are restricted to much more simple description languages. Furthermore, none of the mentioned systems except Blip is a closed loop system, where concepts can be learned using previously learned concepts, or uses the explicit representation of the negative definition in order to learn concepts with negative conditions. As a result our treatment of the problem extends the work of [Michalski 83, Gordon and Perlis 89, Mitchell 77, Tecuci and Kodratoff 90] in several directions. The approach in Blip is most similar to ours. However, there are also some important differences. Blip is a passive learner that does not ask questions to the user. As a consequence, Blip does not know that something is negative unless the user explicitly supplies negative examples to the system. According to Stephan Wrobel (personal communication) from the Blip-team, people tend not to state false facts. This implies that false facts are not acquired during the learning phase but later when the learned knowledge is tested and a contradiction is found by the user. When a contradiction is signaled, Blip recovers from the error. In Clint, negative examples are gathered and treated while learning because of the interaction with the user. A further difference with Blip's approach is that our τ_M can also yield the values unknown and inconsistent.

I have argued that there is a basic problem with negation (by failure) from a machine learning point of view and presented a simple but appealing method to avoid these problems. It explicitly represents and learns the negative definition. The technique has been applied to the system Clint. I have shown that the resulting system is sound and identifies concepts in the limit. I believe that our technique does not only apply to Clint but also to related systems that use a first order representation for concepts and examples as e.g. The Model Inference System [Shapiro 83], Marvin [Sammut and Banerji 86] and Cigol [Muggleton and Buntine 88].

Chapter 8

Constructive Induction by Analogy

Analogies prove nothing, that is quite true,
but they can make one feel more at home.
Sigmund Freud

8.1 Introduction.

Although Clint was presented as an *inductive* learning system, it suffers – like most other approaches to concept-learning – from the following paradox :

> *In order to induce a concept from examples, concept-learning systems perform deduction.*

Although this paradox appears – like most other paradoxes – rather strange, it is quite true when applied to Clint. In order to learn a concept, Clint uses a language and assumes that the concept can be expressed in that language. Given this assumption, the result of the concept-learning task follows by deduction as the incomplete and inconsistent concept-descriptions are discarded from consideration (c.f. also [Russell and Grosof 90]). The truth of the paradox can most easily be seen when considering the versionspace method or our abstract models of concept-learning $\mathcal{VS}$ and $\mathcal{DVS}$.

Because of this paradox, it is sometimes argued that the main problem of inductive techniques is not to learn *given* concepts, but rather to find or invent the appropriate concepts. This problem is often referred to as the *new term* problem. Recently, some authors have begun to address it [Muggleton and Buntine 88, Rendell 89, Mehra, Rendell and Wah 89, Muggleton 87, Wrobel 88]. Certainly, in the learning apprentice context, it is important to let the system suggest useful new concepts to the user. Therefore, we studied the problem in the framework of Clint and have enhanced Clint with the Constructive Induction by Analogy technique (Cia). In a sense, Cia can be seen as a language learning assistant since it interactively proposes new concepts to be included in the knowledge base. This extends the concept-description language. As such, Cia's aim is different from the usual constructive induction techniques as advocated by the Rendell-school [Rendell 89, Mehra, Rendell and Wah 89]. The Rendell-like approaches attempt to derive new terms from existing ones with the aim of better being able to learn one given concept. The Cia approach tries to invent new concepts that are of general interest to the user, without reference to the goal of learning one specific concept. In this respect, Cia is closer to the work of [Lenat 83].

Another and perhaps even more intriguing issue for machine learning is :

Can the learning process itself be learned and if so, how can this be done ? In other words: Can we apply machine learning techniques to machine learning?

Although there has not been much work to answer this question, there are some interesting initial studies like e.g. [Stepp, Whitehall and Lawrence 88] who discusses some general issues in learning learning machines, [Rendell, Seshu and Tcheng 87] who try to learn conditions under which to apply specific machine learning techniques and [Vrain and Lu 88] who apply analogy to do incremental concept learning.

In this chapter, we will present the Constructive Induction by Analogy technique, which addresses both of these current research trends. The technique is completely integrated in the Clint system. The key idea behind the CIA-technique is very simple : when concept-learning is viewed as a transformation process with an input (some examples) and an output (a concept-description) then one can use analogy to solve future concept-learning problems. The analogous method analyzes the result of the source concept-learning problem by deriving a concept-schema from it and tries to apply the schema in future concept-learning situations. The schema is not only used to solve concept-learning problems by analogy but also to invent new concepts.

This chapter is organized as follows : in section 8.2, we introduce some notions concerning schemata, in section 8.3, we show how analogy can be applied on concept-learning, in section 8.4, the constructive induction by analogy technique is presented, section 8.5 summarizes two experiments with the Cia technique and finally sections 8.6 and 8.7 evaluates the technique and discusses related work.

8.2 Some notions

In order to propose interesting new concepts to the user, the system uses second order schemata :

Definition 8.1 *A second order schema is an abstracted Horn-clause, where predicate names are interpreted as existentially quantified predicate variables.*

One such abstracted Horn-clause or second order schema is e.g.

$$S = (\exists p, q, r : p(X,Y) \leftarrow q(X,XW) \ \wedge \ q(YW,Y) \ \wedge \ r(XW, YW))$$

Second order schemata were introduced in the learning literature by the Blip-system and its precursors [Morik 89, Wrobel 89, Emde, Habel and Rollinger 83]. A related, but slightly more general notion was introduced by [Yokomori 86] in the context of logic programming.

Definition 8.2 *A second order substitution is a substitution that replaces predicate-variables by predicate-names.*

For schema S, Θ = { p = grandparent, q = parent, r = eq } is a second order substitution. The instantiated schema $S\Theta$ yields :

```
grandparent(X,Y) ← parent(X,XW) ∧ parent(YW,Y) ∧ eq(XW,YW)
```

The Cia-technique will in certain situations propose a number of predicate definitions to the user. For example, it will ask the user whether p defined as `p(X,Y)` ← `parent(X,Y)` ∧ `female(X)` is an interesting predicate. If the user believes the predicate is of interest, she can name it as e.g. `mother` and the resulting clause `mother(X,Y)` ← `parent(X,Y)` ∧ `female(X)` will be asserted in the knowledge base. Another type of question that is generated by Cia is a question whether a certain clause is correct. As an example, Cia might ask the user, while attempting to learn the definition of `grandparent`, whether `grandparent(X,Y)` ← `parent(X,Z)` ∧ `parent(Z,Y)` is correct. If the user confirms that the clause is correct, the clause will be asserted in the knowledge base. So, Cia uses two types of questions :

Definition 8.3 *A new term question asks to name a predicate p in a schema $\exists p : p(t_1,...,t_n) \leftarrow q_1,...,q_k$, where p is the only predicate-variable in the schema. When the oracle returns the substitution $\Lambda = \{p = P\}$, where P is a predicate-name, the clause $P(t_1,...,t_n) \leftarrow q_1,...,q_k$ is asserted in the knowledge base.*

Definition 8.4 *A clause question asks the oracle whether a clause c is correct. If the clause is correct, i.e. $\tau_2(c) = true$, it is asserted in the knowledge base.*

To generate interesting clause and new term questions, Cia uses the notion of matching a schema with a clause :

Definition 8.5 *A second-order schema S matches a clause c, notation $match(S,c)$, if $\exists \Theta, \rho : head(S)\Theta\rho = head(c)$ and $body(S)\Theta\rho \subseteq c$, where Θ is a second order substitution and ρ is a substitution that possibly renames variables in S.*

Definition 8.6 *A second-order schema S partially matches a clause c, notation $pmatch(S,C)$, if $\exists \Theta, \rho : body(S)\Theta\rho \subseteq c$, where Θ is a second order substitution and ρ is a substitution that possibly renames variables in S.*

These definions are illustrated in example 8.1.

In the Cia technique, partial matches will give rise to new term questions whereas complete matches will result in clause questions.

8.3 Applying analogy to concept-learning.

We first consider the classical analogy paradigm as proposed by [Chouraqui 85, Kodratoff 88, Kodratoff 90]. It is shown in figure 8.1.

Definition 8.7 *In this paradigm, a target* `Problem-2` *is solved by analogy to a source* `Problem-1` *by considering the similarity relations β and β' and the mappings α and α'.* `Solution-2` *is called an analogous solution to a target* `Problem-2` *iff the following relations hold :*

1. α(`Problem-1,Solution-1`).
2. α'(`Problem-2,Solution-2`).
3. β(`Problem-1,Problem-2`).

Example 8.1: Matching schemata

The schema S matches the clause :

```
grandparent(F,C) ← male(F) ∧ male(C) ∧ parent(F,M1)
      ∧ parent(M2,C) ∧ eq(M1,M2)
```

with substitution

Θ = {p = grandparent, q = parent, r = eq }
ρ = { X = F, Y = C, XW = M1, YW = M2 }.

The instantiated schema $S\Theta\rho$ is :

```
grandparent(F,C) ← parent(F,M1) ∧ parent(M2,C) ∧ eq(M1,M2)
```

The same clause partially matches the schema T :

$$T = (\exists p,q,r : \ p(X,Y) \leftarrow q(X) \wedge r(X,Y))$$

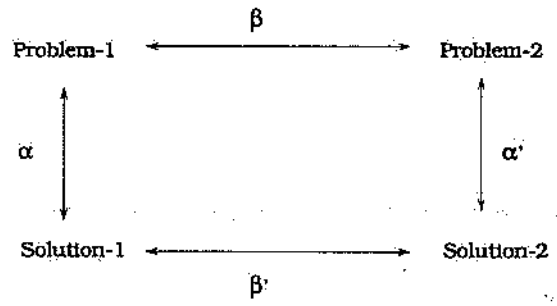
with substitution Λ = { q = male, r = parent}.

The partially instantiated schema $T\Lambda$ is :

$$\exists p : p(F,M1) \leftarrow male(F) \wedge parent(F,M1)$$

We ommit the substitutions ρ that rename the variables because they are straightforward.

Figure 8.1: Analogy



4. β'(Solution-1,Solution-2).

Of course, these relations depend on the particular application of analogy. In our setting we have that Problem-1 and Problem-2 are concept-learning problems, more specifically, sets of examples to be generalized. We also have that Solution-1 and Solution-2 are concept-descriptions which are complete and consistent for the examples of Problem-1 and Problem-2 respectively. The problem which we will address is is the following one :

Problem Statement 8.1 *Applying analogy on concept-learning.*

- **Given :**
  - a target concept-learning problem Problem-2
  - a set of source concept-learning problems :
    { (Problem-i, Solution-i) $\mid \forall i : \alpha$ (Problem-i, Solution-i) holds }
  - the relations α', β and β'

Find : an analogous solution to Problem-2

We will assume that α is accomplished by Clint although the technique could in principle also be applied in the context of other concept-learners. So, in order to describe our Cia technique in detail, we only have to define α', β and β'.

8.4 Constructive Induction by Analogy.

The main assumption behind the Cia technique is the following one :

Assumption 8.1 *Constructive Induction by Analogy*
Concept-descriptions are often very similar in the sense that they are instantiations of the same second order predicate schema.

The instantiations are of course obtained by applying second order substitutions.

An illustration of the assumption can be found in example 8.2.

It is important to note that assumption (or a variant of) 8.1 has been made and exploited by several other researchers :

- The Blip-system [Morik 89, Wrobel 88, Emde, Habel and Rollinger 83] uses second-order schemata in a model-driven approach to learning.
- [Yokomori 86] shows how (a variant of) second-order schemata can be used in a logic programming setting to decrease the memory needs of the knowledge base and to perform some kind of analogical reasoning.
- Dieter Poetschke[1] showed that many programs such as append and plus are analogous.

[1] In a presentation made at the workshop Analogical and Inductive Inference, Turingen, DDR, 1989.

Example 8.2: The assumption of Cia

Consider the concept-description :

$$\texttt{lighter(X,Y)} \leftarrow \texttt{weight(X,XW)} \wedge \texttt{weight(Y,YW)} \wedge \texttt{less(XW,YW)}$$

It is an instantiation of the following second-order schema with substitution

$$\Theta = \{\ p = \texttt{lighter},\ q = \texttt{weight},\ r = \texttt{less}\ \}$$
$$\exists p,q,r : p(\texttt{X,Y}) \leftarrow q(\texttt{X,XW}) \wedge q(\texttt{Y,YW}) \wedge r(\texttt{XW,YW})$$

Other instantiations of this schema together with their substitution are for instance

$$\texttt{same_color(X,Y)} \leftarrow \texttt{color(X,XC)} \wedge \texttt{color(Y,YC)} \wedge \texttt{eq(XC, YC)}$$
$$\Theta = \{\ p = \texttt{same_color},\ q = \texttt{color},\ r = \texttt{eq}\ \}$$
and
$$\texttt{brothers(X,Y)} \leftarrow \texttt{son(X,XP)} \wedge \texttt{son(Y,YP)} \wedge \texttt{eq(XP, YP)}$$
$$\Theta = \{\ p = \texttt{brothers},\ q = \texttt{son},\ r = \texttt{eq}\ \}$$
and
$$\texttt{near(X,Y)} \leftarrow \texttt{part_of(X,XP)} \wedge \texttt{part_of(Y,YP)} \wedge \texttt{next_to(XP,YP)}$$
$$\Theta = \{\ p = \texttt{near},\ q = \texttt{pat_of},\ r = \texttt{next_to}\ \}$$

The Cia technique in Clint employs the observed similarity by deriving from each learned concept-description a second-order schema specifying the relations between predicates in the concept-description. This derivation is straight-forward. One only has to replace all predicate names in the concept-description by predicate variables. Once such a schema is derived, it is stored to be matched with justifications when new concepts are learned.

Algorithm 8.1 shows when Clint calls Cia. This occurs at two different places :

- when Clint has found a consistent justification for a certain positive example, it will call the first part of Cia; this part tries to guess a correct clause for the concept being learned by asking clause questions
- after learning and asserting one clause for a concept, it may call the second part of Cia; this part attempts to invent or discover new concepts by asking new term questions

Let us now present these two parts of Cia (see algorithm 8.2).

cia_part_1 tries to make a guess at the correct concept-description by completely matching a consistent justification derived by Clint with one of its schemata. If the justification matches a schema, Cia proposes the instantiated second-order schema to the user and asks whether the resulting clause is correct for the concept being learned. If it is not, it may be that the proposed clause is actually a clause for an unknown but useful concept. Then the user may name the concept, such that the system can assert it in its knowledge-base. On the other hand if the user confirms that Cia has discovered the target concept-description,

Cia may halt or it can continue to propose concept-descriptions by instantiating other schemata in order to discover new concepts. If the target concept is not found, Clint has to continue in order to learn the concept. Once the concept is learned by Clint, a second order schema is derived from the learned clause. In that case, it may occur that the schemata and the starting clause do not match completely, but partially (only the bodies match). Then, more new predicates may also be proposed. This is done in part-2. When desired, Cia can generate some instances (examples) of proposed concept-descriptions in order to facilitate the recognition of the predicate.

Algorithm 8.1: Cia version of handle_uncovered.

```
procedure handle_uncovered( p(t1,...,tn): uncovered positive example, L: language bias )
    S := JUS(KB, L, p(t1,...,tn))
    c' := void
    repeat
        delete a clause c from S
        if c does not cover negative examples
        then c' := cia_part_1(c)
            if c' = void
            then c' := generalize_clause( c )
                derive a schema from c' and store it in the knowledge base
            endif
        endif
    until c' ≠ void or S = ∅
    if c' ≠ void
    then assert c' in KB
        call cia_part_2(c)
        for all negative examples n of p obtained in the calls to generalize_clause
            of the repeat loop such that φ2(KB, n) = true do
            call handle_covered( n )
        endfor
    else signal anomaly
    endif
endproc
```

Thus far, the basic algorithm of the Cia technique has been presented. Several improvements and variants are possible. The main objection against the use of the basic algorithm as it stands is that its pure syntactic nature gives rise to uninteresting as well as redundant questions (cf. also example 8.3).

Definition 8.8 *A question is redundant if it is logically equivalent to a previously asked question, i.e.*

- *two clause questions q_1 and q_2 are equivalent iff $KB \models (q_1 \leftrightarrow q_2)$*

Algorithm 8.2: Constructive Induction by Analogy

```
procedure cia_part_1( c : consistent clause ) : returns generalization of c
    d := void
    for all second-order schemata T do
        if match(T, c) with substitution Θ
        then if TΘ does not cover negative examples and d = void
            then ask the clause question TΘ to the oracle
                if τ2(TΘ) = true
                    if the oracle does not want to continue
                    then return TΘ
                    else d := TΘ
                    endif
                else find the substitution κ ⊆ Θ such that pmatch(T, c) for κ
                    ask the new term question Tκ to the oracle
                    if the oracle names the predicate variable in Tκ
                      by a substitution Λ
                    then assert TκΛ in the knowledge base
                    endif
                endif
            else find the substitution κ ⊆ Θ such that pmatch(T, c) for κ
                ask the new term question Tκ to the oracle
                if the oracle names the predicate variable in Tκ
                  by a substitution Λ
                then assert TκΛ in the knowledge base
                endif
            endif
        endif
    endfor
    return d
endproc

procedure cia_part_2( c : consistent clause )
    for all second-order schemata T do
        if pmatch(T, c) with substitution Θ and not match(T, c)
        then ask the new term question TΘ to the oracle
            if the oracle names TΘ by a substitution Λ
            then assert TΘΛ in the knowledge base
            endif
        endif
    endfor
endproc
```

- *two new term questions q_1 and q_2 are equivalent iff $KB \models (q_1\Theta \leftrightarrow q_2\Theta)$ where Θ is a second order substitution that replaces the predicate variables in q_1, q_2 by the same predicate name p, such that p does not yet occur in the knowledge base.*

In order to avoid such questions we augmented the algorithm with the following tests:

- Before asking a question, it is verified that it is not redundant. This is done by testing whether $T\Theta$, the concept to be proposed, is not equivalent to a previously proposed one.

  If we want to test whether question q_1 is redundant by comparing it to a previously asked question q_2, we can use two different tests :

  - A naive test which merely verifies: $\models (q_1 \leftrightarrow q_2)$ [2] This can e.g. be done using simple subsumption tests in two directions (see e.g. [Loveland 78])
  - A more complicated test which verifies $KB \models (q_1 \leftrightarrow q_2)$.

  For the second kind of test one can use a depth-bounded version of Buntine's subsumption technique [Buntine 88] (applied in two directions) or an extensional test to check equivalence. Buntine's test is quite fast and sound but unfortunately not complete, this means that all subsumptions found by it are legitimate but existing subsumptions will not always be found. The extensional test is complete (for non-recursive predicates without functors) but rather slow since it derives all ground instances that are covered by both descriptions. Of course, both tests are applied on the current knowledge base; in case there is still an error or an incompleteness in the knowledge base, they may give rise to wrong conclusions.

- Before proposing a new predicate, one must also be sure that it does not exist yet (it might have been in the original knowledge base or it might have been learned by Clint). This is verified in a similar way.

- Proposed predicate-definitions should be in a most simple form. A definition is in a most simple form, if it is impossible to properly generalize the clause without changing the semantics. As an example : the concept `grandfather(GF,GC)` ← `father(GF,C)` ∧ `parent(C,GC)` ∧ `male(GF)` is not in a most simple form since `male(GF)` can be deleted without changing the semantics of the clause. In order to know whether a clause is in a most simple form we build all minimal proper generalizations, and verify (using our subsumption test(s)) whether they are equivalent with the original clause or not.

Appart from asking relevant questions, some further improvements are possible :

- The same remark holds for constraints. If a new clause is proposed for a certain predicate, the constraints that are restrictive and applicable to that predicate may be verified. For similar reasons as for the eager method it is not interesting to test the other constraints because by adding clauses one generalizes.

[2] For clause questions, this form is retained. However, for new term questions we have to instantiate the heads of the proposed schema to the same predicate by applying a second order substitution Θ such that $q_2\Theta = q_1\Theta$ and Θ introduces a predicate name that is not already present in the knowledge base.

- Cia can be extended further by allowing transformations on learned schemata. The extension is motivated by the observation that when $\exists p,q,r : p(X,Y) \leftarrow q(X) \wedge r(X,Y)$ is a useful schema then it is quite likely that symmetric versions of the schema such as $\exists p,q,r : p(X,Y) \leftarrow q(Y) \wedge r(X,Y)$ are also useful. The second schema can be obtained by applying a transformation on the first one (replace the variables of unary predicates that also occur in the head by other variables occurring in the head). Several such transformations can be used in Cia and when Cia learns a new schema, all possible transformations are applied on it. If the result of such a transformation is a new schema that is not a renaming of an old one, it is stored in the memory. Because the use of transformations is less justifiable, from the analogy point of view, its use is optional.

- The schemata encode information about source problems. In the context of Clint-Cia, Clint may sometimes learn an incorrect clause from which Cia will derive a schema. However, when it is later detected that the clause, learned by Clint, is incorrect and the clause is retracted, we should also retract the schemata that are derived from the incorrect clause. The reason is that it can no longer be considered as a source problem. There is of course one exception to this rule, which occurs when the considered schemata can also be derived from other learned clauses that still remain in the knowledge base. This enchancement has been found useful in practice. It does not require much computation as one can index clauses under schemata.

