On practical inductive logic programming

Luboš Popelínský

A thesis submitted to the Czech Technical University for
the Doctorate in Artificial Intelligence and Biocybernetics

Czech Technical University in Prague,
Faculty of Electrical Engineering, Department of Cybernetics

July 2000

Abstract

This work focuses on exact learning of logic programs from examples. We introduce a new paradigm of inductive logic programming called assumption-based learning and the corresponding ILP system WiM.

WiM, the system for synthesis of closed Horn clauses, further elaborates the approach of MIS and $Markus$. It works in top-down manner and uses shift of language bias. If negative examples are missing in the learning set, the system itself is capable to find them due to utilisation of the assumption-based learning paradigm. WiM displays higher efficiency of learning as well as smaller dependency on the quality of the learning set than the other exact learners.

A method for automatic building of domain knowledge from object-oriented database schema was developed. This method has been used for solving two database application tasks with WiM, inductive redesign of object-oriented database schema and data mining in spatial data.

Acknowledgements

My first and foremost thanks go to my supervisor Olga Štěpánková for her strong encouragement and support of my research as well as for her patience. Many thanks to Mojmír Křetínský, my boss at Faculty of Informatics, Masaryk University in Brno for his long-term assistance. Special thanks to my wife Eva and my sons Jan a Tomáš for their care, patience and tolerance.

Three events significantly influenced my work. My warm thanks to Pavel Brazdil, LIACC Universidade do Porto, who accepted several times my applications for a leave-of-absence at his group and who – together with Luis Torgo, João Gama and Alípio Jorge and with other members of his group – helped me a lot. The wonderful months spent in L.R.I. Université Paris-Sud in Yves Kodratoff's group strongly influenced not only several parts of this thesis. Thanks to Yves Kodratoff, Céline Rouveilor, Claire Nédellec, Marta Fraňová, Gil Bisson, Marjorie Moulet and Hervé Mignot. I must not forget the first LOPSTR workshop in Manchester 1991. It was Kung-Kiu Lau and Tim Clement who enabled me to be there, and Norbert Fuchs and Pierre Flener who helped me to start my research.

Thanks to my students and colleagues at the Department of Computing Science, Faculty of Informatics, Masaryk University in Brno as well as to all nice people that I met in the last years for fruitful discussions and support.

This work has been partially supported by Esprit LTR 20237 Inductive Logic Programming II ILP^2. My stays in Porto were facilitated by the TEMPUS Project and the ESPRIT METAL Project. French government scholarship allowed me to stay in Paris.

Contents

List of Figures

List of Tables

ACM Classification

D.1.2 Software PROGRAMMING TECHNIQUES Automatic Programming
D.1.6 Software PROGRAMMING TECHNIQUES Logic Programming

I.2.2 Computing Methodologies ARTIFICIAL INTELLIGENCE
Automatic Programming Program synthesis
I.2.6 Computing Methodologies ARTIFICIAL INTELLIGENCE
Learning Concept learning Induction

H.2.1 Information Systems DATABASE MANAGEMENT Logical Design
Schema and subschema
H.2.8 Information Systems DATABASE MANAGEMENT Database
Applications Data Mining
H.2.8 Information Systems DATABASE MANAGEMENT Database
Applications Spatial databases and GIS

Chapter 1

Introduction

We briefly explain exact learning in first order logic. Then we discuss motivation and objectives of this thesis and outline its structure.

1.1 Inductive synthesis in first order logic

This work deals with inductive synthesis of logic programs from examples. The work is, in the first place, application-oriented. It aims at building tools for logic program synthesis from examples and it focusses on the tasks that are solvable by such tools.

Inductive logic programming (ILP) [53, 57], as that particular field of machine learning is called, explores inductive learning in first order logic. An inductive system learns if for a given set of instances of a particular concept it finds a general description of that concept. The main goal of ILP is then development of theory, algorithms and systems for inductive reasoning in first-order logic.

We focus here on exact learning. It means that no noise in input information (examples, domain knowledge) is allowed. We employ the generate-and-test strategy. First a hypothesis is generated that is afterwards tested on examples.

1.2 Motivation

The development of a new ILP system presented in this thesis was motivated by some tasks that everybody must treat when using ILP. It is a selection of learning set (its cardinality and quality), building domain knowledge and optimal settings of bias. Below we summarise some general difficulties that concern those three tasks.

Selection of a training set. Training set should be big enough to ensure successful solving the particular task and it must not contain a lot of unusable (irrelevant, redundant) examples. The main troubles of the ILP systems are:

1. Cardinality of a training set needed by ILP systems seems to be too big [7, 54, 66] and/or the needed quality od examples is extremely high [2, 26, 67]

2. The used negative examples are more dependent on the used ILP system than on the solved task [26, 54, 67].

3. In the case of interactive systems, the obtained result very often depends on the order of examples [67].

Building domain knowledge. Domain knowledge should be rich enough and, at the same time, it should not contain unusable predicates. Two ways how to build the set of domain knowledge predicates has been proposed. The most frequent one is a *selection of the most appropriate predicates* by the user himself. This selection is based on some informal knowledge of the solved task which the user knows. The second, much less used way is *(semi)automatic synthesis of domain knowledge by means of machine learning.* Predicate invention [21, 69] and multiple predicate learning [17] are instances of that approach. Main drawbacks of the existing solutions are the following:

1. The existing solutions hold for a particular task [68], or a particular kind of domain knowledge [30]; or

2. they are relevant only to a particular ILP system [4, 42, 68].

Settings of bias. Optimal settings of bias [56] can hardly be automatic. However, it should be as easy as possible. Finding the optimal parameters of bias is closer to art than to a science [27]. This field is unfortunately much less explored than the choice of bias (parameter settings) for decision tree learners.

1.3 Objectives

In this thesis, we solve only some of tasks that result in more efficient and more user–friendly ILP systems. These tasks are summarised bellow.

Efficient learning from a small learning set. We will show that the number of hypotheses in generate–and–test paradigm can be lowered with bias settings. By this way, this 'brute force' top-down learning will become quite efficient. When looking for the optimal bias settings we will aim at the minimal need of interaction with a user.

Automatic generation of negative examples. As mentioned before, the useful negative examples are very often dependent on the used ILP system. We will describe a semi-automatic method that for finding negative examples. We will show that the found examples are helpful.

Building of reusable domain knowledge. The set of domain knowledge predicates is of course dependent on the particular task that is to be solved by ILP. However, for some kind of applications the domain knowledge (or its part) can be constructed automatically. We will show how to extract the main part of domain knowledge predicates from object-oriented description of learning data.

Applications. We will describe classes of problems that can be solved with exact learners. We focus on the traditional class of problems – logic program synthesis, and database applications. We will show that WiM system can be applied in the field of database schema redesign as well as in the process of knowledge discovery in geographic data.

1.4 Outline of the thesis

The following text can be split into four parts. The first part is the introductory one. Chapter 2 is an introduction to the basic notions of logic programming. Chapter 3 overviews basics of inductive logic programming. The search space, which has to be analysed in ILP setting has clearly exponential character [61]. We show how to decrease this complexity. Chapter 4 displays ILP systems *MIS* , *Markus* and *Progol* as well as other systems mentioned in this thesis.

The second part – Chapters 5, 6 – are as the nucleus of this thesis. Chapter 5 introduces a new paradigm of ILP, **assumption-based learning (ABL)**. Implementation of this paradigm, ***WiM*** system, is described. *WiM* can efficiently learn logic programs from a small example set. Chapter 6 displays the results obtained with *WiM* for both carefully chosen training sets and randomly chosen training examples. Comparison of the results with the results obtained with other ILP systems is displayed.

Next part concerns the process of building reusable domain knowledge and applications. In Chapter 7 the typical applications areas that are solvable with *WiM* are displayed and automatic building of domain knowledge for this kind of applications is described. Next two chapters concern two application areas – inductive redesign of a database schema (Chapter 8) and data mining in spatial data (Chapter 9). Both solutions exploit *WiM* system and the method for automatic building of domain knowledge that is described in Chapter 7.

Last part – Chapter 10 – summaries this work.

Some preliminary results of this thesis were published in different workshops and conferences or as technical reports. Chapters 3 and 4 are extended versions of [64]. Previous versions of *WiM* was described in [24, 59, 61]. Preliminary results of Chapter 8 can be found in [60]. This chapter and some parts of Chapter 7 are substantial extensions of [60, 63]. Chapter 9 is an improved version of [62].