Technical note

One quite technical optimization concerns the matching procedure with respect to the equality predicate `eq` (see also section 3.4). In order to avoid new term questions like $\exists p : p(X,Y) \leftarrow \texttt{male}(X) \wedge \texttt{eq}(X,Y)$ we allow the equality predicate only to match those predicate-variables of schemata whose source-instantiations were equality predicates. Although the equality predicate may only match such predicate-variables, it is allowed that predicate-variables with as source instantiation the equality predicate match non-equality predicates.

An example session with the integrated Clint-Cia system is shown in example 8.3.

Example 8.3: An example session with Clint-Cia

Suppose that we start with the following knowledge :

```
parent(alice,rose) ←          parent(leon,rose) ←
parent(rose,luc) ←            parent(rose,ann) ←
parent(etienne,luc) ←         parent(etienne,ann) ←
parent(laura, etienne) ←

female(alice) ←               male(luc) ←
female(rose) ←                male(etienne) ←
female(ann) ←                 male(leon) ←
female(laura) ←
```

In the next session, user input is in **bold** and system output typewritten. Furthermore, the Cia-part is surrounded by brackets <Cia Cia>.

Suppose Clint first learns the concept father.

```
?-learn.
Next example ?  @ father(etienne,luc).
Classification ?  @ no.
Starting clause for father(etienne, luc) in L0 :

      father(F,C) ← male(F) ∧  male(C) ∧  parent(F,C)

Is father(rose, luc) true ?  { male(F) } deleted @ no.
Is father(leon, rose) true ?  { male(C) } deleted @ yes.
Generalized clause :

      father(F,C) ← male(F) ∧  parent(F,C)

No more generalizations.  Asserted clause :

      father(F,C) ← male(F) ∧  parent(F,C)

<Cia :
Derived Schemata :
∃ : father, male, parent :  father(F,C) ← male(F) ∧  parent(F,C)
Schema number 1
and after applying transformation (1) :
∃ : father, male, parent :  father(F,C) ← male(C) ∧  parent(F,C)
Schema number 2
Cia>
```

The following transformation was used : if the head of the schema is of arity > 1 then derive a schema such that the variables of the unary predicates in the body are replaced by other ones; these variables also have to occur in the head.

```
?-learn.
```

```
Next example ?  @ grandmother(alice,luc).
Classification ?  @ true.
Starting clause for grandmother(alice,luc) in L0 :

      grandmother(G,C) ← male(C) ∧  female(G)

No more valid generalizations.  Asserted clause :

      grandmother(G,C) ← male(C) ∧  female(G)
```

There are no more generalizations in the language since deleting any of the two conditions would give rise to an unlinked variable (cf. the language assumptions of chapter 3).

```
<Cia
Derived Schemata :
∃ grandmother,male, female :  grandmother(G,C) ← male(C) ∧  female(G)
```

No transformations are applied since all transformations yield a symmetric variant.
Cia>

```
?-learn.
Next example ?  @ grandmother(rose,luc).
Classification ?  @ false.
Is female(rose) true ?  @ yes.
Is male(luc) true ?  @ yes.
Retracting incorrect clause :

      grandmother(G,C) ← male(C) ∧  female(G)

<Cia
Retracting Derived Schemata :
∃ grandmother,male, female :  grandmother(G,C) ← male(C) ∧  female(G)
```

Since there are no more instantiations of the schema in the knowledge base, it is also retracted from the knowledge base.
Cia>

```
True fact grandmother(alice,luc) is not covered
Starting clause for grandmother(alice,luc) in L0

      grandmother(G,C) ← male(C) ∧  female(G)

Starting clause covers false fact grandmother(rose, luc) :
Shifting bias to L0,1
Starting clause for grandmother(alice,luc) in L0,1

      grandmother(G,C) ← male(C) ∧  female(G) ∧  father(F,C) ∧
            parent(M1,C) ∧  parent(G,M2)

Starting clause covers false fact grandmother(rose, luc) :
Shifting bias to L0,1,0
Starting clause for grandmother(alice,luc) in L0,1,0 :
```

```
      grandmother(G,C) ← male(C) ∧ female(G) ∧ father(F,C) ∧
            parent(F,C) ∧ parent(M1,C) ∧ parent(G,M2) ∧
            eq(M1,M2) ∧ female(M1) ∧ male(F)

Is grandmother(leon,luc) true ? { female(G) } deleted @ no.
Is grandmother(alice, ann) true ? { male(C) } deleted @ yes.
Generalized clause :

      grandmother(G,C) ← female(G) ∧ father(F,C) ∧ parent(F,C) ∧
            parent(M1,C) ∧ parent(G,M2) ∧ eq(M1,M2) ∧
            female(M1) ∧ male(F)

Is grandmother(laura, luc) true ? { female(M1) } deleted @ yes.


      grandmother(G,C) ← female(G) ∧ father(F,C) ∧ parent(F,C) ∧
            parent(M1,C) ∧ parent(G,M2) ∧ eq(M1,M2) ∧ male(F)

No more valid generalizations. Asserted clause :

      grandmother(G,C) ← female(G) ∧ parent(M1,C) ∧
            parent(G,M2) ∧ eq(M1, M2)

<Cia
Proposing new predicates by partial matching ...
The starting clause for grandmother partially matches schema 1 :
∃ p : p(F,C) ← male(F) ∧ father(F,C)
This question is not in a most simple form
The starting clause for grandmother partially matches schema 1 :
∃ p : p(F,C) ← male(F) ∧ parent(F,C)
This question is redundant with father
The starting clause for grandmother partially matches schema 1 :
∃ p : p(F,C) ← female(F) ∧ parent(F,C)
Name p ? @ mother.
Asserting :

      mother(F,C) ← female(F) ∧ parent(F,C)

The starting clause for grandmother partially matches schema 2 :
∃ p : p(F,C) ← male(C) ∧ father(F,C)
Name p ? @ ignore.
```

In order not to have too many predicates in the knowledge base, the user ignores this predicate although the proposed predicate is a plausible one.

```
The starting clause for grandmother partially matches schema 2 :
∃ p : p(F,C) ← female(C) ∧ parent(F,C)
Name p ? @ daughter.
Asserting :

      daughter(F,C) ← female(C) ∧ parent(F,C)
```

```
The starting clause for grandmother partially matches schema 2 :
∃ p :  p(F,C) ← male(C) ∧  parent(F,C)
Name p ?  @ son.
Asserting :

      son(F,C) ← male(C) ∧  parent(F,C)

End of partial matching
Derived Schemata :
∃ grandmother, female, parent, eq :
grandmother(G,C) ← female(G) ∧  parent(M1,C) ∧  parent(G,M2) ∧  eq(M1,M2)
Schema 3
```

The system derives some more schemata by applying transformations and finally halts.

8.5 Experiments with Clint-Cia

Although the above example is quite simple, it gives a good idea of our Clint-Cia algorithm and its potential use. In order to better demonstrate Cia's utility we tested it on two extended learning tasks : one using the family domain and another one using a more complicated chess knowledge base.

8.5.1 The family domain.

In this experiment, we started from a knowledge base containing the predicates `parent`, `male`, `female` and `married`. The knowledge base consisted of data from 4 generations; in total there were about 140 facts. The aim was to learn the following 29 predicates: `father`, `mother`, `son`, `daughter`, `grandparent`, `grandson`, `granddaughter`, `grandfather`, `grandmother`, `grandparent_in_law`, `sibling`, `brother`, `sister`, `sibling_in_law`, `sister_in_law`, `brother_in_law`, `parent_in_law`, `mother_in_law`, `father_in_law`, `son_in_law`, `daughter_in_law`, `uncle_or_aunt`, `uncle`, `aunt`, `niece`, `nephew`, `cousin`, `female_cousin` and `male_cousin`. In total the definitions for these predicates consisted of 31 clauses. The only disjunctive predicate was `sibling_in_law` (3 clauses). We used the integrated Clint-Cia system for learning this knowledge base.

Both steps of Cia were applied and once a predicate was learned by Clint, all symmetric variants of the associated schema were asserted. Also, when an erroneous definition was learned by Clint and later retracted, the associated schema (and its variants) were retracted. Furthermore, the option in Clint to reduce the search space by querying the user for the relevant arguments (see example 4.7) was used to some of the irrelevant literals from the body of the starting clause. In our experiment we removed all relations from the justification that mentioned irrelevant objects. So, Clint proposed relations and depending on the nature of the objects in the justifications the relations were retained or discarded. Notice that although this strategy allowed us to discard many irrelevant relations from the explanation, many other irrelevant relations remained in the explanation. Suppose e.g. that you learn `grandfather(leon,luc)`

then the reduced justification would – depending on the known predicates – be based on `mother(rose,luc)`, `female(rose)`, `daughter(leon,rose)`, `parent(leon,rose)`, `son(rose,luc)`, `male(leon)`, `male(rose)`.

A good heuristic to minimize the interaction with Clint-Cia is to learn the basic (and most difficult) predicates first. So, we learned the predicates in the following order : `grandparent`, `sibling`, `parent_in_law`, `grandparent_in_law`, `sibling_in_law`, `cousin`. Afterwards it was easy to learn the other ones. Only three clauses were learned using the pure Clint system. These three clauses resulted in the following schemata :

$\exists$ `grandparent, parent, eq :`
`grandparent(G,C)` $\leftarrow$ `parent(G,M1)` $\wedge$ `parent(M2,C)` $\wedge$ `eq(M1,M2)` Schema 4

$\exists$ `parent_in_law, married, parent :`
`parent_in_law(G,C)` $\leftarrow$ `parent(G,M)` $\wedge$ `married(M,C)` Schema 5

$\exists$ `father,male, parent :`
`father(F,C)` $\leftarrow$ `male(F)` $\wedge$ `parent(F,C)` Schema 6

These schemata and their transformed variants were the only ones that were used by Clint-Cia. They proved sufficient to learn (or help learning) the other predicates. These other predicates were learned in basically two different ways. The most difficult ones (`sibling`, three clauses for `sibling_in_law`, `parent_in_law`, `grandparent_in_law`, and `cousin`) are only learnable in language $L_i, i > 0$ of Clint given the knowledge base and the order of learning the predicates. The used language series was $L_0, L_{0,1}, L_{0,1,0}$. For these predicates, Clint first had to shift the bias and then Cia could learn the correct definitions using its first step. The three clauses for `sibling_in_law` were learned by matching with only one clause. The remaining predicates, were directly derived from the starting clause in the first language of Clint (14 predicates) or invented by matching with this starting clause (8 predicates). Learning those clauses was very similar to the session in example 8.4.

During the learning process Cia proposed only 3 new predicates which were ignored by the user. They corresponded roughly to the following concepts : `mother_of_daughter`, `brother_of_sister`, `sister_in_law_of_brother_in_law`. Notice that they all represent meaningful concepts. Furthermore, 5 symmetric variants of desired predicate-definitions were proposed. With symmetric variants, we mean predicates that are equivalent but whose argument positions are interchanged. For example, if you define `father` in the normal way and define `father'` by `father(X,Y)` $\leftrightarrow$ `father'(Y,X)`, then `father` and `father'` are symmetric variants. Clearly, these questions can be avoided by allowing the user to interchange the argument positions of proposed new terms. Also, Cia avoided 8 redundant questions : either the corresponding concepts already existed or the questions were already posed or the concepts were not in a most simple form.

To summarize : using Clint-Cia, only 3 clauses were learned by the pure mechanism of Clint, 9 clauses were learned by Cia after a shift of bias by Clint, 14 clauses were learned directly by Cia and 8 new predicates were invented by Cia. This in contrast to learning the whole set of predicates with Clint which would have resulted in much more questions. With Clint, each clause is only obtained after several membership questions to the user.

I believe that an important reason for the success of this experiment is the use of the option in Clint of the relevant arguments. When this option would not have been used (or less irrelevant literals would have been identified), then Cia would have posed more questions. On the one hand, this would result in the invention of more predicates but on the other hand more proposed concepts would have to be ignored. I am well aware that the domain of family-relations is particularly well suited for our Cia technique since there are many (structurally) related predicates in it. Therefore, a more complicated learning task in the chess domain is presented below.

8.5.2 Learning simple chess rules

In this experiment[3], it was our aim to learn the rules that govern simple chess movements. The simple chess rules are the first kind of rules that one teaches a novel chess player. Roughly speaking, they state for each piece of chess which movement is allowed without taking into account the notion of chess and special rules such as *en passant* and *rocade*. For instance, for a rook, they state :

> a rook may move from tile t_1 to tile t_2 if t_1 and t_2 are on the same row or the same collumn and there is no piece standing in between t_1 and t_2 and either there is no piece on t_2 or a piece of the opponent.

For the other pieces there are similar rules. The rules for pawns and knights are particularly difficult. For pawns, we have :

- a pawn may move from tile t_1 to tile t_2 if t_1 is the starting position of the pawn and t_2 is two tiles away from t_1 in the forward direction[4]. Furthermore there is no piece between t_1 and t_2 and there is no piece on t_2.
- a pawn may move from tile t_1 to tile t_2 if t_1 is adjacent to t_2 in the forward direction and there is no piece on t_2.
- a pawn may move from tile t_1 to tile t_2 if t_1 is adjacent to t_2 in the forward diagonal direction and there is a piece of the opponent on t_2.

To learn the knowledge base, consisting of definitions of simple chess movements, we started from the knowledge base specified in example 8.4. The aim was then to learn the following predicates : `pawn_move/5`, `bishop_move/5`, `knight_move/5`, `rook_move/5`, `king_move/5` and `queen_move/5`. The argument structure of these predicates is the same:

> `x_move(A1,A2,A3,A4,A5)` succeeds iff piece `A1` that is standing on tile `A2` in boardsituation `A5` may move to tile `A4` that is occupied by piece `A3` (possibly the empty piece) in boardsituation `A5`.

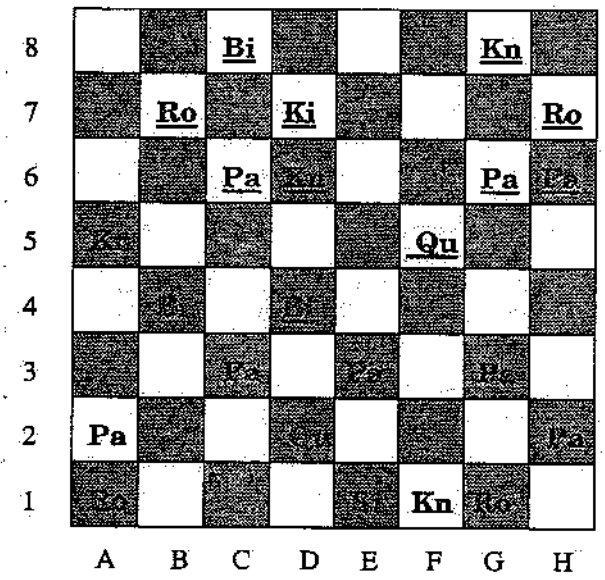
Furthermore, the system was given the following knowledge :

- when learning `x_move`, the condition `x(A1)` should appear in all clauses for `x_move`

[3]These experiments were carried out with the help of Hans Wouters.

[4]The directions have to be seen relative to the player. So, what is forward to one player is backward to the other.

Figure 8.2: A chess board



- when learning x_move, no condition y(A4) should appear in a clauses for x_move
- when learning x_move, the conditions occupies(A1,A2,A5) and occupies(A3,A4,A5) should appear in all clauses for x_move

This information was simply encoded in the language. Alternatively, this knowledge could have been stated by means of constraints. This was not done because this would have required more computation. Although this knowledge is – in no way – needed by the system to learn the given concepts, it was used because :

- The same kind of knowledge is also communicated to novel players when learning the same kind of rules.
- It makes all membership questions to the user relevant from the application point of view.[5]

The target-knowledge base consists of 21 clauses, which can all be described in the language of completely bound Horn-clauses (L_0). Of these 21 clauses, 7 are structurally different, i.e. 7 schemata are needed to make abstraction of these 21 clauses. Two schemata are sufficient to make abstraction of all clauses for rook_move, bishop_move, queen_move, king_move. Two additional schemata are needed to make abstraction of knight_move and for each clause of pawn_move an additional schema is needed. A clause for each schema is shown in example 8.5.

In this second experiment, the aim was to test Cia's first part and to see how well its guessing of concept-descriptions performs.[6] The following conventions were employed in the experiments :

- the normal version of Clint was used together with Cia's part one; no relevance questions about literals in justifications were posed.
- for each clause a schema was derived; this schema was retracted when the corresponding clause was retracted and there were no other learned clauses that justified the schema
- no transformations on schemata were used
- as soon as an incorrect clause was learned, a counter-example to that clause was presented to the system; this resulted in the retraction of the incorrect clause (and possibly its associated schema)

Using these conventions, two different experiments were performed : one learning the predicates in a simple order for Cia, i.e. starting by rook and ending with pawn and the other one in the reverse order. The results are shown in tables 1 and 2. The tables show : the predicate being learned, the number of clauses for the predicate, the number

[5] This point is related to the used knowledge representation. Given the knowledge representation there is no other way of preventing pieces to move from one boardsituation to another one or to move a piece p from a tile on which p does not stand to another tile. Apart from generating additional membership questions, there seems to be no other way around this problem, even when a different knowledge representation would have been used.

[6] In this application with the sketched knowledge base it is impossible to invent novel concepts; this is due to the additional knowledge that was integrated in Clint's language.

Example 8.4: A chess knowledge base to learn acceptable moves.

The knowledge base contains the following predicates:

- `rook/1`, `bishop/1`, `knight/1`, `queen/1`, `king/1` and `pawn/1` : these predicates succeed when their argument is a rook, bishop, knight, queen, king or pawn respectively. E.g. `rook(white(R))` is true.
- `opposite_color/2` succeeds when its arguments are pieces having a different color. E.g. `opposite_color(white(R),black(B))` is true.
- `board_no/1` succeeds when its argument is a number specifying a known chess positions. In order to learn, Clint was given a small number (8) of specific chess boards, which were all numbered. It is for these specific chess boards that Clint is supposed to learn and generate examples. One such situation is illustrated in figure 8.2.
- `occupies/3` succeeds when argument 2 is a location on the board, argument 1 is a piece and argument 3 is a board number such that the piece of argument 1 is located on position argument 2 of board argument 3. If there is no piece on the specified location, argument 1 yields the value `empty`.
- `empty/1` succeeds when its argument is the value `empty`.
- `same_row/2` succeeds when its two arguments are board locations that are in the same row. E.g. `same_row(g8,a8)` is true. There are similar definitions for `same_diagonal/2` and `same_column/2`.
- `nothing_in_between/3` succeeds when argument 1 is a board number, and argument 2 and 3 are locations on the board such that there is no piece in between the locations on the specified board. `nothing_in_between/3` only succeeds when the locations are on the same diagonal, row or collum.
- `startposition/2` succeeds when the first argument is a piece that is initially on the position specified by argument 2. E.g. `startposition(white(Pa),e7)` is true.
- `forward/3` succeeds when the first argument is a piece and the third argument is a position that lies beyond the second argument from the player's point of view, e.g. `forward(white(Pa),a2,a4)` succeeds.
- `adjacent/2` succeeds when its arguments specify adjacent locations on the chess board. E.g. `adjacent(e5,f4)` is true.
- `vertical_distance_1/2`, `vertical_distance_2/2`, `horizontal_distance_1/2`, `horizontal_distance_1/2` succeeds when the number of collums (resp. rows) between the locations is 1 (resp. 2). E.g. `vertical_distance_1(b4,e5)` is true.

Example 8.5: Clauses for schemata of the chess experiments

```
rook_move(A1,A2,A3,A4,A5) ← board_nr(A5) ∧ rook(A1) ∧
      same_row(A2,A3) ∧ occupies(A1,A2,A5) ∧ occupies(A3,A4,A5) ∧
      empty(A4) ∧ nothing_in_between(A5,A2,A3)

rook_move(A1,A2,A3,A4,A5) ← board_nr(A5) ∧ rook(A1) ∧
      same_row(A2,A3) ∧ occupies(A1,A2,A5) ∧ occupies(A3,A4,A5) ∧
      opposite_color(A1,A4) ∧ nothing_in_between(A5,A2,A3)

knight_move(A1,A2,A3,A4,A5) ← board_nr(A5) ∧ knight(A1) ∧
      occupies(A1,A2,A5) ∧ occupies(A3,A4,A5) ∧
      opposite_color(A1,A4) ∧ vertic_distance_1(A2,A3) ∧
      horiz_distance_2(A2,A3)

knight_move(A1,A2,A3,A4,A5) ← board_nr(A5) ∧ knight(A1) ∧
      occupies(A1,A2,A5) ∧ occupies(A3,A4,A5) ∧ empty(A4) ∧
      vertic_distance_1(A2,A3) ∧ horiz_distance_2(A2,A3)

pawn_move(A1,A2,A3,A4,A5) ← board_nr(A5) ∧ pawn(A1) ∧
      occupies(A1,A2,A5) ∧ occupies(A3,A4,A5) ∧ empty(A4) ∧
      startposition(A1,A2) ∧ same_collumn(A2,A3) ∧
      forward(A1,A2,A3) ∧ vertic_distance_2(A2,A3)

pawn_move(A1,A2,A3,A4,A5) ← board_nr(A5) ∧ pawn(A1) ∧
      occupies(A1,A2,A5) ∧ occupies(A3,A4,A5) ∧ empty(A4) ∧
      same_collumn(A2,A3) ∧ forward(A1,A2,A3) ∧
      vertic_distance_1(A2,A3)

pawn_move(A1,A2,A3,A4,A5) ← board_nr(A5) ∧ pawn(A1) ∧
      occupies(A1,A2,A5) ∧ occupies(A3,A4,A5) ∧
      opposite_color(A1,A4) ∧ same_diagonal(A2,A3) ∧
      forward(A1,A2,A3) ∧ adjacent(A2,A3)
```

of user-supplied examples, the number of generated examples by Clint, the number of good guesses by Cia, the number of bad guesses by Cia (this is divided into two different numbers : the first one gives the number of bad guesses before a good clause was guessed, the second one gives the number of guesses after a good clause was guessed; this difference is related to the option in Cia to continue after guessing a good clause; see also the algorithms) and the number of derived schemata from clauses for the predicate. In our context, because Cia part 2 was turned off, the option without continuation is the most relevant one.