Chapter 2

Logic programming

This chapter briefly introduces the basic logic programming terminology that is used in this thesis. Complete definitions can be found in [34, 43, 48].

2.1 Syntax

A **program clause** (or clause) is a formula $A \leftarrow W$ for which A is an atom, the symbol $\leftarrow$ means an implication, and W is a conjunction of literals, i.e. positive or negative atoms. A is called **head** of the clause and W is called **body**. All variables in the head are universally quantified. A clause (or term) is **ground** if it contains no variable. The **closed clause** is a clause with no free occurrences of any variables, i.e. all variables are bound via quantifiers. We will use also the term **rule** for program clauses with unempty body. If W is absent we call the formula a **fact**. A **logic program**[1] is a finite set of program clauses. A **goal** is a clause of the form $\leftarrow W$ where W is a conjunction of literals. We will use very often :- instead of $\leftarrow$. Queries will be written $\leftarrow W$, or ?-W following syntax of Prolog. Thus

$$p(X, f(X)) :- r(X). \quad (1)$$
$$p(X, g(Y)) :- p(X, Z). \quad (2)$$
$$?- p(a, b), p(a, c). \quad (3)$$

is a logic program in which (1),(2) are program clauses and (3) is a goal. The clause (1) is closed.

[1] If it is unambiguous, we will omit the adjective 'logic'.

The **first order language** given by an alphabet consists of the set of all well-formed formulas constructed from the symbols of the alphabet.

The **definition** of a predicate p appearing in a program P is the set of all program clauses in P which have p in their head. We will need the notion of completion of a program $Comp(P)$. Informally the completion of a program is obtained by replacing implication with equivalence in the rules, and adding the axioms of equality theory.

2.2 Semantics

An **pre-interpretation** J of a first order language L consists of the following:

1. A non-empty set D, called **domain**;
2. for each constant in L , the assignment of an element in D;
3. for each n-ary function symbol in L, the assignment of a mapping from D^n to D.

An **interpretation** I of a first order language L consists of

1. a pre-interpretation J with domain D, and
2. for each n-ary predicate symbol in L, the assignment of a mapping from D^n into $(true, false)$ (or, equivalently, a relation on D^n).

We assume that logical connectives as well as quantifiers have the ordinary semantics ([48] p. 13). Then a formula in L is given a truth value applying the definitions above. An interpretation I is a **model for a closed formula** F if F is true wrt I. Let S be a set of closed formulas and F be a closed formula of a language L. We say F is a **logical consequence** of S if, for every interpretation I of L, I is a model for S implies that I is a model for F. We will write $S \models F$. $F \models G$ iff every model of F is a model of G. I is a **model for the program** P iff I is a model for each clause in P. The **reduced clause** r **of a clause** c is a minimal subset of literals of c such that r and c has the same model.

Let L be a first order language. The **Herbrand universe** U_L for L is the set of all ground terms, which can be formed out of the constants and function symbols appearing in L. If L has no constants, we introduce one, to form

ground terms. Let L be a first order language. The **Herbrand base** B_L for L is the set of all ground atoms which can be formed by using predicate symbols from L with ground terms from the Herbrand universe as arguments. An **Herbrand interpretation** for L is any interpretation on the domain of Herbrand universe U_L. Let L be a first order language and S a set of closed formulas of L. An **Herbrand model** for S is an Herbrand interpretation for L which is a model for S. For **normal programs**, i.e. programs with negative literals in the clause body, we refer to the model of the completion $Comp(P)$ of a program P.

2.3 Answer

A **substitution** θ is a finite set of the form $\{v_1/t_1, ..., v_n/t_n\}$, where $v_1, ..., v_n$ are distinct variables and each term t_i is distinct from v_i. Each element v_i/t_i is called **binding** for v_i. θ is called a **ground substitution** if the t_i are all ground terms. θ is called **variable-pure substitution** if the t_i are all variables. Let P be a normal program and G a goal $\leftarrow W$. An **answer** for $P \cup \{G\}$ is a substitution for variables in W. The answer does not necessarily contain a binding for every variable in G. A **correct answer** is an answer θ such that $\forall(W\theta)$ (all variables in $W\theta$ are universally quantified) is a logical consequence of $Comp(P)$.