Experiment 1	Ro	Bi	Qu	Ki	Kn	Pa	Total
clauses	4	2	6	2	4	3	21
user examples	4	2	6	2	5	3	22
clint examples	12	-	-	-	10	10	32
cia 1 good	2	2	6	2	1	1	14
cia 1 bad without continuation	-	2	2	1	-	12	17
cia 1 bad with continunation	-	-	-	3	-	8	11
schemata	2	-	-	-	3	2	7

Table 1 : Experimental results.

Experiment 2	Pa	Kn	Ki	Qu	Bi	Ro	Total
clauses	3	4	2	6	2	4	21
user examples	3	4	2	6	2	4	21
clint examples	15	8	7	-	-	-	30
cia 1 good	-	2	-	6	2	4	14
cia 1 bad without continuation	3	-	13	1	-	-	17
cia 1 bad with continunation	-	-	-	4	-	-	4
schemata	3	2	2	-	-	-	7

Table 2 : Experimental results.

These results indicate that the order of learning the predicates is not always important. Indeed, one would have expected that the first experiment would have required less clause questions than the second one because on the average the system knows less schemata. This is not the case in our experiments because :

- the size of the schemata corresponding to `rook_move` is shorter than the one for `bishop_move` and the ones for `pawn_move`.
- one special event occurred in the first experiment : one clause for `knight_move` that was learned by Clint contained a redundant condition, i.e. given our knowledge base of chessboards, it was a syntactic variant of the target clause. The syntactic variant was retained and the corresponding schema was successfully applied when learning `pawn_move`. This event also explains why 3 schemata were derived from `knight_move` and only two from `pawn_move` in the first experiment.

The experiments are quite successful. Indeed, the number of generated questions is – on the average – quite acceptable to learn a rather complicated knowledge base. If we only count the bad questions of Cia without continuation, we have a rate of success of about 45 percent for clause questions. If they are counted with continuation we get 1/3 for the first experiment and 2/5 for the second one.

Although the number of questions generated by Cia is – from the global point of view – quite acceptable, this is not the case at certain points : the learning of `pawn_move` in the first experiment and the learning of `king_move` in the second experiment. At those point many unsuccessful questions are generated. In both cases this can be explained by the fact that the size of the justification is relatively large as compared to the size of the known schemata.

If we estimate the number of membership questions that would have been required by Clint by taking the average number of generated membership questions for the clauses learned by Clint, we arrive at an expected number of generated membership questions of about 90. If we compare 90 membership questions to 32 (30) membership questions and 31 clause questions, we see that one clause question corresponds to about two membership questions.

The chess experiments also show that there are non-trivial domains in which Clint-Cia can be successfully applied. One might want to argue that our domain of chess is a bit artificial because there are so many structurally related clauses in it. This is revealed by the fact that if one would generalize `opposite_color` to also succeed when one of its arguments is the empty piece, one would also obtain less structurally related clauses. A similar modification could be made to generalize `same_collumn` and `same_row`. Although these modifications would indeed result in less successful experiments, this is no argument against our technique. The reason is very simple : if one can restructure the knowledge base in the proposed way, one can probably also write down the target-definitions. What I want to say here is that if one chooses to apply a learning system, one does not have enough knowledge to encode a knowledge base that will allow the system to derive optimal clauses. So, instead of being artificial, our chess domain appears quite natural for the kinds of applications that one would like to perform using Clint-Cia.

8.6 Evaluation of the experiments and Cia

8.6.1 A fundamental difficulty

Before evaluating our experiment with Clint-Cia, I want to point out a fundamental difficulty with evaluating systems as Cia. Although, it has been argued in the literature [Kibler and Langley 88] that machine learning is an experimental science and that machine learning techniques can be validated by carefully designed and controlled experiments, I believe that for certain techniques (of which Cia is one) this does not apply. The work on discovery systems like e.g. AM and Eurisco [Lenat 83, Lenat and Brown 83] has received much critique [Ritchie and Hanna 83] because there is no straightforward experiment nor a sound theoretical basis which allows to evaluate them. However, it is not doubted that the programs work and that AM and Eurisco are excellent pieces of artificial intelligence research. I believe that the same problem with evaluation methods holds for Cia. The only weak theoretical basis on which it relies is that schemata are useful abstractions of predicate definitions. This usefulness of schemata was also noted and exploited by sev-

eral other researchers (but never justified) such as [Emde, Habel and Rollinger 83, Morik 89, Wrobel 89, Thieme 89, Yokomori 86]. The only other justification we can provide for the Cia technique is a description of a working experiment. Performing systematic experiments is not feasible because the experiments depend too much on the learning situation and application. There is – for instance – no known generator of relevant knowledge bases.

Despite the fundamental difficulties involved in evaluating Clint-Cia, I believe that our experiments show that

> *the* Constructive Induction by Analogy *technique is useful at least for domains with many (structurally) related predicates.*

Furthermore, the experiments in the chess domain show that there are non-trivial domains that have structurally related predicates.

8.6.2 Asking less questions

In other domains, with less structure, the only problem might be that the system asks too much questions to the user or proposes uninteresting predicates. As far a the number of questions is concerned it is very important that Cia learns in the context of Clint. The reason why the number of proposed concept-descriptions is constrained from the start, is that Cia matches schemata only with justifications generated by Clint. These clauses have – usually – only a small number of conditions in their body. Another important property of these starting clauses is that they cover the positive example supplied by the user, which means that the clause can be fully instantiated such that all conditions in it are true. This property assures that all concepts proposed by Cia are meaningful existing concepts because there are known positive examples for these concepts.

This does not mean however that there may not be situations in which the system generates too much questions. There are many ways to ask less questions :

- First, one may want to use only the first part of Cia (this is an option in Cia). This will result in less questions since part 2 is the part in which most matches occur (see e.g. the chess experiments).
- Secondly, bad schemata should be excluded from the knowledge-base. E.g. if Clint learned `vehicle(X) ← car(X)`, it should not enter the derived schema in its knowledge-base, since this schema is too general to be useful. One way to avoid entering such schemata, is to ask the user whether the schema is interesting enough to be added into the knowledge-base. Another possibility is to measure the interestingness of the known schemata and retract the ones that fall below a certain treshold. Such a measure can take into account e.g. the rate of successful applications of the schema and the regularity of the schema. In the first case, one could award and punish schemata for each application, like is done with rules in e.g. Sage [Langley 85].
- Thirdly, the schemata may be refined such that they also take into account some of the semantics of the original predicate. Two promising refinements are :
  - adding information about the types of the arguments in the original predicate and

- use information about the integrity constraints of the predicate (this is related to e.g. number restrictions in KL-ONE [Brachman and Schmolze 85]).

As an example of this second possible extension, consider the `father` definition. For `father` we have that

- the types of `F` and `C` are related : the type of `F` is a subset of the type of `C` (all males are humans) and
- there is exactly one father and two parents for each `C`, while for each `F` there can be many `C`'s.

Refined schemata will match less clauses and hence they will give rise to less questions.

- Fourth, if there would be an explicit goal (like in [Wrobel 88]) for the learning system, then it would also be much easier to decide whether a question is useful or not. However, it is not clear how Clint-Cia can become goal- or demand driven.

8.6.3 Analogy

If we reconsider the framework of analogy from section 8.3, then it is clear that the Cia technique can be viewed as if it were performing analogy. First, although problems are specified by a positive example and a knowledge base, Cia acts on the starting clause computed from the positive example, the knowledge base and the language. So, problems are represented by starting clauses and solutions are clauses that are instantiations of particular schema.

- α(`Problem-1, Solution-1`) is satisfied if `Solution-1` is a clause learned by Clint for `Problem-1`;
- α'(`Problem-2, Solution-2`) is satisfied if `Solution-2` is the instantiated schema $T\Theta$ obtained by (partially or completely) matching the schema with the starting-clause;
- β'(`Solution-1, Solution-2`) is satisfied if both clauses corresponding to `Solution-1` and `Solution-2` are instantiations of the schema T
- β(`Problem-1, Problem-2`) is satisfied if both clauses (partially or completely) match the same schema T.

Some people may argue that Cia only performs a very weak form of analogy since only syntactical information is mapped from source to target and there is no model of causality taken into account[7]. In terms of the framework of [Kodratoff 90], our notion of analogy would be termed trivial. However, there is no reason why trivial analogies cannot be useful. First, augmented with more semantic information about the source problem, more information could be mapped (see example 8.7). Secondly, some researchers as e.g. G. Tecuci (personal communication) have argued that one could explain Cia in a causal

[7]The problems with viewing Cia as performing analogy were pointed out to us by Yves Kodratoff, Katharina Morik, Gheorge Tecuci and Stefan Wrobel

context. However, I feel that the point whether Cia performs analogy or not, is not really important and more a matter of terminology. The main reason why I explained Cia as performing analogy is that it was inspired in this way and also that it is a very convenient way to explain what happens[8]. Thirdly, the experiments with Cia have shown that some questions generated by Cia certainly involve analogy. This point is worked out in some detail in example 8.6.

Example 8.6: Analoguous questions generated by Cia

Let us consider the following clause generated by Clint :

```
rook_move(A1,A2,A3,A4,A5) ← board_nr(A5) ∧ rook(A1) ∧
      same_row(A2,A3) ∧ occupies(A1,A2,A5) ∧
      occupies(A3,A4,A5) ∧ opposite_color(A1,A4) ∧
      nothing_in_between(A5,A2,A3)
```

The corresponding schema was used by Cia to generate the following clause question :

```
bishop_move(A1,A2,A3,A4,A5) ← board_nr(A5) ∧ bishop(A1) ∧
      same_diagonal(A2,A3) ∧ occupies(A1,A2,A5) ∧
      occupies(A3,A4,A5) ∧ opposite_color(A1,A4) ∧
      nothing_in_between(A5,A2,A3)
```

Clearly those two clauses are very much related; the only difference being that `same_row` is replaced by `same_diagonal` and `rook` by `bishop`. Now, for everyone knowing the rules of chess it is known that a bishop is allowed to move along diagonals in a way that a rook is allowed to move along rows or collumns. This is also the way in which a bishop is analogous to a rook.

8.6.4 Towards an intelligent predicate editor

Example 8.6 does not only show that Cia sometimes realizes real analogy, but it also motivates the use of our Constructive Induction by Analogy technique in a different fashion. In the example, Cia might also have generated a different question, which corresponds to the proposed clause but with `same_diagonal` replaced by `vertic_distance_2`. Such a question might be generated when there is also a literal for `vertic_distance_2` in the starting clause. Clause questions related to other instantiations of the same schema can also be imagined.

Therefore, it seems interesting to present the clause questions corresponding to instantiations of one schema not as individual clause questions but as one more complicated, but more compact question. This question would then allow the user to edit the clause she wants. This is illustrated in example 8.7.

Observe that this method is also applicable when there are more than two instantiations of one schema. Furthermore, the proposed method generates at most one question

[8] I agree with Sigmund Freud in this respect.

Example 8.7: Editing clauses

Consider the previous example and the two clause questions :

```
bishop_move(A1,A2,A3,A4,A5) ← board_nr(A5) ∧ bishop(A1) ∧
      same_diagonal(A2,A3) ∧ occupies(A1,A2,A5) ∧
      occupies(A3,A4,A5) ∧ opposite_color(A1,A4) ∧
      nothing_in_between(A5,A2,A3)

bishop_move(A1,A2,A3,A4,A5) ← board_nr(A5) ∧ bishop(A1) ∧
      vertic_distance_2(A2,A3) ∧ occupies(A1,A2,A5) ∧
      occupies(A3,A4,A5) ∧ opposite_color(A1,A4) ∧
      nothing_in_between(A5,A2,A3)
```

These questions could be replaced by showing the user :

```
bishop_move(A1,A2,A3,A4,A5) ← board_nr(A5) ∧ bishop(A1) ∧
      (same_diagonal(A2,A3) or vertic_distance_2(A2,A3) ) ∧
      occupies(A1,A2,A5) ∧ occupies(A3,A4,A5) ∧
      opposite_color(A1,A4) ∧ nothing_in_between(A5,A2,A3)
```

and asking her which instantiation(s) she wants. In order to point out the analogy with previously learning clauses or concepts the source clause should be shown next to these questions

```
rook_move(A1,A2,A3,A4,A5) ← board_nr(A5) ∧ rook(A1) ∧
      same_row(A2,A3)
      occupies(A1,A2,A5) ∧ occupies(A3,A4,A5) ∧
      opposite_color(A1,A4) ∧ nothing_in_between(A5,A2,A3)
```

for each schema and exploits the analogy in a better way. Therefore, it may be even more feasible in practical methods than the original Cia technique.

8.6.5 Learning how to learn.

The integrated system Clint-Cia realizes constructive induction since it is able to invent new concepts. However, it also implements a kind of learning how to learn. Indeed, learning new concepts with Clint allows to derive (and use) new schemata and since the behaviour of Clint-Cia depends very much on the schemata, changing the set of schemata results in changing the learning. Hence, Clint-Cia adapts its learning : it gradually evolves from a pure data-driven learner (Clint) to a mixed model- and data-driven algorithm. One expects, that as more useful schemata are known, that this learning behaviour will improve (this was certainly the case in our experiments). One promising further aspect is that schemata (i.e. learning knowledge) acquired in one domain could also be used in other domains.

8.7 Related work.

The Cia technique was already presented in [De Raedt and Bruynooghe 89b, De Raedt and Bruynooghe 89c, De Raedt and Bruynooghe 90b]. However, the experimental results are extended here as the chess experiment is new.

The Cia technique is related to model driven learning in Blip [Morik 89, Emde, Habel and Rollinger 83, Wrobel 89], applying analogy on concept-learning [Vrain and Lu 88], analogy [Yokomori 86], constructive induction [Rendell 89, Muggleton and Buntine 88, Muggleton 87] and discovery [Lenat 83].

The second order schemata and the model-driven way of using them is very similar to the Blip learner [Morik 89, Emde, Habel and Rollinger 83, Wrobel 89]. However, Blip mainly uses the schemata to learn given concepts while Cia also uses them to invent new concepts. Also, the application conditions of a schema are completely different in Blip, where a heuristically restricted generate and test method is used, while in Cia a schema is matched against the starting-clause derived by Clint. The procedure to decide whether to accept or reject the instantiated schema is also different : Blip takes into account the number of positive and negative instances while Cia also relies on the user. In Clint-Cia it is not always possible to consider the positive and negative instances, since there may only be one example given. Blip can also invent new concepts in context [Wrobel 88]. In [Wrobel 88] the invention is demand-driven while in Cia it occurs as opportunities for learning new concepts arise. This learning in context allows to reduce the search-space. An important difference w.r.t. learning how to learn is that the schemata are automatically derived by Clint-Cia but not by Blip (but see [Thieme 89]). Also, Blip is a passive learner, which does not want to ask the user questions while Clint-Cia is an active learner, which asks many questions to the user.

In the much more general setting of logic programming the analogy between different predicate definitions was also noticed and exploited by [Yokomori 86]. He defines a notion of analogy of which ours seems to be a special case. He also indicates how it might be used to solve problems by analogy but he does not address issues of learning (and learning how to learn) and when to apply this analogical reasoning.

Cia uses the same paradigm as [Vrain and Lu 88] in which concept-learning is viewed as problem-solving and analogy is applied. However, Vrain and Lu apply analogy on a part of the concept-learning process (changing a concept-description for a certain example) whereas we apply it on whole concept-learning tasks.

Cia is also related to work on constructive induction [Muggleton and Buntine 88, Muggleton 87, Rendell 89, Mehra, Rendell and Wah 89]. However, this work on constructive induction has concentrated on discovering an appropriate vocabulary for reexpressing one particular given concept learning problem. Our approach differs from these approaches since it attempts to suggest the user valuable concepts to include in the knowledge base. Also, the suggested new concepts are not used for the given task but rather for future learning tasks. Seen in this light, Cia is closer to systems like AM and Eurisco [Lenat 83] which attempt to define interesting concepts. Nevertheless, the context, the heuristics and evaluation methods in the approach of Lenat and ours are completely different.

If we consider the integrated system Clint-Cia (see also [De Raedt and Bruynooghe 89c, De Raedt and Bruynooghe 89b, De Raedt and Bruynooghe 90b]) and view it in the light of the learning machine learning systems discussed by ([Stepp, Whitehall and Lawrence 88] in figure 2 page 338) we see that it possesses most of the characterizing features of learning machine learning systems :

- Cia allows to transform the representation of the back-ground knowledge;
- the concepts are organized in a hierarchy : one concept is built using the already known ones;
- the background knowledge is used to derive the starting-clause, hence it helps to compose the hypotheses;
- Cia discovers patterns or regularities in the learned concepts;
- the back-ground knowledge is updated : learned concepts are asserted into the knowledge base, this means that Clint-Cia is a closed-loop system, also knowledge about the learning (schemata) is asserted;
- in a sense Cia is able to recognize similar learning situations (when a schema matches a starting clause);
- the concept-language is augmented with previously learned concepts
- in a sense it can also be said to optimize its bias and its heuristics for more control, for two reasons
  - the combined effect of shifting the bias and a closed-loop learner (cf. also [Wrobel 88]) and
  - because it evolves from purely data-driven to more model-driven learning.

So, Clint-Cia has all characteristics discussed by Stepp et al. except for one, i.e. it does not chunk or transform knowledge into a more operational form. None of the systems discussed by Stepp et al. posses so many of these characteristics.

Chapter 9

Applications

9.1 Introduction.

In this chapter, I want to give a short and intuitive overview of some possible applications of the concept-learning environment Clint. On the one hand such an overview is indispensable to show the relevance of the theory. On the other hand, one should not expect such an overview to be complete and to thoroughly investigate all aspects of the outlined applications. This is beyond the scope of this overview as I merely want to demonstrate the applicability of the theory. Furthermore, applications are by nature domain dependent and often require an investigation of their own.

In the following sections, I will briefly show how Clint potentially contributes to

- program-synthesis
- learning apprentices
- autonomous agents
- knowledge acquisition and sloppy modeling
- knowledge base testing and verification

9.2 Program synthesis

The possibilities for using Clint for program-synthesis were already indicated in chapter 3, where it was also shown that the non-existence of a perfect query answering mechanism prevents the direct application of Clint to this area. However, given certain complexity bounds on considered terms and considered queries, it may be possible to come to an implementation of Clint that is able to synthesize programs for subsets of Prolog. Some preliminary, but promising results were already obtained in the Finix environment [Zinnsmeister and Hoppe 89].

One particularly relevant feature of such a program synthesizer based on Clint would be the possibility to use constraints while synthesizing programs. This would not only result in reducing the search-space and generating less and more intelligent questions but also in guaranteeing that the resulting programs are correct in certain respect. E.g.

when attempting to synthesize a sorting program such as `quicksort/2`, the use of constraints would allow to assure the correctness of the synthesized program. This could be achieved by supplying the constraints `quicksort(X,Y)` → `permutation(X,Y)` and `quicksort(X,Y)` → `sorted(Y)` where a correct definition of `sorted` and `permutation` should be known. As far as I know, the use of constraints while synthesizing programs from examples, is new.

9.3 Learning apprentices.

As outlined in chapter 1, one of the main design goals behind Clint was to realize an inductive learner that would be applicable in the learning apprentice generation of knowledge based systems.

To show that this design goal has been achieved, I will now argue that Clint could be successfully applied in the context of the learning apprentice Disciple [Tecuci and Kodratoff 90, Kodratoff and Tecuci 87, Tecuci 88].

The system Disciple contains an empty expert system shell and a multi-strategy learner. Disciple is an intelligent aid, which assists an expert while solving problems. It has been applied on the design of loudspeakers amongst others [Tecuci and Kodratoff 90].