2.4 Error diagnosis

A program LP **covers** a fact e if $LP \models e$, i.e. if in every possible interpretation I $LP \models_I e$. It is useful to split the set of predicate definitions LP into 2 parts. Let P be the definition of predicate of the same name and the same arity as the fact e, and $B = LP - P$. Than we can rewrite the above definition in the form $P \cup B \models e$. That notion of coverage is sometimes called **intensional coverage**. In machine learning there is often used the notion of **extensional coverage** of fact e [43]. In the latter all predicates in B are defined extensionally, i.e. by ground facts only.

An **intended interpretation** for a program P is a normal Herbrand interpretation [48] for the completion $Comp(P)$ of P. The aim of logic program synthesis is to find a program which has the intended interpretation as a model.

Let P be a program, G a goal $\leftarrow W$, and I an intended interpretation for P.

1. Program P is **correct wrt** I if I is a model for $Comp(P)$.
2. If θ is an answer for $P \cup \{G\}$ and $W\theta$ is not valid in I, then P is **inconsistent wrt** I.
3. If $P \cup \{G\}$ has a finitely failed SLDNF-tree[48] and W is satisfiable in I, then P is **incomplete wrt** I.

In the following text we will omit the clause "wrt I" if no ambiguity may arise. We say that an instance $A \leftarrow W$ of a program statement in P is an **incorrect statement instance** for P wrt I if A is unsatisfiable in I and W is valid in I. Then we can summarise that P is incorrect wrt I iff there is an uncovered atom for P wrt I or there is an incorrect statement instance for P wrt I.

2.5 Types and modes

We assume that every argument of a predicate have its **type** and an input/output **mode**. The type is a nonempty set of ground terms that the argument can take. The mode says whether the argument is input (its value must be known before an evaluation of the predicate) or output (the value is computed inside the predicate). Let us have $reverse/2$ predicate for reverting lists, e.g. $?\text{-}reverse([k,a,b,a,t],X)$ returns $X = [t,a,b,a,k]$. For this predicate the type and mode definition can be written as $[+list, -list]$, i.e. the both arguments are lists, the mode of the first argument is input (+) and the mode of the second one is output (-).

Chapter 3

Inductive logic programming

This chapter introduces basic notions of inductive logic programming. More attention is paid to general-to-specific framework. We show how to decrease complexity of the search space in ILP setting.

3.1 Basic task of ILP

The goal of inductive logic programming(ILP) is to develop theory, algorithms and systems for inductive inference in first order predicate calculus[1]. Informally, for a given example set and background knowledge we aim at

For given background(prior) knowledge B and evidence $E = E^+ \wedge E^-$ such that	
Prior Satisfiability:	$B \wedge E^- \not\models false$
Prior Necessity:	$B \not\models E^+$
we aim to find a hypothesis H such that the following conditions hold:	
Posterior Satisfiability:	$B \wedge H \wedge E^- \not\models false$ (consistency)
Posterior Sufficiency:	$B \wedge H \models E^+$ (completeness)

Figure 3.1: General setting

finding a hypothesis in first order logic that explain those examples using

[1]This text is based on [53, 57, 67]

the background knowledge. In the general setting, examples, background knowledge and hypotheses may be any formula. More formally, the problem of **inductive logic programming** [53] is displayed in Fig. 3.1. It is reasonable to assume two posteriori conditions [22] $B \wedge E \not\models \neg H$, $B \not\models H$. The main used setting in ILP is **example setting**, or 'specification by examples' [22], where the evidence is restricted to true and false ground facts called examples. Background knowledge is a normal program. The definition in

For given sets of positive and negative examples E^+ a E^- and background knowledge B such that

Prior Satisfiability:	$\forall e \in E^- : \; B \not\vdash e$
Prior Necessity:	$\exists e \in E^+ : \; B \not\vdash e$

find a program P such that

Completeness	$\forall e \in E^+ : \; B \cup P \vdash e$
Consistency	$\forall e \in E^- : \; B \cup P \not\vdash e$

Figure 3.2: Example setting

Fig. 3.2 hold even if P is a conjunction of all the positive examples from the example set. We require in addition that P is a generalisation of examples, i.e. it holds even for examples that do not appear in the example set. In the following text we assume the example setting as default[2].