Disciple's knowledge base consists basically of two parts :

- knowledge about the objects of the domain; this involves facts as well as general inference rules
- knowledge about actions that can be applied on objects

To solve a problem, the expert can apply certain actions. These actions can be of two different kinds : actions that decompose a goal into a set of subgoals that are easier to solve or primitive actions that achieve a certain effect. An action may only be applied when certain preconditions are satisfied. Disciple starts from an incomplete and possibly also incorrect knowledge base. While solving problems Disciple will observe the expert at work and improve its knowledge base when possible.

To improve the knowledge base, Disciple has three different learning modes, which correspond to different development stages of its knowledge base : learning in a complete theory, which corresponds to applying explanation based learning [Mitchell, Keller and Kedar-Cabelli 86, De Jong and Mooney 86]; learning in a weak theory, where basically empirical learning techniques are employed and learning in an incomplete theory, where a mixture of both other techniques is used.

In its weak theory mode, Disciple behaves quite similar as Clint (or Clint very similar as Disciple) : it generates questions to the user to find good justifications and learns preconditions of actions by generating examples. Clearly, Clint could also be used in such a context. Furthermore, certain techniques of Clint may be useful to extend the capabilities of Disciple. In particular, I think of the following issues :

- As pointed out by [Tecuci and Kodratoff 90], it may in certain situations be desirable to extend the justifications found by Disciple, i.e. to shift the bias, in a similar way as Clint does when it discovers that its current knowledge or language does not allow to find a satisfactory justification.[1]

[1] Justifications in our framework correspond to explanations in the framework of Disciple.

- Also, in the context of Disciple, it may be interesting to use meta-knowledge in the form of constraints.

- Although Disciple is able to acquire knowledge concerning new objects, it only acquires such knowledge by asserting facts that it is told by the user. Under certain conditions, it may be desirable to employ our method to acquire fact knowledge in Disciple.

Although it may be a good idea to incorporate some features of Clint in Disciple, the reverse is also true and has – in part – already been done. Indeed, many ideas in Clint were inspired on Disciple during my stay in Yves Kodratoff's group in Orsay. I think for instance about the example generation and the method to reduce the size of the starting clauses. Disciple incorporates more than one learning mode and is also an expert system shell. It would be interesting to see how Clint would behave as a component of such an expert system shell.

A system that is related to Disciple it the Odyseus approach of [Wilkins 89]. Odyseus attempts to understand actions undertaken by experts. When it cannot explain why a certain action was undertaken it will attempt to acquire missing rules. In order to do so, it will analyse examples, generate a set of rules, and select a plausible one for insertion in the knowledge base. Also in this context Clint may be of use.

9.4 Autonomous agents

In the previous section, we saw how Clint may be used in a learning apprentice. It was also shown that Clint – in a way similar to Disciple – can be applied to learning preconditions of actions. One cannot only apply actions in expert systems such as Disciple but also in planners such as e.g. Strips [Fikes and Nilsson 71]. In the undergraduate thesis of Patrick Lambrix [Lambrix 90], it was studied how to integrate Clint into a Strips-like planner with the aim of learning preconditions of actions.

Let us briefly summarize Lambrix's work. Lambrix's planner starts from Strips-like actions, i.e. actions whose pre- and post-conditions are represented by add and delete-lists. Lambrix's planner starts from incompletely specified actions. The specified actions are incomplete in the sense that the known pre-conditions of actions may be incorrect or incomplete. To cope with incomplete pre-conditions, the planner is equipped with the capability to detect when an action goes wrong. This occurs when the expected post-conditions of an action differ from the realized ones. When an action goes wrong, Lambrix's planner attempts to learn the correct preconditions. In order to do so, it employs Clint in the following manner :

- The pre-conditions of an action can be represented by a predicate not(p). So, the pre-conditions are satisfied when the predicate p does not hold.

- When an action goes wrong, the situation before the action was applied is a negative example for the pre-conditions of the actions. This corresponds to a positive example for the predicate p .

- When an action succeeds, the situation before the action was applied is a positive example for the pre-condition and a negative example for the predicate p.

- The learning process is triggered when an action goes wrong. In that case, the corresponding example is passed to Clint, which derives a starting clause for the predicate p using the facts that were true before applying the action.
- To generate experiments, Clint derives a generalization g of the starting clause c (or of a generalization of that starting clause) using its normal generalization procedure. Clint then passes the following goal to the planner : $g \wedge \neg c$.
- The planner attempts to generate a plan to achieve this goal. There are several cases to consider :
  - the planner is unable to generate such a plan : in this case Clint backtracks and attempts to generate another goal.
  - the planner generates a plan and the agent succeeds in executing the plan, i.e. no action in the generated plan goes wrong. After executing this plan, the agent executes the original action a second time. This results in an additional example for the same pre-conditions.
  - the planner generates a plan but during execution of the plan one of the actions goes wrong : in this case, the learning of the pre-conditions of the original action is put on an agenda mechanism, and the pre-conditions of the last action are learned. After learning these actions, the agenda is popped and the agent continues learning the preconditions of the original action.

This briefly sketches Lambrix's approach to learning and planning. For more details concerning the theoretical feasibility and the avoidance of deadlocks see [Lambrix 90].

Lambrix's approach clearly shows that Clint can be successfully applied in autonomous agents. An interesting research issue is whether Clint could also be applied in Lambrix's planner to learn post-conditions. Related problems have been studied in the context of the system Prodigy [Gil and Carbonell 87].

9.5 Knowledge acquisition and sloppy modeling

Katharina Morik [Morik 89] presents an interesting unifying view on knowledge acquisition and machine learning. She states that,

> knowledge acquisition systems should support the user in doing the modeling of the domain, and machine learning systems are those which perform part of the modeling autonomously.

This principle forms the basis of the sloppy modeling paradigm that has been incorporated in the Blip system [Morik 89, Wrobel 89, Thieme 89, Emde 89, Kietz 88]. Blip has a number of interesting features :

- Using Blip the user can input certain information about the model she has in mind.
- From the input information, Blip will attempt to derive a theory. This is achieved by a learning by observation module that derives rules from facts.

- The implications of modifications to the model (or knowledge base) are always shown to the user, who can take appropriate action to make unwanted implications undone.

- An important feature of Blip is its reversibility. All decisions made by the system or the user can be undone. This is of particular importance for the knowledge representation. Indeed, [Morik 89] shows that – during the modeling phase – it occurs frequently that the knowledge representation format has to be revised. Such revisions are fully supported by Blip.

Blip is a passive learner, which does not ask questions to the user. However, it notifies the user of the implications of any action. Because it also supports reversibility of all decisions, it seems very interesting to integrate Blip and Clint. This would result in several advantages :

- Clint sometimes suffers from the irreversibility of previous decisions concerning the knowledge representation. Using Blip such revisions would be reversible.

- For Clint, it would also be interesting to show the user the implications of the proposed generalizations and learned clauses. This would allow to more quickly detect remaining errors.

- For Blip, which is a passive learner, it is also interesting to have the ability to learn by asking questions. Indeed, one can imagine situations, in which the user does not know in which direction to proceed. In such situations, an active systematic learner such as Clint must be of interest.

In an integrated Blip-Clint-system, the user could decide at any point whether she wants to proceed by herself or whether she is prepared to answer questions.

9.6 Testing, verification and validation

In this section, we describe how the interactive concept-learning environment Clint can be used for knowledge base validation, verification and testing (see also [De Raedt and Bruynooghe 90e])[2]. There are two main contributions of Clint to this area :

1. Clint can be used to generate in a systematic way relevant experiments; these experiments can be considered as tests, which validate the knowledge base if they succeed

2. if an experiment does not succeed, or in case an error is signaled by the user, Clint can be used as an intelligent debugger, which locates the source of the error and replaces the incorrect rules by new rules that are automatically constructed from the experiments and the existing knowledge base.

[2]The ideas presented in this section were inspired by suggestions of Katharina Morik and Thomas Hoppe and also by listening to a talk of Yves Kodratoff.

9.6.1 Using Clint for generating experiments.

Clint can be used to systematically generate experiments — in the form of examples. These examples constitute tests whether the given knowledge base is correct.

Before presenting an algorithm let us first recall the notion of a dependency graph for predicates.

Definition 9.1 *A predicate p directly depends on a predicate q iff q occurs in a clause of the definition for p.*

The relation *depends on* is the transitive closure of the relation *directly depends on* (see section 3.2 for the definition of depends on). This relation organizes the predicates of a certain knowledge base in a graph structure.

Following a bottom up approach to validate a knowledge base, we first test the predicates which depend on no other predicates, then the ones which directly depend on the former ones, and so on ... The dependency graph determines the order in which the predicates will be tested by our approach.

In order to verify a knowledge base KB, a knowledge base KB' which contains the same knowledge, will be learned. For testing a predicate p in KB, a predicate p' in KB' will be learned. Before testing p, we assume that all predicates q on which p depends have already been validated[3]. This means that for all these predicates q there is a semantically equivalent definition q' in KB'. (Two definitions are semantically equivalent if they cover the same set of examples. This can e.g. be verified by the tests described in [Buntine 88]).

In order to learn p' in KB', we initialize the definition of p' in KB' to the empty definition. Each time Clint has a definition for p' in KB', an example e is generated that is covered by p in KB and not by p' in KB' or by p' in KB' and not by p in KB. The example e then constitutes a test on which further learning can be based and from which further tests can be generated. If at a certain moment in time an example is generated (by Clint or by the above method to decrease the differences between p and p'), which the user classifies as true (resp. false) while according to the definition for p it should be false (resp. true), an error has been found in the knowledge base KB.

As long as no error has been found, the system continues learning until a definition for p' in KB' is learned that is equivalent to p in KB. At that time we have strong indications that p and p' are correct since they agree on crucial examples. Then p is considered to be validated. Provided that the knowledge base KB obeys the syntactic restrictions imposed by the languages of Clint, this verification is sound because of the identification in the limit property of Clint. Of course, during the learning process integrity constraints can be taken into account. In case one only wants to validate or verify a certain predicate r, the above method can be straightforwardly modified (copy all predicates t in KB on which r depends, to t' in KB').

9.6.2 Modifying the knowledge base.

If during the above testing phase, errors in the knowledge base KB are found, Clint can be used to propose modifications to the knowledge base KB. This can be done by learning

[3] In the case of mutually recursive predicates this is impossible. Then one has to validate the predicates that are mutually recursive together.

the predicate p' in KB' using the techniques of Clint. After p' has been learned, it should be validated. For this validation, we can use the same method but with different examples! One should not use the original examples, since otherwise one will arrive at the same tests and definitions and perform no validation at all.

9.6.3 Using integrity constraints.

An additional feature of Clint that is certainly of interest in the validation application is the use of integrity constraints. While examples correspond to tests that are concerned with a kind of local correctness, constraints constitute tests for a more global kind of correctness.

Constraints can easily be used in the proposed framework for validation and verification. To use constraints one only has to declare these constraints in the knowledge base KB' and work as usual with Clint.

Of course, one may also want to start the verification process by testing the constraints. If this is desired one should copy the whole knowledge base KB to KB' and supply the constraints to KB'.

9.6.4 Summary.

We have briefly sketched how a technique based on interactive concept-learning can be used to systematically validate, verify and test a knowledge base or part of it. The main advantage of the proposed technique is that Clint systematically generates relevant tests for the knowledge base. If errors are found during verification or testing, concept-learning techniques can be used to propose modifications to the given knowledge base. The proposed method is quite general, as it is applicable to knowledge bases which satisfy the syntactic restrictions imposed by the languages of Clint, and integrity constraints can be used.

9.7 Conclusion

In this chapter I have tried to show that the concept-learning environment Clint and the techniques behind it, are of use in a wide variety of applications. This proves that the main goal of this thesis, the development of a sound basis for interactive concept-learning, has been achieved. Although it has been indicated how Clint or the techniques behind Clint are useful in a wide variety of applications, further research is necessary to turn Clint into a practical tool for these applications.

Chapter 10

Conclusions

To conclude, let us briefly summarize the main contributions of our work and present some possible directions for future research.

10.1 Summary of the Achievements.

The contributions of this thesis are twofold. On the one hand, we have developed a general framework for concept-learning and on the other hand, a novel theory and methodology of interactive concept-learning has been presented.

Our first contribution is concentrated around the generic algorithm Gencol. Gencol is generic in the sense that it makes abstraction of the most important issues of concept-learning:

- the search-strategy
- the description languages of concepts and examples
- heuristics
- the operations that are allowed on hypotheses
- the way examples are supplied to the system
- the quality and the stop criteria

Although Gencol is a quite detailed algorithm, it allows to make abstraction of most known approaches to symbolic concept-learning. This was shown by proving how 5 seemingly disparate algorithms are an instantiation of Gencol. I argue that Gencol contributes to the theory, practice and didactics of concept-learning. From the point of view of theory, it provides a general framework for comparing, evaluating and understanding concept-learning systems. For the same reason Gencol is interesting from a didactical point of view as it can be used (and has already been used) as a thorough introduction to the most important aspects of concept-learning. Gencol is also interesting from a practical point of view since it clearly shows which decisions have to be made when developing a new concept-learner. For some applications, it is even possible to use a tool based on Gencol in order to automatically derive prototype concept-learning systems.

The second and most important contribution of this thesis was the introduction of an integrated framework and system to interactively learn concepts from examples. In chapter 3, we presented the basic algorithms of Clint and extensively studied its properties. In the next chapters it was shown how these basic algorithms could be extended with other interesting features:

- the ability to shift the bias and to cope with indirect relevance
- the possibility to specify meta-knowledge in the form of constraints
- the acquisition of fact knowledge concerning new objects
- the use of a multi-valued logic for concept-learning
- the Constructive Induction by Analogy technique.

Although the presented techniques are integrated in the Clint environment, I believe that most of them are interesting in their own right and applicable to other concept-learning approaches. This is certainly the case for the meta-knowledge component, the approach to knowledge acquisition and the use of a multi-valued logic. Besides these individual contributions, I think one of the most significant aspects of the presented framework is the integration of many interesting features into a working whole. In the learning apprentice context, which imposes high demands with regard to user-friendliness and opportunism, this is certainly of use. The fact that these features could be integrated in a smooth way into Clint is – in my opinion – largely due to the soundness of the basic algorithms of Clint. Without having a sound basic algorithm of Clint, it would have been much harder to integrate these features into Clint. Also, the discovery of these features would have been more complicated. I believe that this justifies our theoretical investigations of the basic algorithm of Clint. It also provides hope that additional features may be incorporated into our framework and that it will be relatively easy to apply Clint in practical applications.

10.2 Future research.

Personally, I see three interesting directions for further research. These are concerned with building applications of the framework, using inverse resolution to generalize and incorporating some notion of uncertainty into the framework.

The possibilities for applying the framework in practice has already been discussed in chapter 9. For one of these applications, i.e. the autonomous agent, the first step towards an implementation has already been made. It would be interesting to work towards implementations of the other proposed applications.

Recently, [Muggleton and Buntine 88] developed a logical theory of induction. The theory is based on two observations :

- induction is the opposite of deduction
- most approaches to deduction are based on resolution

Therefore they attempt to invert the resolution principle and hope that this inverted resolution principle can be used as a sound basis for induction. Although this idea is fascinating and some interesting first results have been obtained, there are still many open

questions concerning inverse resolution. Nevertheless, it would be an interesting research project to see whether the issues involved in Clint such as the generation of examples, the identification in the limit and the constraint handling, could be retained when using inverse resolution as the generalization transformer in Clint. Such an approach would allow Clint to optimize its search as inverse resolution uses a semantical theory of generalization instead of a syntactical one. This is for instance useful to avoid the generation of syntactic variants. At the moment, however, I believe that there are too many remaining open questions concerning inverse resolution, in order to apply it directly to our framework. To mention one : it is not clear whether inverse resolution allows to generate all clauses in a given language and if so whether it does not have to rely on boundary cases of inverse resolution which are roughly equivalent to using our framework.

A third major challenge would be to adapt Clint in order to work with probabilities. At the moment Clint can work with a multi-valued logic. For certain applications this is not sufficient. There are many interesting applications domains such as e.g. medicine, where rules are never correct or incorrect, but rather correct in a certain percentage of the cases. It would be a major challenge to try to extend Clint in order to cope with probabilities. In such extensions, the user could for instance supply constraints to Clint that have to be correct in x percent of the cases.

Bibliography

[Angluin 87] Dana Angluin. Queries and concept-learning. *Machine Learning*, 2:319–342, 1987.

[Angluin and Smith 83] D. Angluin and C.S. Smith. Inductive inference : theory and methods. *Computing Surveys*, 15:237–269, 1983.

[Baes 90] E. Baes. Een kennisgebaseerde benadering van pionneneindspelen in het schaakspel. Master's thesis, Department of Computer Science, Katholieke Universiteit Leuven, 1990. in Dutch.

[Biermann 86] A. Biermann. Fundamental mechanisms in machine learning and inductive inference. In W. Bibel and P. Jorrand, editors, *Fundamentals of Artificial Intelligence*, volume 232 of *Lecture Notes in Computer Science*. Springer-Verlag, 1986.

[Brachman and Schmolze 85] R.J. Brachman and J.G. Schmolze. An overview of the kl-one knowledge representation system. *Cognitive Science*, 9:171–216, 1985.

[Bratko 86] I. Bratko. *Prolog Programming for Artificial Intelligence*. Addison-Wesley, 1986.

[Bry 90] Francois Bry. Intensional updates : abduction via deduction. In *Proceedings of the 7th International Conference on Logic Programming*. The MIT press, 1990.

[Buchanan and Mitchell 78] B.G. Buchanan and T.M. Mitchell. Model-directed learning of production rules. In D.A. Waterman and F. Hayes-Roth, editors, *Pattern-directed inference systems*. Academic Press, 1978.

[Bundy, Silver and Plummer 85] A. Bundy, B. Silver, and D. Plummer. An analytical comparison of some rule-learning programs. *Artificial Intelligence*, 27:137–181, 1985.

[Buntine 87] Wray Buntine. Induction of horn-clauses : methods and the plausible generalization algorithm. *International Journal of Man-Machine Studies*, 26:499–520, 1987.

[Buntine 88] Wray Buntine. Generalized subsumption and its application to induction and redundancy. *Artificial Intelligence*, 36:375–399, 1988.

[Carbonell, Michalski and Mitchell 83] J.G. Carbonell, R.S. Michalski, and T.M. Mitchell. An overview of machine learning. In R.S. Michalski, J.G. Carbonell, and T.M. Mitchell, editors, *Machine Learning : an artificial intelligence approach*, volume 1. Morgan Kaufmann, 1983.

[Ceri, Gottlob and Tanca 90] S. Ceri, G. Gottlob, and L. Tanca. *Logic programming and databases*. Springer-Verlag, 1990.

[Cestnik, Kononenko and Bratko 87] B. Cestnik, I. Kononenko, and I. Bratko. Assistant 86 : a knowledge-elicitation tool for sophisticated users. In *Proceedings of the 2nd European Working Session on Learning*, pages 31–45. Sigma Press, 1987.

[Chan 88] D. Chan. Constructive negation based on the completed database. In *Proceedings of the 5th Logic Programming Symposium*, pages 111–125. Morgan Kaufmann, 1988.

[Chouraqui 85] E. Chouraqui. Construction of a model for reasoning by analogy. In L. Steels and J.A. Campbell, editors, *Progress in Artificial Intelligence*. Ellis Horwood, 1985.

[Cohen and Feigenbaum 81] P.R. Cohen and E.A. Feigenbaum, editors. *The handbook of artificial intelligence*, volume 3. Morgan Kaufmann, 1981.

[De Jong 88] K. De Jong. Learning with genetic algorithms. *Machine Learning*, 3:121–138, 1988.

[De Jong and Mooney 86] G. De Jong and R. Mooney. Explanation based learning : an alternative view. *Machine Learning*, 1:145–176, 1986.

[De Raedt and Bruynooghe 88] L. De Raedt and M. Bruynooghe. On interactive concept-learning and assimilation. In D. Sleeman, editor, *Proceedings of the 3rd European Working Session On Learning*, pages 167–176. Pitman, 1988.

[De Raedt and Bruynooghe 89a] L. De Raedt and M. Bruynooghe. Concept-learning as search revisited : a generic algorithm. Technical report, Katholieke Universiteit Leuven, Department of Computer Science, 1989. submitted.

[De Raedt and Bruynooghe 89b] L. De Raedt and M. Bruynooghe. Constructive induction by analogy : a method to learn how to learn ? In *Proceedings of the 4th European Working Session On Learning*, pages 189–200. Pitman, 1989.

[De Raedt and Bruynooghe 89c] L. De Raedt and M. Bruynooghe. Constructive induction by analogy. In *Proceedings of the 6th International Workshop on Machine Learning*, pages 476–477. Morgan Kaufmann, 1989.

[De Raedt and Bruynooghe 89d] L. De Raedt and M. Bruynooghe. On explanation and bias in inductive concept-learning. In *Proceedings of the 3rd European Knowledge Acquisition for knowledge based systems Workshop*, pages 338–353, 1989.

[De Raedt and Bruynooghe 89e] L. De Raedt and M. Bruynooghe. Towards friendly concept-learners. In *Proceedings of the 11th International Joint Conference on Artificial Intelligence*, pages 849–856. Morgan Kaufmann, 1989.

[De Raedt and Bruynooghe 90a] L. De Raedt and M. Bruynooghe. Indirect relevance and bias in inductive concept-learning. *Knowledge Acquisition*, 2:365–390, 1990.

[De Raedt and Bruynooghe 90b] L. De Raedt and M. Bruynooghe. Interactive concept-learning and constructive induction by analogy. Technical report, Department of Computer Science, K. U. Leuven, 1990. submitted.

[De Raedt and Bruynooghe 90c] L. De Raedt and M. Bruynooghe. On negation and three-valued logic in interactive concept-learning. In L.C. Aiello, editor, *Proceedings of the 9th European Conference on Artificial Intelligence*, pages 207–212. Pitman, 1990.

[De Raedt and Bruynooghe 90d] L. De Raedt and M. Bruynooghe. Unifying incremental concept-learning and intentional knowledge base updating. Technical report, Department of Computer Science, K. U. Leuven, 1990. submitted.

[De Raedt and Bruynooghe 90e] L. De Raedt and M. Bruynooghe. Using interactive concept-learning for knowledge base validation, verification and testing, 1990. abstract presented at the ECAI-workshop on validation, verification and testing of knowledge-based systems, Stockholm.

[De Raedt, Feyaerts and Bruynooghe 91] L. De Raedt, J. Feyaerts, and M. Bruynooghe. Acquiring object-knowledge for learning systems. In Yves Kodratoff, editor, *Proceedings of the 5th European Working Session on Learning*, 1991. to appear.

[De Raedt, Krekels, Bruynooghe and Van Meir 87] L. De Raedt, B. Krekels, M. Bruynooghe, and D. Van Meir. Using shapiro's model inference system for concept-learning. In *Proceedings of the fourth international conference on Artificial Intelligence and Control Systems of Robots*, pages 191–196. Smolenice, 1987.

[De Raedt, Martens and Bruynooghe 90] L. De Raedt, B. Martens, and M. Bruynooghe. A practical multi-valued logic to reason about the unknown and the inconsistent (draft). Technical report, Department of Computer Science, K. U. Leuven, 1990.

[Declercq 90] E. Declercq. Een uitwerking van het generisch algoritme gencol dat concepten leert. Master's thesis, Department of Computer Science, Katholieke Universiteit Leuven, 1990. in Dutch.

[Dietterich and Michalski 83] T.G. Dietterich and R.S. Michalski. A comparative review of selected methods for learning from examples. In R.S. Michalski, J.G. Carbonell, and T.M. Mitchell, editors, *Machine Learning : an artificial intelligence approach*, volume 1. Tioga Publishing Company, 1983.

[Emde 89] W. Emde. An inference engine for multiple theories. In K. Morik, editor, *Knowledge Representation and Organization in Machine Learning*, volume 347 of *Lecture Notes in Artificial Intelligence*. Springer-Verlag, 1989.

[Emde, Habel and Rollinger 83] W. Emde, C.U. Habel, and C.R. Rollinger. The discovery of the equator or concept driven learning. In *Proceedings of the 8th International Joint Conference on Artificial Intelligence*, pages 455–458. Morgan Kaufmann, 1983.

[Feyaerts 89] J. Feyaerts. Twee methodes voor het vergaren van object-kennis. Master's thesis, Department of Computer Science, Katholieke Universiteit Leuven, 1989. in Dutch.

[Fikes and Nilsson 71] R.E. Fikes and N.J. Nilsson. Strips : a new approach to the application of theorem proving to problem solving. *Artificial Intelligence*, 2:189 – 208, 1971.

[Gaines and Boose] B. Gaines and J. Boose. The knowledge acquisition journal. Academic Press.

[Gams and Lavrac 87] M. Gams and N. Lavrac. Review of five empirical learning systems within a proposed schemata. In *Proceedings of the 2nd European Working Session on Learning*, pages 46–66. Sigma Press, 1987.

[Genesereth and Nilsson 87] M. Genesereth and N. Nilsson. *Logical foundations of artificial intelligence*. Morgan Kaufmann, 1987.

[Gil and Carbonell 87] Y. Gil and J.G. Carbonell. Learning by experimentation. In *Proceedings of the 4th Machine Learning Workshop*. Morgan Kaufmann, 1987.

[Gold 67] E.M. Gold. Language identification in the limit. *Information and control*, 10:447–474, 1967.

[Gordon and Perlis 89] D. Gordon and D. Perlis. Explicitly biased generalization. *Computational Intelligence*, 5:57–81, 1989.

[Grosof and Russell 90] B. Grosof and S. Russell. Shift of bias as non-monotonic reasoning. In P.B. Brazdil and K. Konolidge, editors, *Machine Learning, Meta-Reasoning and Logics*. Kluwer Academic Publishers, 1990.

[Guessoum and Lloyd 90] A. Guessoum and J.W Lloyd. Updating knowledge bases. *New Generation Computing*, 8:71–88, 1990.

[Haas and Hendrix 83] N. Haas and G. Hendrix. Learning by being told : acquiring knowledge for information management. In R.S. Michalski, J.G. Carbonell, and T.M. Mitchell, editors, *Machine Learning : an artificial intelligence approach*, volume 1. Morgan Kaufmann, 1983.

[Hayes-Roth and McDermott 78] F. Hayes-Roth and J. McDermott. An interference matching technique for inducing abstractions. *Communications of the Association for Computing Machinery*, 21:401–410, 1978.

[Hirsh 90] H. Hirsh. Knowledge as bias. In P.D. Benjamin, editor, *Change of representation and inductive bias*. Kluwer Academic Publishers, 1990.

[Kibler and Langley 88] D. Kibler and P. Langley. Machine learning as an experimental science. In *Proceedings of the 3rd European Working Session On Learning*, pages 81–92. Pitman, 1988.

[Kietz 88] J.U. Kietz. Incremental and reversible acquisition of taxonomies. In M. Linster, B. Gaines, and J. Boose, editors, *Proceedings of the 2nd European Knowledge Acquisition for Knowledge Based Systems Workshop*, 1988.

[Kodratoff 88] Y. Kodratoff. *Introduction to Machine Learning*. Pitman, 1988.

[Kodratoff 90] Y. Kodratoff. Combining similarity and causality in creative analogy. In L.C. Aiello, editor, *Proceedings of the 9th European Conference on Artificial Intelligence*, pages 398–403. Pitman, 1990.

[Kodratoff and Michalski 90] Yves Kodratoff and R.S Michalski, editors. *Machine Learning : an artificial intelligence approach, Volume 3*. Morgan Kaufmann, 1990.

[Kodratoff and Tecuci 87] Y. Kodratoff and G. Tecuci. Disciple-1 : interactive system in weak theory fields. In *Proceedings of the 10th International Joint Conference on Artificial Intelligence*. Morgan Kaufmann, 1987.

[Kowalski 79] R. Kowalski. *Logic for problem solving*. North-Holland, 1979.

[Krawchuk and Witten 88] B. Krawchuk and I. Witten. On asking the right questions. In *Proceedings of the 5th Machine Learning Conference*, pages 15–22. Morgan Kaufmann, 1988.

[Laird 86] P.D Laird. Inductive inference by refinement. In *Proceedings of the 5th American Conference on Artificial Intelligence*, pages 472–476. Morgan Kaufmann, 1986.

[Lambrix 90] P. Lambrix. Een planner voor een autonome agent. Master's thesis, Department of Computer Science, Katholieke Universiteit Leuven, 1990. in Dutch.

[Langley 85] P. Langley. Strategy acquisition governed by experimentation. In L. Steels and J. Campbell, editors, *Progress in Artificial Intelligence*. Ellis Horwood, 1985.

[Langley and Carbonell 87] P. Langley and J.G. Carbonell. Machine learning. In S. Shapiro, editor, *Encyclopedia of artificial intelligence*. Wiley, 1987.

[Lenat 83] D. Lenat. The role of heuristics in learning by discovery : three case studies. In R.S Michalski, J.G. Carbonell, and T.M. Mitchell, editors, *Machine Learning : an artificial intelligence approach*. Tioga publishing company, 1983.

[Lenat and Brown 83] D. Lenat and J.S. Brown. Why am and eurisco appear to work. *Artificial Intelligence*, 23:269–294, 1983.

[Lloyd 87] J.W. Lloyd. *Foundations of logic programming*. Springer-Verlag, 2nd edition, 1987.

[Loveland 78] D.W. Loveland. *Automated Theorem Proving : a logical basis.* North-Holland, 1978.

[Mehra, Rendell and Wah 89] P. Mehra, L.A. Rendell, and B.W. Wah. Principled constructive induction. In *Proceedings of the 11th International Joint Conference on Artificial Intelligence*, pages 651–657. Morgan Kaufmann, 1989.

[Michalski 75] R.S. Michalski. Variable-valued logic and its applications to pattern recognition and machine learning. In D. Rine, editor, *Multiple-Valued Logic and Computer Science.* North-Holland, 1975.

[Michalski 83] R.S. Michalski. A theory and methodology of inductive learning. In R.S Michalski, J.G. Carbonell, and T.M. Mitchell, editors, *Machine Learning : an artificial intelligence approach*, volume 1. Morgan Kaufmann, 1983.

[Michalski 87] R.S. Michalski. Clustering. In S. Shapiro, editor, *Encyclopedia of artificial intelligence.* Wiley, 1987.

[Michalski and Chilausky 80] R.S. Michalski and R.L. Chilausky. Learning by being told and learning from examples : an experimental comparaison of the two methods of knowledge acquisition in the context of developing an expert system for soybean disease diagnosis. *Policy analysis and information systems*, 4, 1980.

[Michalski and Stepp 83] R.S. Michalski and R.E. Stepp. Learning from observation : conceptual clustering. In R.S Michalski, J.G. Carbonell, and T.M. Mitchell, editors, *Machine Learning : an artificial intelligence approach*, volume 1. Tioga Publishing Company, 1983.

[Michalski, Carbonell and Mitchell 83] R.S Michalski, J.G. Carbonell, and T.M. Mitchell. *Machine Learning : an artificial intelligence approach, Volume 1.* Morgan Kaufmann, 1983.

[Michalski, Carbonell and Mitchell 86] R.S Michalski, J.G. Carbonell, and T.M. Mitchell. *Machine Learning : an artificial intelligence approach, Volume 2.* Morgan Kaufmann, 1986.

[Michalski, Davis, Bisht and Sinclair 85] R.S. Michalski, J.H. Davis, V.S. Bisht, and J.B. Sinclair. Plant/ds : an expert consulting system for the diagnosis of soybean diseases. In L. Steels and J.A. Campbell, editors, *Progress in Artificial Intelligence.* Ellis Horwood, 1985.

[Mitchell 77] T.M. Mitchell. Version spaces : a candidate elimination approach to rule learning. In *Proceedings of the 5th International Joint Conference on Artificial Intelligence*, pages 305–310. Morgan Kaufmann, 1977.

[Mitchell 80] T.M. Mitchell. The need for biases in learning generalizations. Technical Report CBM-TR-117, Department of Computer Science, Rutgers University, 1980.

[Mitchell 82] T.M. Mitchell. Generalization as search. *Artificial Intelligence*, 18:203–226, 1982.

[Mitchell, Keller and Kedar-Cabelli 86] T.M. Mitchell, R.M. Keller, and S.T. Kedar-Cabelli. Explanation based generalization : a unifying view. *Machine Learning*, 1:47–80, 1986.

[Mitchell, Mahadevan and Steinberg 85] T.M. Mitchell, S. Mahadevan, and L.I. Steinberg. Leap : a learning apprentice for vlsi design. In *Proceedings of the ninth International Joint Conference on Artificial Intelligence*, pages 573–580. Morgan Kaufmann, 1985.

[Mitchell, Utgoff and Banerji 83] T.M. Mitchell, P.E. Utgoff, and R. Banerji. Learning by experimentation : acquiring and refining problem-solving heuristics. In R.S Michalski, J.G. Carbonell, and T.M. Mitchell, editors, *Machine Learning : an artificial intelligence approach*, volume 1. Tioga publishing company, 1983.

[Morik 89] Katharina Morik. Sloppy modeling. In Katharina Morik, editor, *Knowledge Representation and Organization in Machine Learning*, volume 347 of *Lecture Notes in Artificial Intelligence*. Springer-Verlag, 1989.

[Muggleton 87] S. Muggleton. Duce, an oracle based approach to constructive induction. In *Proceedings of the 10th International Joint Conference on Artificial Intelligence*, pages 287–292. Morgan Kaufmann, 1987.

[Muggleton and Buntine 88] S. Muggleton and W. Buntine. Machine invention of first order predicates by inverting resolution. In *Proceedings of the 5th International Conference on Machine Learning*. Morgan Kaufmann, 1988.

[Murray 87] K.S. Murray. Multiple convergence : an approach to disjunctive concept acquisition. In *Proceedings of the 10th International Joint Conference on Artificial Intelligence*. Morgan Kaufmann, 1987.

[Naish 86] L. Naish. *Negation and Control in Prolog*, volume 238 of *Lecture Notes in Computer Science*. Springer-Verlag, 1986.

[Nilsson 80] N.J. Nilsson. *Principles of Artificial Intelligence*. Tioga Publishing Company, 1980.

[Plotkin 70] G. Plotkin. A note on inductive generalization. In *Machine Intelligence*, volume 5. Edinburgh University Press, 1970.

[Puget 89] J.F. Puget. Learning concept negation. In *Proceedings of the 4th European Working Session on Learning*, pages 179–188. Pitman, 1989.

[Quinlan 79] J.R. Quinlan. Discovering rules from large collections of examples : a case study. In D. Michie, editor, *Expert Systems in the Micro Electronic Age*. Edinburgh University Press, 1979.

[Quinlan 86] J.R. Quinlan. Induction of decision trees. *Machine Learning*, 1:81–106, 1986.

[Quinlan 87] J.R. Quinlan. Generating production rules from decision trees. In *Proceedings of the Tenth International Joint Conference on Artificial Intelligence*, pages 304–307. Morgan Kaufmann, 1987.

[Quinlan 90] J.R. Quinlan. Learning logical definition from relations. *Machine Learning*, 5:239–266, 1990.

[Rajamoney and De Jong 87] S. Rajamoney and G. De Jong. The classification, detection and handling of imperfect theory problems. In *Proceedings of the 10th International Joint Conference on Artificial Intelligence*. Morgan Kaufmann, 1987.

[Reiter 87] R. Reiter. Non-monotonic reasoning. In *Annual Reviews of Computer Science*, pages 147–180. 1987.

[Reiter 90] R. Reiter. On asking what a database knows. In J.W. Lloyd, editor, *Computational Logic*. Springer-Verlag, 1990.

[Rendell 86] L. Rendell. A general framework for induction and a study of selective induction. *Machine Learning*, 1:177–226, 1986.

[Rendell 89] L. Rendell. Comparing systems and analyzing functions to improve constructive induction. In *Proceedings of the 6th International Workshop on Machine Learning*, pages 461–464. Morgan Kaufmann, 1989.

[Rendell, Seshu and Tcheng 87] L. Rendell, R. Seshu, and D. Tcheng. More robust concept-learning using dynamically variable bias. In *Proceedings of the 4th International Workshop on Machine Learning*, pages 66–78. Morgan Kaufmann, 1987.

[Ritchie and Hanna 83] G.D. Ritchie and F.K. Hanna. Am : A case study in ai methodology. *Artificial Intelligence*, 23:249–268, 1983.

[Rouveirol and Puget 89] Celine Rouveirol and Jean-Francois Puget. A simple solution for inverting resolution. In Katharina Morik, editor, *Proceedings of the 4th European Working Session on Learning*. Pitman, 1989.

[Ruff and Dietterich 89] R.A. Ruff and T.G. Dietterich. What goods are experiments ? In *Proceedings of the 6th International Workshop on Machine Learning*. Morgan Kaufmann, 1989.

[Russell and Grosof 90] S. Russell and B. Grosof. A sketch of autonomous learning using declarative bias. In P.B. Brazdil and K. Konolige, editors, *Machine Learning, Meta-Reasoning and Logics*. Kluwer Academic Publishers, 1990.

[Russell and Subramanian 90] S. Russell and D. Subramanian. Mutual constraints on representation and inference. In P.B. Brazdil and K. Konolige, editors, *Machine Learning, Meta-Reasoning and Logics*. Kluwer Academic Publishers, 1990.

[Sammut and Banerji 86] C. Sammut and R. Banerji. Learning concepts by asking questions. In R.S. Michalski, J.G. Carbonell, and T.M. Mitchell, editors, *Machine Learning : an artificial intelligence approach*, volume 2. Morgan Kaufmann, 1986.

[Samuel 67] A. Samuel. Some studies in machine learning using the game of checkers ii - recent progress. *IBM Journal of Research and Development*, 11:601–617, 1967.

[Shapiro 81] E.Y. Shapiro. An algorithm that infers theories from facts. In *Proceedings of the Seventh International Joint Conference on Artificial Intelligence*, pages 446–452. Morgan Kaufmann, 1981.

[Shapiro 83] Ehud Y. Shapiro. *Algorithmic Program Debugging*. The MIT press, 1983.

[Smith, Mitchell, Winston and Buchanan 85] R.G. Smith, T.M. Mitchell, H.A. Winston, and B.G. Buchanan. Representation and use of explicit justifications for knowledge base refinement. In *Proceedings of the 9th International Joint Conference on Artificial Intelligence*. Morgan Kaufmann, 1985.

[Stepp, Whitehall and Lawrence 88] R.E. Stepp, B.L. Whitehall, and B.H. Lawrence. Towards intelligent machine learning algorithms. In *Proceedings of the 8th European Conference on Artificial Intelligence*, pages 333–338. Pitman, 1988.

[Sterling and Shapiro 86] Leon Sterling and Ehud Shapiro. *The art of Prolog*. The MIT press, 1986.

[Subramanian 90] D. Subramanian. A theory of justified reformulations. In P.D. Benjamin, editor, *Change of representation and inductive bias*. Kluwer Academic Publishers, 1990.

[Subramanian and Feigenbaum 86] D. Subramanian and J. Feigenbaum. Factorization in experiment generation. In *Proceedings of the fifth National Conference On Artificial Intelligence*, pages 518–522. Morgan Kaufmann, 1986.

[Subramanian and Genesereth 87] D. Subramanian and M. Genesereth. The relevance of irrelevance. In *Proceedings of the 10th International Joint Conference on Artificial Intelligence*, pages 416–422. Morgan Kaufmann, 1987.

[Tecuci 88] G. Tecuci. *DISCIPLE : A theory, methodology and system of expert knowledge acquisition*. PhD thesis, Universite Paris-Sud, Orsay, 1988.

[Tecuci and Kodratoff 90] G. Tecuci and Y. Kodratoff. Apprenticeship learning in nonhomogeneous domain theories. In Y. Kodratoff and R.S. Michalski, editors, *Machine Learning : an artificial intelligence approach*, volume 3. Morgan Kaufmann, 1990.

[Thieme 89] S. Thieme. The acquisition of model-knowledge for a model-driven machine learning approach. In K. Morik, editor, *Knowledge Representation and Organization in Machine Learning*, volume 347 of *Lecture Notes in Artificial Intelligence*. Springer Verlag, 1989.

[Tomasovic 88] A. Tomasovic. View update translation via deduction and annotation. In *Proceedings 2nd International Conference on Database Theory*, volume 326 of *Lecture Notes in Computer Science*, pages 338–351. Springer-Verlag, 1988.

[Turner 84] R. Turner. *Logics for artificial intelligence.* Horwood, Chichester, 1984.

[Ullmann 89] J.D. Ullmann. *Principles of data bases.* Computer Science Press, 1989.

[Utgoff 86] P.E. Utgoff. Shift of bias for inductive concept-learning. In R.S Michalski, J.G. Carbonell, and T.M. Mitchell, editors, *Machine Learning : an artificial intelligence approach.* Morgan Kaufmann, 1986.

[Utgoff and Mitchell 82] P.E. Utgoff and T.M. Mitchell. Acquisition of appropriate bias for concept learning. In *Proceedings of the 2nd National Conference On Artificial Intelligence*, pages 414–418. Morgan Kaufmann, 1982.

[Valiant 84] L. Valiant. A theory of the learnable. *Communications of the Association for Computing Machinery*, 27:1134–1142, 1984.

[Vere 75] S.A. Vere. Induction of concepts in the predicate calculus. In *Proceedings of the Fourth International Joint Conference on Artificial Intelligence*, pages 282–287. Morgan Kaufmann, 1975.

[Vrain and Lu 88] C. Vrain and C.R. Lu. An analogical method to do incremental learning of concepts. In *Proceedings of the 3rd European Working Session On Learning*, pages 227–235. Pitman, 1988.

[Wilkins 89] D.C. Wilkins. Knowledge base refinement using apprenticeship learning techniques. In K. Morik, editor, *Knowledge Representation and Organization in Machine Learning*, volume 347 of *Lecture Notes in Artificial Intelligence*. Springer-Verlag, 1989.

[Wirth 88] R. Wirth. Learning by failure to prove. In D. Sleeman, editor, *Proceedings of the 3rd European Working Session on Learning.* Pitman, 1988.