3.2 Generic ILP algorithm

The definitions above are not constructive. In this section we do the first step to find an efficient algorithm that computes as a result a hypothesis H under the completeness and consistency conditions introduced above. ILP task, as any machine learning task in general [51], can be regarded as a search problem. There is a space of formulas – hypotheses, and the conditions of completeness and consistency form an acceptance criterion on a hypothesis H. So called **enumeration algorithm** can solve ILP simply by a naive **generate and test** algorithm. However, it is out of practical interest because of a computational complexity which is $\mathcal{O}(card\mathcal{L})$ where $\mathcal{L}$ is the set of all

[2]For non-monotonic semantics see [53, p. 636].

possible hypotheses.

The generic algorithm for ILP [53] is in Fig. 3.3. Actual ILP system only differs in implementation of the functions of this generic algorithm. The

Given: $B, E = E^+ \wedge E^-$
$QH :=$ *initialise*$(B, E+, E-)$
while not(*stop_ Criterion* (QH)) **do**
    delete H from QH
    choose the inference rules $r_1, ..., r_k \in R$ to be applied to H
    Apply the rules $r_1, ..., r_k$ to yield $H_1, ..., H_n$
    Add $H_1, ..., H_n$ to QH
    prune QH
Output: *choose_ hypothesis* P from QH

Figure 3.3: Generic algorithm for ILP

algorithm starts with an initial queue of candidate hypotheses QH. In the while loop, a hypothesis H is chosen from the set QH. Then inference rules $r_1, ..., r_n$ are applied to H. It yields $H_1, ..., H_n$ hypotheses that are added into QH and the set QH is pruned. The algorithm stops when the *stop_ Criterion* holds on the set QH.

The algorithm has the following parameters:

- The *initialise* function builds the initial portion of hypotheses ;
- R is the set of inference rules applied;
- Different instantions of *delete* allow to implement a search strategy: a depth-first (*delete*=LIFO), breadth-first (*delete*=FIFO), best-first (e.g. *delete* H such that $P(H|B \wedge E)$ is maximal $\forall H^{'} \in QH$) ;
- *choose* determines what inference rule to apply on H ;
- *prune* determines what hypotheses to delete from the queue of QH;
- The *stop_ Criterion* holds if an adequate solution has been found, or the QH queue is empty;

- *choose_hypothesis* chooses from QH one of possible solutions[3].

Any of advanced search strategies (hill-climbing, beam-search etc.) can be realized by *delete* and *prune* together with *choose* and *stop_Criterion*.

In the next section we focus on general-to-specific framework which is further elaborated in this thesis. We first introduce the notion of specialisation in logic and describe the generic algorithm for general-to-specific ILP. We explain the notion of a refinement operator and a refinement graph. We conclude with brief summary of bias.

3.3 General-to-specific ILP

3.3.1 Specialisation

We say that F **is more specific than** G [53] iff $G \models F$. We will write $F \preceq G$. It means that any model of G is a model of F. F is called a **specialisation** of G. A **specialisation operator** maps a conjunction of clauses G into set $\mathcal{S}$ of maximal specialisations. A **maximal specialisation** S **of** G (also called the most general specification) is a specialisation of G such that G is not specialisation of S, and there is no specialisation S' of G such that S is a specialisation of S'.

Example: If the set of function symbols F contains only the element a, and $G = p(X, Y)$, then the set $\mathcal{S}$ of all maximal specifications of the clause G is containing $p(a, Y), p(X, a)$, but not $p(a, a)$.

The notions of generalisation, maximal generalisation and generalisation operator are defined as inverse to those ones [53, p.642] concerning specialisation.

3.3.2 Specialisation operators in first-order logic

Most of ILP systems which work in general-to-specific manner employ two operators of specialisation

- **binding of 2 distinct variables**
  e.g. for a predicate $p/2$ we have $spec(p(X, Y)) = p(X, X)$;

[3]This function is not explicitely introduced in [53], however we find it important.

- **adding a most general literal to a clause body** (arguments are so far unused variables). Let the domain knowledge contains only one predicate $q/3$. Applying this rule we obtain two specialisations of $p(X, Y)$
  $spec(p(X, Y)) = p(X, Y) \leftarrow p(U, V)$
  $spec(p(X, Y)) = p(X, Y) \leftarrow q(U, V, W)$;

If working with programs that contain constants and complex terms we need two more operators

- **replacing a variable with a constant**
  e.g. $spec(p(X, Y)) = p([\,], Y)$;
- **replacing a variable with a most general term** (arguments are so far unused variables)
  e.g. $spec(p(X, Y)) = p([U|V], Y)$;

Although the definition of a minimal specialization is simple when learning one clause, it is not so clear when refining a whole theory. The problem of what clause to choose for further specialization newly appears [70].

3.3.3 General-to-specific algorithm

Now we can introduce the generic algorithm for general-to-specific ILP. Let us start with the generic ILP algorithm (Fig 3.3). Let the training set contains examples of predicate $p(X_1, X_2, ..., X_n)$. In the case of general-to-specific algorithm, the *initialise* function returns the most general clause $p(X_1, X_2, ..., X_n)$:- $true$. and the set of inference rules consists of four specialisation rules from Section 3.3.2.

A systematic investigation of specialisation operators in logic programming was started by Ehud Shapiro [67]. The most important notions and characteristics are summarised in the next section.

3.4 Refinement operator

3.4.1 Definition

In the definition below we assume a specific language $\mathcal{L}$ is used. Without loss of generality, we assume $\mathcal{L}$ has a most general element $\top$.

Def.: A **refinement graph** is a directed, acyclic graph in which nodes are clauses [4] and arcs correspond to refinement operations, defined bellow.

Def.: Let $\mathcal{C}$ be a set of clauses and ρ a mapping from $\mathcal{C}$ to finite subsets of $\mathcal{C}$. We define $<_\rho$ to be the binary relation over $\mathcal{C}$ for which

> $p <_\rho q$ iff there is a finite sequence of clauses $p_1, p_2, ..., p_n$ such that $p_1 = p, p_n = q$, and $p_{i+1} \in \rho(p_i)$ for $O \leq i < n$.

We say that $p \leq_\rho q$ iff $p <_\rho q$ or $p = q$ (we don't distinguish between clauses that differs only in the variable names).

The mapping ρ is said to be a **refinement operator** over $\mathcal{C}$ iff the following two conditions hold:

1. The relation $<_\rho$ is a well-founded ordering over $\mathcal{C}$.
2. For every interpretation I and goal G, if q covers G in I and $p <_\rho q$ then p covers G in I.

operator. It can be proved [67] that the operators of specialisation from Section 3.3.2 together make a refinement operator.

An example of the refinement graph for $reverse(X, Y)$ is in Fig 3.4. A root of the graph is the most general clause $reverse(X, Y)$. Two clauses C_1, C_2 are connected with an edge if the clause C_2 arises from C_1 after application of one of the specialisation operators. Clauses of the correct definition of $reverse/2$ predicate are printed in bold. To be brief, some unimportant parts of the graph are missing. E.g. on the first level the replacement $X/[]$ as well as adding the literal $reverse(A, B)$ are left out.

3.4.2 Properties

A refinement operator ρ (with transitive closure ρ^*) is

1. **globally complete** for a language $\mathcal{L}$ iff $\rho^*(\top) = \mathcal{L}$.

[4] Shapiro [67] defined the refinement graph for definite clauses. We use here more general definition for program clauses.