[Wrobel 88] S. Wrobel. Automatic representation adjustment in an observational discovery system. In Sleeman D., editor, *Proceedings of the 3rd European Working Session on Learning.* Pitman, 1988.

[Wrobel 89] S. Wrobel. Demand driven concept-formation. In K. Morik, editor, *Knowledge Representation and Organization in Machine Learning*, volume 347 of *Lecture Notes in Artificial Intelligence*. Springer-Verlag, 1989.

[Yokomori 86] T. Yokomori. Logic program forms. *New Generation Computing*, 4:305–320, 1986.

[Zinnsmeister and Hoppe 89] G. Zinnsmeister and H.V. Hoppe. Acquisition of task-knowledge for an adaptive user interface. In *Proceedings of the 3rd European Knowledge Acquisition for knowledge based systems Workshop*, 1989.

Inhoud

1 Situering en motivatie. 2
1.1 Bijdragen. 3

2 Het generisch algoritme Gencol. 3
2.1 Concepten leren uit voorbeelden. 4
2.2 Het leren van concepten als een zoekprobleem. 6
2.3 Het algoritme Gencol. 6
2.4 Generische parameters. 8
2.4.1 Talen en transformatoren. 8
2.4.2 Initialisatie. 8
2.4.3 Stop_Criterium. 8
2.4.4 Kwaliteits_Criterium. 8
2.4.5 De zoekstrategie. 8
2.4.6 Het gebruik van heuristieken. 9
2.4.7 Het geheugen. 9
2.4.8 Voorbeelden. 9
2.5 Evaluatie van Gencol. 9

3 Interactief leren van concepten. 10
3.1 De basismethode van Clint. 10
3.1.1 Het kader van Clint. 10
3.1.2 Een negatief voorbeeld wordt gedekt. 11
3.1.3 Een positief voorbeeld wordt niet gedekt. 11
3.2 Het gebruik van integriteitsbeperkingen. 14
3.3 Het vergaren van objectkennis. 15
3.4 Een driewaardige logica voor Clint. 18
3.5 Constructieve Inductie door Analogie. 19
3.6 Toepassingen. 21
3.7 Conclusie. 21

Samenvatting.

1 Situering en motivatie.

Automatisch leren is een belangrijk deelgebied van kunstmatige intelligentie. Het onderzoek rond automatisch leren heeft tot doel systemen te ontwikkelen die leren. Men zegt dat een systeem leert, wanneer het door zichzelf aan te passen beter een bepaalde taak kan vervullen. Hieruit volgt dat een systeem leert, wanneer het na verloop van tijd nauwkeuriger werkt, een ruimere klasse van problemen kan behandelen of efficiënter wordt.

De motivatie voor het onderzoek rond automatisch is tweeledig. Ten eerste is de mogelijkheid tot leren een noodzakelijke voorwaarde om te komen tot echte intelligentie. Daarom kan kunstmatige intelligentie nooit zijn doelstellingen bereiken zonder automatisch leren. Ten tweede kan automatisch leren bijdragen tot het automatiseren van het kennisverwervingsproces [Morik 89, Michalski, Davis, Bisht en Sinclair 85]. Het is algemeen bekend dat het kennisverwervingsproces het belangrijkste probleem is bij de bouw van kennisgebaseerde systemen.

De meest onderzochte vorm van automatisch leren is het leren van concepten uit voorbeelden. Systemen die concepten leren uit voorbeelden, leiden algemene regels af uit voorbeelden. Voorbeelden zijn (formele) voorstellingen van specifieke observaties. Het leren uit voorbeelden wordt reeds lang bestudeerd (zie bijvoorbeeld [Michalski, Carbonell en Mitchell 83, Cohen en Feigenbaum 81]) en er bestaan heel wat verschillende systemen en benaderingen. Typerend voor het merendeel van deze systemen is echter dat ze moeilijk te hanteren zijn, weinig kennis gebruiken, een beperkt kennisvoorstellingsformalisme veronderstellen en vertrekken van een gegeven verzameling voorbeelden. Deze problemen zorgen ervoor dat deze systemen niet bruikbaar zijn in de kontekst van *learning apprentice* systemen [Mitchell, Mahadevan en Steinberg 85, Smith, Mitchell, Winston en Buchanan 85]. Learning apprentice systemen zijn interactieve programma's die de gebruiker assisteren bij de ontwikkeling en het onderhoud van kennisbanken. Volgens [Smith, Mitchell, Winston en Buchanan 85] moet een learning apprentice systeem twee verschillende aspecten ondersteunen :

- het gedeeltelijk automatiseren van de ontwikkeling van de initiële kennisbank door het genereren van regels vanuit een benaderende domein-theorie.
- interactie met de gebruiker om de bestaande kennisbank aan te passen opdat het systeem zou bijleren uit ervaring. Deze interactie dient plaats te vinden tijdens de normale sessies met het kennisgebaseerde systeem.

Ons onderzoek rond het interactief leren van concepten werd vooral geïnspireerd vanuit de discrepantie tussen de bestaande systemen die leren uit voorbeelden, en de kenmerken waaraan een leersysteem moet voldoen om bruikbaar te zijn bij het kennisverwervingsproces en voor learning apprentice systemen. De belangrijkste doelstelling voor ons onderzoek bestaat er dan ook in om een inductief leersysteem te ontwikkelen dat toepasbaar is in *learning apprentice* systemen. Hiertoe dient het leersysteem opportunistisch en gebruiksvriendelijk te zijn. Een leersysteem is opportunistisch wanneer het de geboden

kansen tot leren zo veel mogelijk benut. Het is gebruiksvriendelijk wanneer het op eenvoudige wijze benut kan worden en de gebruiker de werking van het systeem niet dient te verstaan.

1.1 Bijdragen.

Alvorens een nieuwe techniek te ontwerpen, is het noodzakelijk om het bestaande werk te kennen. Daarom zullen we als eerste bijdrage een algemeen kader schetsen voor systemen die concepten leren uit voorbeelden. Centraal in dit kader staat het generisch algoritme Gencol. Het werk rond Gencol kan beschouwd worden als een inleiding tot het leren van concepten uit voorbeelden. We willen echter benadrukken dat het geen overzicht van de gekende systemen vormt, maar een uniek en algemeen kader dat abstractie maakt van algoritmen die concepten leren uit voorbeelden. Deze abstractie wordt gerealiseerd door het invoeren van een aantal generische parameters die elk een specifiek aspect van het leren uit voorbeelden omvatten. Zo zijn er parameters die abstractie maken van de zoekstrategie, de beschrijvingstalen, de heuristieken en de operatoren of transformatoren. Om de algemeenheid van het kader aan te tonen, werd bewezen dat het algoritme abstractie maakt van 5 gekende systemen die op het eerst zicht weinig met elkaar te maken hebben. Het kader rond Gencol is dan ook uniek omdat het gebruikt kan worden om bijna elk bestaand leersysteem dat concepten leert uit voorbeelden, te realiseren en omdat het aantoont dat verschillende technieken een speciaal geval zijn van een meer algemene techniek. Bovendien is het algoritme Gencol vrij gedetailleerd. Het kader rond Gencol zal verder behandeld worden in hoofdstuk 2.

Als tweede bijdrage werd een omgeving voor het leren van concepten ontwikkeld rond het systeem Clint. Tijdens de ontwikkeling van de verschillende technieken in Clint hebben we steeds de vereisten van de learning apprentice kontekst voor ogen gehouden. Clint wil dan ook een goede leerling zijn : het systeem leert vlug en is ook leergierig. Hiertoe hebben we geprobeerd om zo veel mogelijk gebruiksvriendelijke en opportunistische eigenschappen te integreren in Clint. Clint genereert zelf voorbeelden, kan de beschrijvingstaal van de concepten automatisch aanpassen, herstelt van fouten, benut kennis in de vorm van regels en beperkingen, gebruikt een logisch kader gebaseerd op logisch programmeren, vergaart kennis over ongekende objecten, maakt een onderscheid tussen wat het weet en wat het niet weet en wil zelf nieuwe concepten voorstellen. Clint gaat heel wat verder dan de bestaande aanpakken, zowel wat betreft gebruiksvriendelijkheid en opportunisme als wat betreft de geboden mogelijkheden. We zijn er dan ook van overtuigd dat Clint de basis kan vormen voor een praktische component van learning apprentice systemen. Clint wordt in hoofdstuk 3 behandeld.

2 Het generisch algoritme Gencol.

In dit hoofdstuk schetsen we het kader voor het leren van concepten uit voorbeelden rond het generisch algoritme Gencol. Hiertoe specifiëren we eerst de probleemstelling van het leren van concepten en geven we vervolgens een samenvatting van het generisch algoritme Gencol.

2.1 Concepten leren uit voorbeelden.

Het leren van concepten uit voorbeelden houdt in dat men veralgemeningen construeert van specifieke observaties. Om dit proces door programma's of systemen te laten ondersteunen is het noodzakelijk om een aantal aspecten te formaliseren. Aangezien programma's onmogelijk met de observaties en de concepten (verzamelingen van observaties) als dusdanig kunnen werken, is het nodig om beschrijvingen van concepten en voorbeelden in te voeren. Beschrijvingen van concepten worden uitgedrukt in een taal van concepten L_C en beschrijvingen van observaties – dit zijn voorbeelden – in een taal van voorbeelden L_e. Vanzelfsprekend moet het leersysteem ook het verband tussen beide talen kennen. Dit wordt mogelijk door het predicaat $dekt(C, v)$, dat slaagt indien v de beschrijving is van een observatie en C de beschrijving van een concept, zodat de observatie tot het concept behoort. Men onderscheidt ook twee soorten van voorbeelden : positieve voorbeelden en negatieve. Een positief voorbeeld is een voorbeeld dat aan het concept voldoet, terwijl een negatief voorbeeld dat niet doet. De bedoeling van het leren van concepten is dan om een concept-beschrijving in L_C te vinden die aan bepaalde voorwaarden voldoet. Deze voorwaarden specifiëren meestal dat zoveel mogelijk positieve en zo weinig mogelijk negatieve voorbeelden gedekt dienen te worden. We zullen deze voorwaarden het *kwaliteitscriterium* noemen. Samengevat kunnen we het probleem van het leren van concepten als volgt formuleren :

Probleemstelling 1 : het leren van concepten

- **Gegeven :**
  - een taal van voorbeelden L_e
  - een taal van concept-beschrijvingen L_C
  - het predicaat *dekt* dat het verband tussen L_e en L_C specifieert
  - een verzameling van positieve voorbeelden P in L_e
  - een verzameling van negatieve voorbeelden N in L_e
  - een kwaliteitscriterium K dat aangeeft of een bepaalde concept-beschrijving C een aanvaardbare oplossing is met betrekking tot de voorbeelden en de talen
- **Vind :** een concept-beschrijving C, uitgedrukt in L_C, die aan het kwaliteitscriterium K voldoet met betrekking tot de voorbeelden in P en N.

Deze probleemstelling wordt geïllustreerd in voorbeeld 1.

Voorbeeld 1 : concepten leren uit voorbeelden

Beschouw het volgende probleem :

- $L_e = \{$ `vb(`x,y`)` $\mid x \in T_x$ en $y \in T_y$ $\}$ hierbij is
  $T_x = \{$ `driehoek, rechthoek, vierkant, trapezium, cirkel, ellips` $\}$ en
  $T_y = \{$ `rood, blauw, wit, zwart, oranje, roze` $\}$

- L_C = $\{$ `vb(X,Y)` $\leftarrow$ `iseen(X,`h_x`)` $\wedge$ `iseen(Y,`h_y`)` $\mid h_x \in T_x \cup A_x$ en $h_y \in T_y \cup A_y\}$
  hierbij is $A_x = \{$ `vierhoekig, veelhoek, ovaal, vorm` $\}$ en
  $A_y = \{$ `mono, poly, kleur` $\}$

- *dekt*(C,v) slaagt indien het voorbeeld v logisch volgt uit de hieronder geschetste kennisbank en de concept-beschrijving C

- het kwaliteitscriterium eist dat alle positieve en geen enkel negatief voorbeeld gedekt moeten zijn door de concept-beschrijving

- $P = \{$ `vb(vierkant,rood), vb(rechthoek, blauw)` $\}$

- N = $\{$ `vb(cirkel,oranje), vb(driehoek, roze)` $\}$

De kennisbank van dit voorbeeld wordt gespecifieerd door de volgende feiten en regel :

```
zoon(driehoek,veelhoek) ←          zoon(rood,mono) ←
zoon(rechthoek,vierhoek) ←         zoon(blauw,mono) ←
zoon(vierkant,vierhoek) ←          zoon(wit,mono) ←
zoon(trapezium,vierhoek) ←         zoon(zwart,mono) ←
zoon(cirkel,ovaal) ←               zoon(oranje,poly) ←
zoon(ellips,ovaal) ←               zoon(roze,poly) ←
zoon(vierhoek,veelhoek) ←          zoon(mono,kleur) ←
zoon(ovaal, vorm) ←                zoon(poly,kleur) ←
zoon(veelhoek,vorm) ←

iseen(X,X) ←
iseen(X,Y) ← zoon(X,Z) ∧ iseen(Z,Y)
```

Een aantal concept-beschrijvingen die een oplossing vormen voor het geschetste probleem zijn :

```
vb(X,Y) ← iseen(X,vierhoek) ∧ iseen(Y,kleur)
vb(X,Y) ← iseen(X,vorm) ∧ iseen(Y,mono)
```

2.2 Het leren van concepten als een zoekprobleem.

Uit probleemstelling 1 kan men een heel eenvoudig algoritme afleiden : genereer alle concept-beschrijvingen en test of ze aan het kwaliteitscriterium voldoen. Uiteraard is dit algoritme erg naïef en houdt het geen rekening met het aantal mogelijke concept-beschrijvingen, dat erg groot kan zijn. Daarom is het veel interessanter om deze ruimte van concept-beschrijvingen te structureren door middel van het predicaat *algemener-dan*(C_1, C_2), dat slaagt indien C_1 en C_2 concept-beschrijvingen zijn zodat alle voorbeelden die gedekt worden door C_2 ook gedekt worden door C_1. C_1 wordt een veralgemening van C_2 genoemd. Dit predicaat laat toe om de zoekruimte te snoeien. Immers[1],

- Wanneer een concept-beschrijving een voorbeeld dekt, zullen ook alle veralgemeningen van die beschrijving het voorbeeld dekken. Indien het gedekte voorbeeld negatief is, betekent dit dat men alle veralgemeningen van de concept-beschrijving mag elimineren.

- Wanneer een concept-beschrijving een voorbeeld niet dekt, zal geen enkele beschrijving die minder algemeen is, het voorbeeld dekken. Indien het betreffende voorbeeld positief is, kunnen de minder algemene concept-beschrijvingen worden weggesnoeid.

De structuur die door de relatie *algemener-dan* wordt opgelegd aan de zoekruimte L_C, maakt het mogelijk om L_C op een efficiënte wijze te doorzoeken. Dit kan gerealiseerd worden door middel van transformatoren. Een veralgemeningstransformator vt (respectievelijk verfijningstransformator) is een functie van L_C naar de verzameling van alle deelverzamelingen van L_C zodat $\forall C \forall C' \in vt(C)$: *algemener-dan*(C', C) (respectievelijk *algemener-dan*(C, C')). Naargelang de structuur van de beschouwde taal en de aard van de functie met betrekking tot de relatie *algemener-dan* kan men verschillende soorten transformatoren onderscheiden. Zo zijn de transformatoren die beschrijvingen afbeelden op minimale veralgemeningen (of verfijningen) operatoren.

Laat ons ter illustratie voorbeeld 1 opnieuw beschouwen. Daar hebben we dat `vb(X,Y)` ← `isa(X,veelhoek)` ∧ `isa(Y,kleur)` algemener is dan `vb(X,Y)` ← `isa(X,driehoek)` ∧ `isa(Y,rood)`. Bovendien kan men transformatoren definiëren die een beschrijving veralgemenen of verfijnen door het aanpassen van de constanten in de concept-beschrijvingen. Wanneer de constanten vervangen worden door constanten die hoger in de hiërarchie zitten, heeft men een veralgemeningstransformator. In het omgekeerde geval krijgt men een verfijningstransformator.

2.3 Het algoritme Gencol.

In algoritme 1 wordt het algoritme Gencol samengevat. De generische parameters zijn *schuin gedrukt.* Bovendien wordt verondersteld dat alle parameters en procedures binnen Gencol toegang hebben tot de variabelen P, N, L_e, L_C en het Geheugen.

De parameter Kandidaat_Oplossingen wordt gebruikt als een lijst die de concept-beschrijvingen bevat die nog verder onderzocht dienen te worden. Oplossingen bevat de uitvoer van Gencol. Het Geheugen wordt soms gebruikt om concept-beschrijvingen bij te houden die niet tot een oplossing kunnen leiden. De variabele kan bevat de kandidaat-oplossing die ofwel getransformeerd, ofwel toegevoegd zal worden aan de verzameling

[1] Deze eigenschappen hangen ook af van het kwaliteitscriterium.

Algoritme 1: Gencol

```
procedure Gencol
    Kandidaat_Oplossingen := Initialiseer
    Oplossingen := ∅
    Geheugen := ∅
    while not Stop_Criterium( Kandidaat_Oplossingen, Oplossingen) do
        kan := Selecteer ( Kandidaat_Oplossingen )
        Kandidaat_Oplossingen := Kandidaat_Oplossingen − { kan }
        if Kwaliteits_Criterium(kan) voldaan
        then Oplossingen := Oplossingen ∪ { kan }
        else if Optie
            then vb := Genereer_Voorbeeld( Kandidaat_Oplossingen, Oplossingen)
            endif
            Oneof [
              spec : Kandidaat_Oplossingen :=
                 Kandidaat_Oplossingen ∪ VerfijningsTransformator(kan, vb),
              gen : Kandidaat_Oplossingen :=
                 Kandidaat_Oplossingen ∪ VeralgemeningsTransformator(kan, vb)
                ]
            Geheugen := Geheugen ∪ Onthou(kan)
        endif
        Geheugen := Wijzig(Geheugen)
        Kandidaat_Oplossingen := Filter( Kandidaat_Oplossingen )
    endwhile
endproc
```

Oplossingen. In de variabele vb mag Gencol een voorbeeld bijhouden dat gebruikt kan worden door de transformators. *Optie* duidt aan dat de instructie vb := *Genereer_Voorbeeld* alleen onder bepaalde voorwaarden uitgevoerd zal worden. *Oneof*[**spec, gen**] betekent dat één van beide takken uitgevoerd zal worden.

2.4 Generische parameters.

We bespreken nu kort de verschillende generische parameters van Gencol. Dit vormt tevens een indicatie van de mogelijkheden om Gencol te instantiëren.

2.4.1 Talen en transformatoren.

De keuze van de talen en de transformatoren is bijzonder belangrijk. De talen bepalen namelijk de kennisrepresentatie van het probleem en de zoekruimte. De transformatoren leggen de acties vast die het systeem kan uitvoeren om de ruimte te doorzoeken.

2.4.2 Initialisatie.

De functie *Initialiseer* specifieert het vertrekpunt van het zoekproces. De functie kan één of meerdere concept-beschrijvingen als resultaat hebben. Bovendien kan Gencol vertrekken van de meest algemene concept-beschrijving, van de meest specifieke, of eventueel van tussenliggende beschrijvingen door een gepaste keuze van *Initialiseer* te maken.

2.4.3 Stop_Criterium.

Het *Stop_Criterium* legt de condities vast waaronder het leerproces beëindigd wordt. Aldus is het mogelijk om één, meerdere of alle concept-beschrijvingen die aan het kwaliteitscriterium voldoen, te vinden.

2.4.4 Kwaliteits_Criterium.

Het *Kwaliteits_Criterium* karakteriseert de oplossingen van het leerprobleem. Wanneer een kandidaat oplossing voldoet aan dit criterium wordt de kandidaat een volwaardige oplossing. Verschillende kwaliteitscriteria kunnen benut worden in Gencol. Bijvoorbeeld: volledigheid en consistentie [Mitchell 82], maximaal algemene of minimaal algemene concept-beschrijvingen of waarschijnlijk benaderende concept-beschrijvingen [Valiant 84].

2.4.5 De zoekstrategie.

Naargelang de keuze van de functie *Selecteer* worden verschillende zoekstrategieën in Gencol gerealiseerd. Wanneer *Selecteer* van het type eerst-in-eerst-uit is krijgen we een breedte-eerst zoekstrategie, voor laatst-in-eerst-uit wordt dit een diepte-eerst strategie, en voor het selecteren van een beste beschrijving krijgen we een best-eerst strategie.

De zoekstrategie wordt ook beïnvloed door de condities die vastleggen wanneer een transformator wordt toegepast. Dit aspect is weerspiegeld in de definitie van *Oneof*. Een aantal mogelijkheden hiervoor zijn : enkel veralgemenen, enkel verfijnen of de voorkeur geven aan verfijnen boven veralgemenen. In voorbeeld-gedreven algoritmen hangt *Oneof* af van het gedrag van de huidige kandidaat-beschrijving op een voorbeeld. Indien de

kandidaat een negatief voorbeeld dekt, wordt verfijnd. Indien een positief voorbeeld niet gedekt wordt, zal veralgemeend worden.