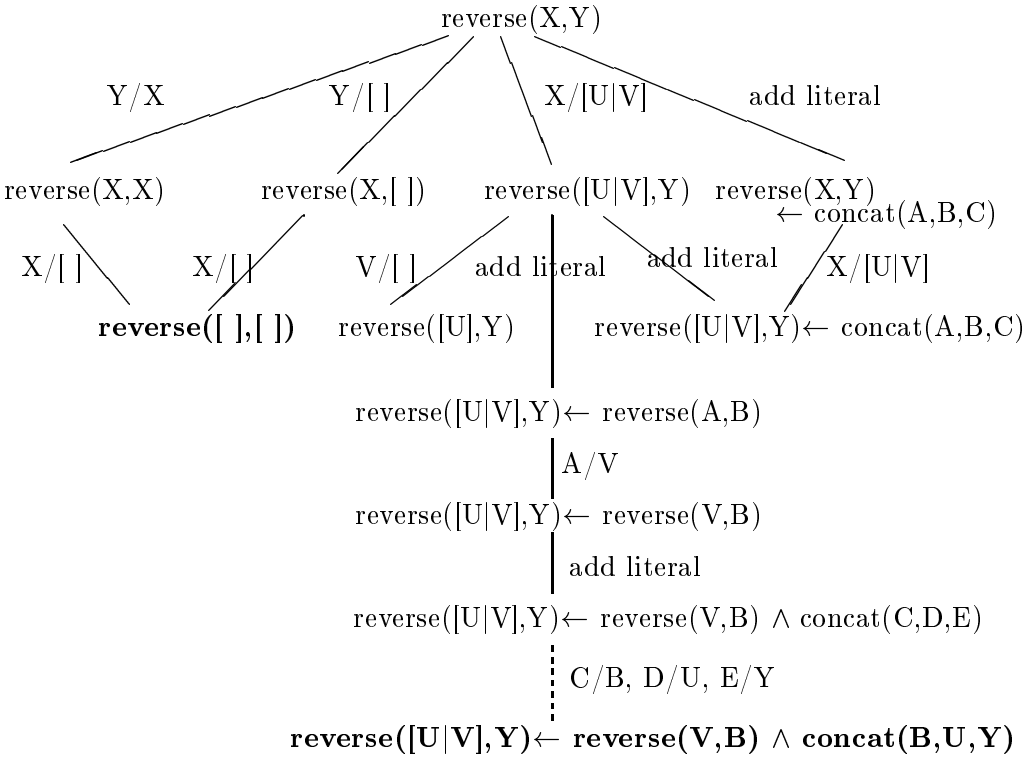

Figure 3.4: Refinement graph for $reverse/2$

2. **locally complete** for a language $\mathcal{L}$ iff $\forall c \in \mathcal{L} : \rho(c) = \{c' \in \mathcal{L} \mid c' \text{ is a maximal specialisation of } c\}$.

3. **optimal** for a language $\mathcal{L}$ iff $\forall c, c_1, c_2 \in \mathcal{L} : c \in \rho^*(c_1)$ and $c \in \rho^*(c_2) \rightarrow c_1 \in \rho^*(c_2)$ or $c_2 \in \rho^*(c_1)$

Let us have a language $\mathcal{L}$ where $\mathcal{P}, \mathcal{F}$ are finite sets of predicate and function symbols. There is a ρ_0 refinement operator that is globally complete for the language. It was proved in [67] that two syntactic operations are enough for ρ_0:

1. Instantiate a clause.
2. Add a goal to the condition of a clause

3.5 Bias

The computational complexity of ILP algorithm is an important problem. In general there are three ways how to limit the size of the set generated by a refinement oprator: to define **bias** (syntactic as well as semantic restrictions on the search space) [27, 42, 56], to accept assumptions on the **quality of examples**[46], or to use an **oracle**[12, 13, 67]. Even in the case of a finite relation we assume that the number of examples is (significantly) less than the number of all instances of the relation.

Bias which is discussed in this section is usually split into two groups, **language bias** that narrows the space of possible solutions and **search bias** that defines how to search that space and when to stop.

Language bias. These constraints defines a form of possible solutions. More frequent constraints limit the maximal number of clauses in the solution or maximal number of literals in a clause body [26, 54, 67]. Languages were developed [6, 11] that enable to define almost any syntactic feature of the desirable solution.

Search bias. It says how to search the graph of potential solutions. It also defines the condition under which the search are to stop. The latter is sometimes called validation bias.

Shift of bias. Most of characteristics of bias – like the complexity of the intended solution, or the maximal number of nodes-hypotheses in the search space – may be expressed via parameters. Usually it is uneasy to set the parameters optimally. The techniques of the shift of bias can help. We start with such a setting that defines the minimal search space that is reasonable. If the solution is not found in that space the bias can be weakened, i.e. the search space is increasing.