2.4.6 Het gebruik van heuristieken.

Aangezien de zoekruimte bijzonder groot kan zijn, is het gebruik van heuristieken aangewezen. Heuristieken kunnen op verschillende plaatsen door Gencol gebruikt worden :

- *Selecteer* : kan de interessantste beschrijvingen eerst selecteren
- *Filter* : kan gebruikt worden om de minder interessante beschrijvingen weg te snoeien. Wanneer steeds een vast aantal beschrijvingen overgehouden wordt, krijgt men een straal-zoekstrategie. *Filter* is ook nuttig om beschrijvingen die strijdig zijn met een domein-theorie weg te snoeien.
- De transformatoren kunnen ook heuristieken gebruiken. Meestal heeft dit te maken met de keuze van het disjunctief gedeelte van een beschrijving dat getransformeerd zal worden.

2.4.7 Het geheugen.

Sommige systemen, zoals het Model Inference System van [Shapiro 83], houden foutieve concept-beschrijvingen bij om te vermijden dat die meer dan eens zouden beschouwd worden en ook om eventuele oneindige lussen te vermijden. In Gencol kan dit gerealiseerd worden met behulp van de functies *Onthou* en *Wijzig*.

2.4.8 Voorbeelden.

P en N zijn de verzamelingen van positieve en negatieve voorbeelden. Zij kunnen initieel gegeven zijn of opgebouwd worden tijdens het leerproces. In het tweede geval mag het systeem of de gebruiker op welbepaalde tijdstippen nieuwe voorbeelden genereren. Hiertoe kan de functie *Genereer_Voorbeeld* gebruikt worden. De functie *Optie* geeft aan wanneer een nieuw voorbeeld behandeld zal worden.

2.5 Evaluatie van Gencol.

Om de algemeenheid van het kader en het algoritme Gencol aan te tonen, werd bewezen dat Gencol abstractie maakt van 5 gekende leeralgoritmen, die op het eerste gezicht maar weinig met elkaar te maken hebben. Het betreft : de versionspace methode van [Mitchell 82], de beslissingsboom-familie van [Quinlan 86], het Model Inference System van [Shapiro 83], de AQ-methodologie van [Michalski 83] en het systeem Clint van [De Raedt en Bruynooghe 89c]. Bovendien is het algoritme Gencol vrij gedetailleerd, wat toelaat om van Gencol te vertrekken bij de implementatie van een leersysteem. Daarom ben ik ervan overtuigd dat het kader rond Gencol een bijzonder interessante bijdrage vormt voor de didactiek, de theorie en de praktijk van het automatisch leren van concepten uit voorbeelden.

3 Interactief leren van concepten.

In dit hoofdstuk wordt een overzicht gegeven van de verschillende technieken die benut worden in het systeem Clint. Dit hoofdstuk is gebaseerd op [De Raedt en Bruynooghe 90b]. Clint is een leersysteem dat op interactieve en gebruiksvriendelijke wijze concepten leert. Eerst wordt de basismethode van Clint besproken en nadien wordt dieper ingegaan op enkele uitbreidingen van het systeem : het behandelen van integriteitsbeperkingen, het leren en vergaren van kennis over objecten, het gebruik van een driewaardige logica en negatie, en het automatisch voorstellen van nieuwe concepten door middel van constructieve inductie en analogie.

3.1 De basismethode van Clint.

3.1.1 Het kader van Clint.

Clint gebruikt een logisch kader [Genesereth en Nilsson 87] om concepten en kennis voor te stellen. In dit kader is een concept een predicaat, een concept-beschrijving is een verzameling Horn-clausules voor het concept en voorbeelden zijn volledig geïnstantieerde feiten. Positieve voorbeelden zijn ware feiten en negatieve voorbeelden zijn onwaar. Een concept-beschrijving dekt een voorbeeld indien het voorbeeld logisch volgt uit de concept-beschrijving en de kennisbank. Het basisprobleem dat Clint oplost, kan als volgt gespecifieerd worden :

Probleemstelling 2 : het basisprobleem van Clint

- **Gegeven :**
  - een voorbeeld dat ingevoerd wordt door de gebruiker
  - een kennisbank die predicaat-definities bevat en ook voor alle geleerde predicaten een verzameling positieve en negatieve voorbeelden
  - een reeks van talen $L_1, L_2, L_3, ..., L_M$
  - een orakel (de gebruiker) dat voorbeelden wil classificeren als positief of negatief
- **Vind :** een concept-beschrijving zodat alle positieve en geen negatieve voorbeelden gedekt worden.

Elke taal L_i legt syntactische beperkingen op aan de te leren clausules. Omwille van deze beperkingen is het zo dat er voor iedere taal concepten bestaan die niet beschrijfbaar zijn in de taal. Daarom wordt ook een reeks van talen gebruikt : indien een concept niet beschrijfbaar is in taal L_i, zal geprobeerd worden om het concept te leren in taal L_{i+1}.

In probleemstelling 2 doen zich twee verschillende opportuniteiten voor : een positief voorbeeld dat niet gedekt wordt en een negatief voorbeeld dat wel gedekt wordt.

3.1.2 Een negatief voorbeeld wordt gedekt.

In dit geval gaat Clint als volgt te werk : eerst wordt een bewijsboom (een EN-boom) geconstrueerd die aantoont dat het negatief voorbeeld waar is ; vervolgens wordt de EN-boom geanalyseerd met de bedoeling een clausule te vinden die gebruikt werd in het bewijs, maar foutief is. Hiertoe worden vragen geformuleerd aan de gebruiker omtrent de waarheidswaarde van de knopen in de bewijsboom. Ieder antwoord komt overeen met de classificatie van een voorbeeld. Het vinden van de foutieve clausule is analoog aan de debuggingsprocedure van [Shapiro 83]. Eenmaal de foutieve clausule geïdentificeerd is, wordt zij verwijderd uit de kennisbank en worden de positieve voorbeelden die erdoor gedekt waren, opnieuw behandeld.

3.1.3 Een positief voorbeeld wordt niet gedekt.

In dit geval gaat Clint in twee stappen te werk : in een eerste stap wordt een start-clausule gegenereerd die het positieve voorbeeld dekt en geen enkel negatief voorbeeld ; in de tweede stap wordt deze start-clausule veralgemeend via het genereren van bijkomende voorbeelden die door de gebruiker moeten geclassificeerd worden.

Om de start-clausule te berekenen gebruikt Clint een functie $\mathcal{F}$, die een positief voorbeeld vb, een kennisbank KB en een concept-beschrijvingstaal L afbeeldt op een verzameling clausules. Het resultaat van de functie $\mathcal{F}$ is de verzameling van maximaal specifieke clausules die het voorbeeld vb dekken en uit te drukken zijn in de beschrijvingstaal L, $\mathcal{F}(vb, KB, L)$. Aangezien het kan voorkomen dat een maximaal specifieke clausule negatieve voorbeelden dekt, dient Clint verscheidene maximaal specifieke clausules te beschouwen [De Raedt en Bruynooghe 89b, De Raedt en Bruynooghe 90a]. Hiertoe gebruikt Clint een reeks van talen. De start-clausule c is dan een clausule $c \in \mathcal{F}(vb, KB, L_i)$, zodat alle clausules in $\mathcal{F}(vb, KB, L_j)$ voor $j < i$ negatieve voorbeelden dekken.

Indien er geen consistente maximaal specifieke clausule bestaat met betrekking tot de huidige kennisbank en alle talen van Clint, zal de behandeling van het betreffende positieve voorbeeld uitgesteld worden. Het voorbeeld zal dan opnieuw behandeld worden na wijziging van de kennisbank. Meestal zal er echter wel een start-clausule gevonden worden. Deze start-clausule wordt dan doorgespeeld naar de tweede stap van het algoritme, de veralgemeningsstap.

De functie $\mathcal{F}$ laat toe om het probleem van indirecte relevantie te behandelen, d.w.z. de functie bepaalt een eindige verzameling van relevante kenmerken die een verklaring vormen voor het voorbeeld en het concept. Deze manier van werken laat ook toe om de taal aan te passen wanneer dit nodig is.

In de tweede stap van het algoritme vertrekt Clint van een oude clausule c (initieel is dit de start-clausule) en wordt een veralgemening c' van c geconstrueerd zodat :

- c' een voorbeeld v dekt dat niet gedekt wordt door c
- c' geen negatieve voorbeelden dekt
- c' een minimale veralgemening is zodat de vorige twee voorwaarden geldig zijn.

Wanneer c' en v aan deze voorwaarden voldoen, wordt nagegaan of v reeds geclassificeerd werd door de gebruiker. Indien dit nog niet het geval is, wordt de gebruiker gevraagd om v te classificeren. Indien v positief is, wordt de oude clausule c vervangen door de nieuwe

c' en wordt het proces herhaald voor de nieuwe clausule. Indien v negatief is, wordt het voorbeeld bewaard en worden andere veralgemeningen van c onderzocht. Wanneer het onmogelijk is om andere veralgemeningen te construeren die aan de voorwaarden voldoen, wordt de oude clausule toegevoegd aan de kennisbank. Een voorbeeld dat de verschillende stappen toelicht, kan gevonden worden in voorbeeld 2.

Voorbeeld 2 : een sessie met Clint

Het relevante deel van de kennisbank voor ons voorbeeld ziet er als volgt uit :

```
vrouw(alice) ←                ouder(alice,roos) ←
vrouw(kaatje) ←               ouder(leo,roos) ←
vrouw(lieve) ←                ouder(roos,luc) ←
vrouw(roos) ←                 ouder(alice,stefaan) ←

man(luc) ←                    ouder(stefaan,jan) ←
man(leo) ←                    ouder(leo,stefaan) ←
man(stefaan) ←                ouder(etienne,luc) ←
man(etienne) ←                ouder(stefaan,kaatje) ←
```

In de volgende sessie met het algoritme wordt de invoer van de gebruiker **vetjes** gedrukt.

```
?-leer.
Volgend voorbeeld ?  @ grootouder(alice,luc).
Classificatie ?  @ positief.
Start-Clausule in L1 :

      grootouder(O,K) ← vrouw(O) ∧  man(K)

Geen verdere veralgemeningen.
Clausule toegevoegd aan de kennisbank.
```

Clint kan deze clausule niet verder veralgemenen omdat ze anders niet meer in de taal zou zitten. Taal L_1 eist dat alle variabelen in de hoofding van een clausule ook in het lichaam van de clausule voorkomen en omgekeerd.

```
?-leer.
Volgend voorbeeld ?  @ grootouder(lieve,luc).
Classificatie ?  @ negatief.
Is vrouw(lieve) waar ?  @ ja.
Is man(luc) waar ?  @ ja.
Verwijdering clausule :

      grootouder(O,K) ← vrouw(O) ∧  man(K)

grootouder(alice,luc) is niet langer gedekt
Start-Clausule in L1 :
```

```
grootouder(O,K) ← vrouw(O) ∧ man(K)

Start-Clausule dekt negatief voorbeeld
Volgende taal L1, Start-Clausule in L1 :

grootouder(O,K) ← vrouw(O) ∧ man(K) ∧ ouder(O,P) ∧ ouder(R,K)
```

Taal L_1 laat 1 existentieel gekwantificeerde variabele per conditie toe. Alle existentieel gekwantificeerde variabelen hebben echter slechts één voorkomen in de clausule.

```
Start-Clausule dekt negatief voorbeeld
Volgende taal L2, Start-Clausule in L2 :

grootouder(O,K) ← vrouw(O) ∧ man(K) ∧ ouder(O,P) ∧ ouder(R,K)
      ∧ vrouw(R) ∧ eq(P,R)
```

In deze taal worden ook relaties tussen de verschillende existentieel gekwantificeerde variabelen van L_1 toegelaten.

```
Is grootouder(leo,luc) waar ?  @ ja.
Volgende clausule :

grootouder(O,K) ← man(K) ∧ ouder(O,P) ∧ ouder(R,K)
      ∧ vrouw(R) ∧ eq(P,R)

Is grootouder(alice,jan) waar ?  @ ja.
Volgende clausule :

grootouder(O,K) ← man(K) ∧ ouder(O,P) ∧ ouder(R,K) ∧ eq(P,R)

Is grootouder(alice, kaatje) waar ?  @ ja.
Volgende clausule :

grootouder(O,K) ← ouder(O,P) ∧ ouder(R,K) ∧ eq(P,R)

Geen verdere veralgemeningen
Clausule toegevoegd aan kennisbank.
```

Er werd aangetoond dat Clint in de limiet een verzameling concepten identificeert op voorwaarde dat de concepten definieerbaar zijn in termen van de kennisbank en de talen in de reeks. Ook de mogelijkheden om het algoritme te gebruiken voor programma-synthese uit voorbeelden werden onderzocht.

3.2 Het gebruik van integriteitsbeperkingen.

In deductieve gegevensbanken is het gebruik van integriteitsbeperkingen een erg natuurlijke wijze om (beperkte) kennis over het domein voor te stellen. Het gebruik van integriteitsbeperkingen is ook erg nuttig voor het vergaren van kennis en voor het formuleren van bijkomende kennis in leersystemen. In praktische leersituaties is het vaak eenvoudig om een aantal integriteitsbeperkingen te formuleren ook al is het niet mogelijk om een sluitende definitie van het concept te geven. Deze beperkingen kunnen ook beschouwd worden als meta-kennis over het probleem. Het gebruik ervan versnelt het leerproces, aangezien het toelaat bepaalde hypothesen uit te sluiten en andere voorop te stellen. We hebben Clint aangepast opdat het rekening zou houden met integriteitsbeperkingen onder de vorm van algemene clausules. Een voorbeeld van zo een integriteitsbeperking is :

```
ouder(O,K) → vader(O,K) ∨  moeder(O,K)
  vader(O,K) ∧  moeder(O,K) → false
```

De eerste beperking zegt dat elke ouder ook een vader of een moeder is. De tweede beperking zegt dat de vader en de moeder niet dezelfde kunnen zijn. Een integriteitsbeperking is voldaan indien ze een stelling vormt in de theorie. Probleemstelling 2 dient nu ook gewijzigd te worden opdat Clint rekening zou houden met integriteitsbeperkingen. De bijkomende voorwaarde die hiertoe opgelegd wordt, is dat alle integriteitsbeperkingen stellingen moeten zijn in de geleerde kennisbank.

Om aan deze bijkomende voorwaarde te voldoen, werd Clint uitgebreid met twee techieken: een naïeve methode en een gretige. Hier zullen we enkel het principe van de naïeve methode uiteenzetten.

Wanneer een beperking $p_1 \wedge ... \wedge p_k \rightarrow q_1 \vee ... \vee q_m$ inconsistent is voor een substitutie θ en een kennisbank, dan is :

- één van de definities voor de predicaten in het lichaam van de beperking te algemeen. Dit wil zeggen dat $p_i\theta$ geïmpliceerd wordt door de kennisbank, maar onwaar is volgens de gebruiker.
- of één van de definities voor de predicaten in de hoofding van de beperking te specifiek. Dit wil zeggen dat $q_j\theta$ onwaar is volgens de kennisbank, maar wel waar zou moeten zijn volgens de gebruiker.

Deze analyse toont duidelijk aan dat inconsistente beperkingen toelaten om voorbeelden af te leiden. Indien er slechts één atomaire formule voorkomt in een beperking, dan is het duidelijk dat de oorzaak voor het probleem aan de betreffende formule te wijten is. Indien er echter meer dan één atomaire formule voorkomt in een tegengesproken beperking, dan moet de oorzaak nader bepaald worden. In Clint gebeurt dit door de gebruiker te vragen naar de waarheidswaarde van de atomaire formules in de geïnstantieerde beperking $(p_1 \wedge ... \wedge p_k \rightarrow q_1 \vee ... \vee q_m)\theta$. De atomaire formules die verantwoordelijk zijn voor de inconsistentie kunnen aldus geïdentificeerd worden en nadien als voorbeelden doorgegeven worden aan Clint. Dit wordt geïllustreerd in voorbeeld 3. Clint zal dan deze voorbeelden op de gewone manier verwerken. Wanneer Clint herstelt van de gekende fouten worden de beperkingen opnieuw geverifieerd. Dit kan uiteraard aanleiding geven tot nieuwe inconsistenties, bijkomende vragen aan de gebruiker en eventueel een nieuwe leerfase met Clint. In deze naïeve methode worden de beperkingen dus gebruikt om de kennisbank die Clint

afleidt, te controleren. Het is pas wanneer de kennisbank niet voldoet aan de voorwaarden die opgelegd worden door de beperkingen, dat bijkomende voorbeelden geconstrueerd worden die toelaten om de kennisbank verder te wijzigen. Het verschil met de gretige methode bestaat erin dat de gretige methode de beperkingen op een meer aktieve wijze benut. Men kan namelijk ook de beperkingen testen tijdens de afleiding van clausules door Clint. De gretige methode is vrij technisch van aard en wordt daarom hier niet verder toegelicht.

Voorbeeld 3 : het gebruik van beperkingen

Veronderstel dat de beperking `ouder(O,K)` $\rightarrow$ `vader(O,K)` $\vee$ `moeder(O,K)` tegengesproken wordt door de kennisbank voor een substitutie $\theta = \{$ `O = frank, K = lieve` $\}$. In dit geval hebben we dat `ouder(frank,lieve)` geïmpliceerd wordt door de kennisbank en `vader(frank,lieve)` en `moeder(frank,lieve)` niet. De ware toedracht van de zaak kan uitgevist worden door aan een orakel (de gebruiker) te vragen welke atomaire formules waar moeten zijn en welke niet. Stel dat de gebruiker zegt dat `ouder(frank, lieve)` onwaar is. Dan kan deze atomaire formule als negatief voorbeeld aan Clint doorgegeven worden opdat de inconsistentie zou verwijderd worden.

Voor de naïeve en de gretige methoden werd aangetoond dat Clint zijn convergentiegedrag behoudt.

3.3 Het vergaren van objectkennis.

Een frequent voorkomend probleem met het basisalgoritme van Clint is dat een voorbeeld een object o bevat waarvoor geen kennis aanwezig is. D.w.z. dat o in geen enkel afleidbaar feit voorkomt. In zo'n situatie kan Clint natuurlijk geen interessante start-clausule afleiden. (Beschouw het geval dat er geen kennis over het object `alice` in de kennisbank van voorbeeld 2 aanwezig is ; dan zullen de door Clint afgeleide clausules voor het grootouderconcept steeds een ongebonden variabele `O` bevatten.) Daarom werd Clint uitgebreid met twee methodes om kennis over dergelijke objecten te vergaren [Feyaerts 89, De Raedt, Feyaerts en Bruynooghe 91].

De eerste methode steunt op de reeds gekende concept-beschrijving voor het predicaat dat voorkomt in het voorbeeld. We illustreren de eerste methode in voorbeeld 4.

Voorbeeld 4 : kennis vergaren m.b.v. clausules

Veronderstel dat we de kennisbank van voorbeeld 2 gebruiken, maar dat de twee feiten voor `leo` weggelaten zijn en dat de twee volgende clausules toegevoegd werden aan de kennisbank :

`grootvader(V,K)` ← `vader(V,I)` ∧ `ouder(I,K)`

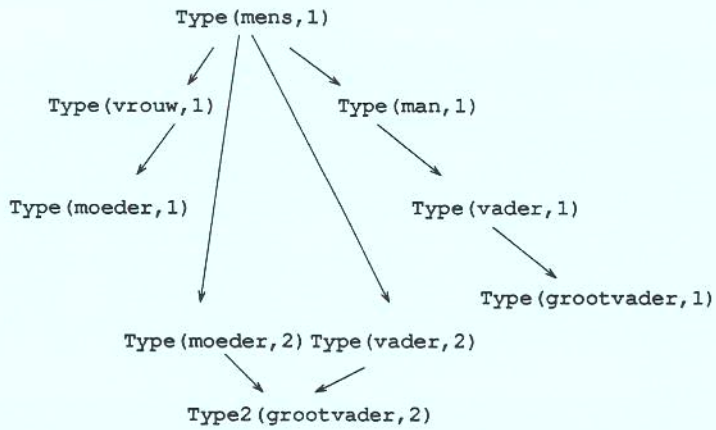
`vader(V,K)` ← `man(V)` ∧ `ouder(I,K)`

Stel dat het systeem als nieuw positief voorbeeld `grootvader(leo,luc)` ontvangt. Dan is `leo` een nieuw object dat in geen enkel afleidbaar feit voorkomt.

In deze situatie zal het systeem eerst het nieuwe voorbeeld unificeren met de bestaande clausules (dit geeft `grootvader(leo,luc)` ← `vader(leo,I)` ∧ `ouder(I,luc)`) en daarna de bewijsbare condities elimineren. Aangezien `ouder(roos,luc)` en `ouder(etienne,luc)` beide waar zijn, resulteert dit in twee mogelijkheden (`vader(leo,roos)` of `vader(leo,etienne)`) om `grootvader(leo, luc)` te bewijzen. Het systeem zal deze niet afleidbare feiten aan de gebruiker tonen. Veronderstel dat die zegt dat `vader(leo,roos)` waar is. Hierop zal het proces recursief verder gaan en zal `vader(leo,roos)` worden geanalyseerd. Dit zal leiden tot de vragen `man(leo)` en `ouder(leo,roos)`. Stel dat ze beide waar zijn, dan zullen deze beide feiten toegevoegd worden aan de kennisbank. Zij worden niet verder geanalyseerd, omdat de predicaten `man` en `ouder` uitsluitend door feiten gedefinieerd zijn. Merk op dat het oorspronkelijke voorbeeld nu bewijsbaar is. In het geval dat noch `ouder(roos,luc)` noch `ouder(etienne, luc)` gekend zouden zijn, zou het systeem eerst vragen naar een `I` zodat `ouder( I, luc)` waar is en eens zo'n `I` gekend, zou het verder gaan met `vader(luc, I)`. Indien er geen `I` is en er geen verder clausules voor `grootvader` bestaan, zou `grootvader(leo,luc)` toegevoegd worden aan de kennisbank. Noteer ook dat indien nodig het systeem backtrackt naar andere clausules of waarden en dat er een heuristiek gebruikt wordt om de interessantste vragen eerst te formuleren.

Indien het predicaat `grootvader` niet gedefinieerd is door clausules, maar wel door feiten, gaat deze methode natuurlijk niet op. We zouden het probleem op eenvoudige wijze kunnen oplossen door het feit toe te voegen aan de kennisbank, maar dit benut de opportuniteit niet optimaal. Daarom gebruiken we voor dit geval de types of soorten van de predicaten [Morik 89, Kietz 88]. Een type is de verzameling van waarden die op een bepaalde argumentpositie van een predicaat in een waar feit voorkomen. Types kunnen in een grafe voorgesteld worden aangezien zij partieel geordend zijn door middel van de *deelverzameling van* relatie (zie figuur 1).

Figuur 1 : een typegrafe



Deze grafe impliceert dat alle objecten die voorkomen op de tweede plaats in een waar feit (gedefinieerd of afleidbaar) voor **vader**, ook voorkomen als een argument van het predicaat **mens** in een waar feit. Het gebruik van de typegrafe voor onze doeleinden wordt geïllustreerd in voorbeeld 5.

Voorbeeld 5 : objectkennis vergaren m.b.v. types

Stel dat we het voorbeeld **vader(leo,roos)** binnenkrijgen met **leo** als nieuw object, dan gaan we eerst kijken naar Type(vader,1). Als de typegrafe correct of stabiel is, dan moet ook **man(leo)** waar zijn en **mens(leo)**. Dit wordt geverifieerd door vragen te stellen aan de gebruiker. Indien nodig kan het systeem de typegrafe automatisch corrigeren. Het kan uiteraard ook voorkomen dat **leo** ook tot Type(grootvader,1) behoort (en tot andere types die niet disjunct zijn met het Type(vader,1)). Dit wordt nagegaan door de vraag **grootvader(leo,I)** te stellen. Indien het antwoord ja is voor een bepaalde waarde van **I**, wordt dit op recursieve wijze behandeld. Bij het afdalen in de grafe wordt, om oninteressante vragen te vermijden, rekening gehouden met een heuristiek gebaseerd

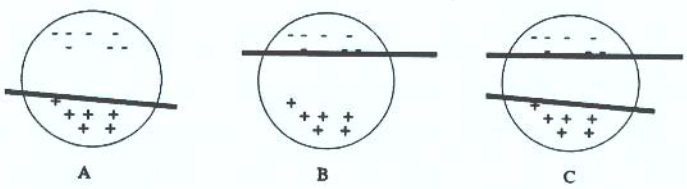
op analogie, de structuur van de typegrafe en ook met types die niet disjunct zijn (in het systeem houden we ook expliciet de niet-lege doorsneden van twee types bij zoals in [Kietz 88]).

We vermelden nog dat de twee methodes voor het vergaren van objectkennis volledig met elkaar geïntegreerd zijn.

3.4 Een driewaardige logica voor Clint.

In de algoritmen van Clint zoals die tot hiertoe gepresenteerd werden, is het onmogelijk om negatieve condities toe te laten in clausules. De belangrijkste reden hiervoor is dat negatie in logische programmeertalen normaal gezien niet constructief is en dat het gebruik van negatie door faling onaanvaardbaar is voor leersystemen. De reden waarom negatie door faling onaanvaardbaar is, is dat ze impliciet steunt op de gesloten-wereld-assumptie (CWA). De CWA gaat duidelijk niet op voor leersystemen, omdat ze veronderstelt dat alles gekend is en dat indien alles gekend is er niets meer dient geleerd te worden. In de praktijk bestempelt de CWA alle uitspraken ofwel als waar, ofwel als onwaar. Dit is niet wenselijk, omdat men in leersituaties steeds over onvolledige informatie beschikt. Dit wordt grafisch geïllustreerd in figuur 2. Hierbij stelt figuur 2.a de situatie van een specifiek naar algemeen leersysteem en de CWA voor, figuur2.b die van een algemeen naar specifiek systeem en de CWA en figuur 2.c die situatie die zich eigenlijk voordoet : er zijn een aantal feiten waarvan we weten (of induceren) dat ze waar zijn, een aantal feiten waarvan we weten (of induceren) dat ze onwaar zijn en een aantal feiten waarvan we niets weten.

Figuur 2 : een driewaardige logica



Een bijkomend probleem is dat het onzorgvuldig gebruik van de CWA aanleiding geeft tot absurde uitspraken zoals `niet(gehuwd (slang(boa-1), stoel(zetel-2)))`. Het probleem hiermee is dat wanneer men dergelijke afleidingen en negatieve condities in clausules geconstrueerd door Clint toelaat, het systeem ook dergelijke absurde en voor de gebruiker enerverende voorbeelden zal genereren.

Beide problemen kunnen vermeden worden door het gebruik van een driewaardige logica. In Clint wordt die gerealiseerd door het afleiden van twee definities voor elk predicaat : een positieve definitie (zoals vroeger), die aangeeft welke feiten waar zijn en een negatieve definitie, die aangeeft welke feiten onwaar zijn. De feiten die dan aan geen enkele definitie voldoen, zijn ongekend. Het invoeren van zo'n derde waarde is een erg natuurlijke manier om de fundamentele onwetendheid voor te stellen. Om de definities af te leiden, gebruiken we het basisalgoritme van Clint om twee predicaten te leren (p en `niet_p`). Hierbij zijn de ware feiten positieve voorbeelden voor de positieve definitie en negatieve voorbeelden voor de negatieve, en de onware feiten negatieve voorbeelden voor de positieve definitie en positieve voor de negatieve. Verder wordt ervoor gezorgd dat de definities van p en `niet_p` elkaar nooit overlappen [De Raedt en Bruynooghe 90d]. Dit kan eenvoudig geïmplementeerd worden door de beperkingen $p \wedge$ `niet_p` $\rightarrow$ `false` toe te voegen aan de integriteitstheorie. Vanzelfsprekend behoudt deze versie van Clint zijn identificatie-eigenschappen.

Er werden ook een aantal varianten van deze driewaardige logica bestudeerd.

3.5 Constructieve Inductie door Analogie.

Een kritiek die vaak gegeven wordt op het leren van concepten door inductie, is dat bestaande leersystemen wel in staat zijn om gegeven concepten te leren uit voorbeelden, maar niet zelf het initiatief nemen tot het definiëren van nieuwe, interessante concepten. Hierbij wordt dan aangevoerd dat het bepalen van welke concepten men wil leren de grootste moeilijkheid vormt. Een tweede interessant punt wordt gevormd door de vraag of men leertechnieken kan toepassen om het leergedrag van leersystemen te leren, met andere woorden kan het leren geleerd worden ? In Clint hebben we gepoogd beide problemen aan te pakken en dit gaf aanleiding tot de Cia (Constructieve Inductie door Analogie) techniek [De Raedt en Bruynooghe 90c, De Raedt en Bruynooghe 89a].

De basisveronderstelling die door Cia gemaakt wordt, is dat concept-beschrijvingen vaak erg gelijkaardig zijn. Met gelijkaardig bedoelen we dan dat ze instantiaties zijn van hetzelfde tweede-orde-schema. Bijvoorbeeld het concept

`lichter(X,Y)` $\leftarrow$ `gewicht(X,XW)` $\wedge$ `gewicht(Y,YW)` $\wedge$ `kleiner(XW,YW)`

is een instantiatie van het volgende schema met substitutie Θ :

$$\Theta = \{\ p = \texttt{lichter},\ q = \texttt{gewicht},\ r = \texttt{kleiner}\ \}$$
$$\exists p,q,r\ :p(\texttt{X},\texttt{Y}) \leftarrow q(\texttt{X},\texttt{XW}) \wedge q(\texttt{Y},\texttt{YW}) \wedge r(\texttt{XW},\texttt{YW})$$

Hetzelfde schema maakt ook abstractie van

`zelfde_kleur(X,Y)` $\leftarrow$ `kleur(X,XC)` $\wedge$ `kleur(Y,YC)` $\wedge$ `eq(XC,YC)`

en

`broers(X,Y)` $\leftarrow$ `zoon(X,XP)` $\wedge$ `zoon(Y,YP)` $\wedge$ `eq(XP,YP)`

De Cia-techniek benut deze gelijkenissen door voor iedere geleerde clausule een tweede-orde-schema af te leiden (verander hiertoe elke predicaatnaam in de clausule in een predicaatvariabele). Wanneer later nieuwe predicaten geleerd worden, wordt de start-clausule c dan vergeleken met dergelijke schema's S. Er zijn dan drie situaties : de clausule c en het schema S komen volledig overeen (d.w.z. $\exists\Theta : S\Theta \subset c$, waarbij S en c gezien worden als verzamelingen van atomaire formules), er is een gedeeltelijke overeenkomst (d.w.z. $\exists\Theta$: *lichaam*$(S)\Theta \subset c$) of er is geen overeenkomst. In het eerste geval wordt het geïnstantieerde schema $S\Theta$ getoond aan de gebruiker en wordt gevraagd of dit de gezochte concept-beschrijving is. Indien zo, wordt het leerproces beëindigd en $S\Theta$ aan de kennisbank toegevoegd. In het tweede geval kan $S\Theta$ niet de gezochte concept-beschrijving vormen, maar wel de definitie van een ander interessant concept. Daarom wordt in dit geval eerst het predicaat geleerd met de normale technieken van Clint en wordt daarna, indien gewenst, het geïnstantieerde schema $S\Theta$ getoond aan de gebruiker. Deze krijgt dan de kans om het concept te benoemen en toe te voegen aan de kennisbank. In het derde geval wordt de normale procedure van Clint gevolgd. Uiteraard werd de techniek geoptimiseerd om redundante vragen te vermijden. Stel dat we bijvoorbeeld als start-clausule hebben :

$$\texttt{grootouder(F,C)} \leftarrow \texttt{man(F)} \wedge \texttt{man(C)} \wedge \texttt{ouder(F,M1)} \wedge \texttt{ouder(M1,C)} \wedge \texttt{vader(F,M1)} \wedge \texttt{vrouw(M1)}$$

en als schema's :

$$\exists p,q : p(\texttt{X},\texttt{Y}) \leftarrow q(\texttt{X},\texttt{XW}) \wedge q(\texttt{XW},\texttt{Y})$$
$$\exists p,q,r : p(\texttt{X},\texttt{Y}) \leftarrow q(\texttt{X}) \wedge r(\texttt{X},\texttt{Y})$$

Dan geeft een volledige overeenkomst aanleiding tot het volgende schema en de vraag:

$$\texttt{grootouder(F,C)} \leftarrow \texttt{ouder(F,M1)} \wedge \texttt{ouder(M1,C)}$$

en een gedeeltelijke overeenkomst tot het voorstellen en de vraag tot benoeming van

$$\texttt{p(X,Y)} \leftarrow \texttt{vrouw(X), ouder(X,Y)}$$
(het **moeder**-concept)

De Cia-techniek gaf aanleiding tot een aantal interessante experimenten in het domein van familie-relaties en voor het leren van de elementaire regels voor schaken. In het familie-domein was het de bedoeling om een dertigtal clausules te leren ; hiervan werden er enkel 3 clausules geleerd met het normale algoritme van Clint, 6 met Cia na een verandering van de taal, 14 direct door Cia en bovendien werden 7 nieuwe predicaten ingevoerd. 12 redundante vragen werden vermeden en slechts 4 nieuwe predicaten waren niet gewenst. De conclusie die hieruit volgt, is dat Cia nuttig is voor domeinen met predicaten die een verwante structuur hebben.

Een bijkomend interessant gegeven is dat het geïntegreerde systeem Clint-Cia een adaptief leergedrag vertoont. Immers, wanneer meer schema's gekend zijn zal het leergedrag van Clint-Cia ook wijzigen. Men kan dus zeggen dat het systeem leert hoe te leren.

3.6 Toepassingen.

In het proefschrift werden ook een aantal mogelijke toepassingen van het leersysteem Clint besproken. Het betreft :

- **Programma-Synthese** : In dit verband werd aangetoond dat Clint een aantal interessante mogelijkheden biedt voor het synthetiseren van programma's uit voorbeelden. Vooral het gebruik van beperkingen lijkt veelbelovend voor dit toepassingsdomein. Er werd ook gewezen op een fundamenteel probleem voor programmasynthese, namelijk dat er geen procedure bestaat die antwoorden genereert op vragen aan de kennisbank en die steeds eindigt.

- **Learning apprentice systemen** : We hebben geargumenteerd dat Clint erg nuttig is als een component van learning apprentice systemen zoals Disciple [Tecuci en Kodratoff 90]. De technieken om objectkennis te vergaren en om beperkingen te behandelen kunnen zeker in een systeem zoal Disciple van pas komen.

- **Autonome Agenten** : Voor autonome agenten en planners die beschikken over onvolledige kennis over de wereld, is een component die leert, onontbeerlijk. Intelligente agenten voeren vaak experimenten uit om hun kennis over de wereld te vergroten. Omdat de voorbeelden die gegenereerd worden door Clint beschouwd kunnen worden als dergelijke experimenten, kan Clint ook gebruikt worden in deze kontekst. Een eerste terreinverkenning van het gebruik van Clint in deze kontekst kan men vinden in het eindwerk van Patrick Lambrix [Lambrix 90].

- **Kennisvergaring** : Heel wat kennisvergaringssystemen zoals Blip [Morik 89] bevatten een leermodule. In Blip wordt die benut om een domein-model af te leiden uit informatie die door de gebruiker wordt ingevoerd. Blip is een passief leersysteem: het wil geen vragen stellen aan de gebruiker. Uiteraard zijn er situaties waarin het stellen van vragen het modeleringsproces aanzienlijk kan versnellen. Daarom lijkt een combinatie van Blip en Clint erg interessant. Wanneer de gebruiker geen vragen wil beantwoorden kan Blip gebruikt worden, en in het andere geval Clint.

- **Verificatie en Validatie van Kennisbanken** : Een belangrijk probleem bij de bouw van kennisbanken is het valideren en verificeren van de afgeleide kennisbank. Clint kan hierbij helpen, omdat het toelaat op een systematische wijze testen (voorbeelden) te formuleren. Wanneer deze testen succesvol blijken, valideren of verificeren zij de beschouwde kennisbank. In het andere geval kan Clint wijzigingen aan de kennisbank voorstellen.

3.7 Conclusie.

We hebben het leersysteem Clint in al zijn aspecten belicht en hopen hiermee de lezer een intuïtief begrip bijgebracht te hebben van de werking van het systeem. We zijn de mening toegedaan dat het systeem Clint heel wat verder gaat dan de meeste andere systemen door zijn gebruiksvriendelijkheid en de mogelijkheid om verschillende vormen van kennis in te brengen. We zijn er dan ook van overtuigd dat Clint een theoretische basis kan vormen voor een praktisch bruikbaar hulpmiddel voor het vergaren van kennis.

Referenties

[Cohen en Feigenbaum 81] P.R. Cohen en E.A. Feigenbaum, uitgevers. *The handbook of artificial intelligence*, volume 3. Morgan Kaufmann, 1981.

[De Raedt en Bruynooghe 89a] L. De Raedt en M. Bruynooghe. Constructive induction by analogy : a method to learn how to learn ? In *Proceedings of the 4th European Working Session On Learning*, p. 189–200. Pitman, 1989.

[De Raedt en Bruynooghe 89b] L. De Raedt en M. Bruynooghe. On explanation and bias in inductive concept-learning. In *Proceedings of the 3rd European Knowledge Acquisition for knowledge based systems Workshop*, p. 338–353, 1989.

[De Raedt en Bruynooghe 89c] L. De Raedt en M. Bruynooghe. Towards friendly concept-learners. In *Proceedings of the 11th International Joint Conference on Artificial Intelligence*, p. 849–856. Morgan Kaufmann, 1989.

[De Raedt en Bruynooghe 90a] L. De Raedt en M. Bruynooghe. Indirect relevance and bias in inductive concept-learning. *Knowledge Acquisition*, 1990. te verschijnen.

[De Raedt en Bruynooghe 90b] L. De Raedt en M. Bruynooghe. Interactief leren van concepten : een overzicht. In *Proceedings van de Nederlandstalige Artificiele Intelligentie Conferentie*, 1990.

[De Raedt en Bruynooghe 90c] L. De Raedt en M. Bruynooghe. Interactive concept-learning and constructive induction by analogy. Technical report, Department of Computer Science, K. U. Leuven, 1990. ingezonden.

[De Raedt en Bruynooghe 90d] L. De Raedt en M. Bruynooghe. On negation and three-valued logic in interactive concept-learning. In L.C. Aiello, uitgever, *Proceedings of the 9th European Conference on Artificial Intelligence*, p. 207–212. Pitman, 1990.

[De Raedt, Feyaerts en Bruynooghe 91] L. De Raedt, J. Feyaerts, en M. Bruynooghe. Acquiring object-knowledge for learning systems. In Yves Kodratoff, uitgever, *Proceedings of the 5th European Working Session on Learning*, 1991. te verschijnen.

[Feyaerts 89] J. Feyaerts. Twee methodes voor het vergaren van object-kennis. Eindwerk, Departement Computerwetenschappen, Katholieke Universiteit Leuven, 1989.

[Genesereth en Nilsson 87] M. Genesereth en N. Nilsson. *Logical foundations of artificial intelligence*. Morgan Kaufmann, 1987.

[Kietz 88] J.U. Kietz. Incremental and reversible acquisition of taxonomies. In M. Linster, B. Gaines, en J. Boose, uitgevers, *Proceedings of the 2nd European Knowledge Acquisition for Knowledge Based Systems Workshops*, 1988.

[Lambrix 90] P. Lambrix. Een planner voor een autonome agent. Eindwerk, Departement Computerwetenschappen, Katholieke Universiteit Leuven, 1990.

[Michalski 83] R.S. Michalski. A theory and methodology of inductive learning. In R.S Michalski, J.G. Carbonell, en T.M. Mitchell, uitgevers, *Machine Learning : an artificial intelligence approach*, volume 1. Morgan Kaufmann, 1983.

[Michalski, Carbonell en Mitchell 83] R.S Michalski, J.G. Carbonell, en T.M. Mitchell. *Machine Learning : an artificial intelligence approach, Volume 1.* Morgan Kaufmann, 1983.

[Michalski, Davis, Bisht en Sinclair 85] R.S. Michalski, J.H. Davis, V.S. Bisht, en J.B. Sinclair. Plant/DS : an expert consulting system for the diagnosis of soybean diseases. In L. Steels en J.A. Campbell, uitgevers, *Progress in Artificial Intelligence*. Ellis Horwood, 1985.

[Mitchell 82] T.M. Mitchell. Generalization as search. *Artificial Intelligence*, 18:203–226, 1982.

[Mitchell, Mahadevan en Steinberg 85] T.M. Mitchell, S. Mahadevan, en L.I. Steinberg. Leap : a learning apprentice for VLSI design. In *Proceedings of the ninth International Joint Conference on Artificial Intelligence*, p. 573–580. Morgan Kaufmann, 1985.

[Morik 89] Katharina Morik. Sloppy modeling. In Katharina Morik, uitgever, *Knowledge Representation and Organization in Machine Learning*, volume 347 van *Lecture Notes in Artificial Intelligence*. Springer-Verlag, 1989.

[Quinlan 86] J.R. Quinlan. Induction of decision trees. *Machine Learning*, 1:81–106, 1986.

[Shapiro 83] Ehud Y. Shapiro. *Algorithmic Program Debugging*. The MIT press, 1983.

[Smith, Mitchell, Winston en Buchanan 85] R.G. Smith, T.M. Mitchell, H.A. Winston, en B.G. Buchanan. Representation and use of explicit justifications for knowledge base refinement. In *Proceedings of the 9th International Joint Conference on Artificial Intelligence*. Morgan Kaufmann, 1985.

[Tecuci en Kodratoff 90] G. Tecuci en Y. Kodratoff. Apprenticeship learning in non-homogeneous domain theories. In Y. Kodratoff en R.S. Michalski, uitgevers, *Machine Learning : an artificial intelligence approach*, volume 3. Morgan Kaufmann, 1990.

[Valiant 84] L. Valiant. A theory of the learnable. *Communications of the ACM*, 27:1134–1142, 1984.