It is clear that the complexity fo learning strongly depends on the bias settings. In the next section we show how to decrease the cardinality of the search space in top-down learning algorithms. We try to estimate cardinality of the search space as a function of the size of the background knowledge and of the maximal length of clause bodies.

3.6 Cardinality of the search space for given settings

3.6.1 Upper estimate

Let $|BK|$ mean the number of background knowledge predicates + 1 (for the target predicate), A the highest arity among the predicates in background knowledge and the target predicate, L the maximal length of a clause body, i.e. the maximal number of predicates in a clause body.

The number of positions of variables in a clause for a given length l is equal to a sum of the positions in the head and in the body, its upper bound is $(1+l)*A$. E.g. for $member/2$ predicate, maximal length of the clause body for $l = L = 2$ and background knowledge which contains only `list(List, HeadOfList, BodyOfList)`[5] we have

```
member(X1,X2) :- P1(X3,X4,X5),P2(X6,X7,X8)
```

i.e. 8 positions.

Now we find the number $NC(n)$ of clauses for a given number n of variable positions. Having a set of variables {`X1,X2,X3,X4,X5,X6,X7,X8`} as in the example above, we can find all different clauses for fixed $P1$, $P2$. The number of those clauses is less than 8^8 because some of them are equivalent, e.g. `member(X,Y) :- member(X,Z)` is the same as `member(U,V) :- member(U,W)`. See Appendix A for detailed treatment of this subject. In Table 3.1 the values of $NC(n)$ for small values of n are displayed. As each combination of background predicates as well as the target predicate can appear in the body, we have to multiply $NC(n)$ by the number of all allowed combinations of predicate symbols. E.g. for $member/2$ predicate

```
member(X1,X2) :- P1(X3,X4,X5),P2(X6,X7,X8)
```

we have 2 positions for predicates. If the maximal length of the clause body is $L = 2$ and the maximal arity $A = 3$, the number of all clauses in the search space is the sum of number of clauses of the length 0 (body == true), of the length 1 and 2

$$NC(A)+2*NC(2A)+3*NC(3A) = 2+2*52+3*21147 = 63547$$

[5] `list(List, HeadOfList, BodyOfList)` splits `List` into its head `HeadOfList` and its body `BodyOfList`.

Variable positions	Clauses	Variable positions	Clauses
1	1	6	203
2	2	7	878
3	5	8	4140
4	15	9	21147
5	52	10	115975

Table 3.1: $NC(n)$ for small values of variable positions

The coefficients 2 and 3 are equal to the number of combinations with repetition of possible predicates in the clause body for a clause with its body length 1 and 2 respectively.

The general formula for the number of all clauses for given BK, L, A is

$$NCA = \sum_{l=0}^{L} \left(\begin{array}{c} |BK| + l - 1 \\ l \end{array} \right) * NC((1 + l) * A),$$

This formula inherits its exponential character from the function NC (see Appendix A). That is why we need more information to decrease cardinality of the search space. The declaration of types and the maximal number of free variables allowed during learning can help. It was shown elsewhere [16], that we can focus on linked clauses only. It limits the number of distinct variables significantly. In the next paragraph we show how to exploit such information to narrow the search space.

3.6.2 How to narrow the search space

We will demonstrate a way of narrowing search space on a simple example. Let us learn the base clause of the predicate *member*/2. The `list(List, HeadOfList,BodyOfList)` predicate is the only background knowledge predicate. Suppose we know that the maximal length of the body of the clause is 1. Then two skeletons have to be considered as candidates

(1) member(_,_)
(2) member(_,_) :- list(_,_,_)

For (1) and (2), there are $NC(2) = 2$ and $NC(5) = 52$ instances respectively, in total 54 clauses in the search space.

Let us assume that only 1 free variable may appear in the body of the clause. For the case (2) it implies introduction of 1 new variable by predicate `list/3` so that no more than 3 distinct variables are allowed. The number of clauses is than $h(1,5) + h(2,5) + h(3,5) = 41$.

If we know types of arguments, `member(nom,list)`, `list(list, nom,list)` in our example, the search space is further narrowing because `member(X,X)` cannot appear. The remaining `member(X,Y)` is not taken in account as it is the most general clause and it could not be consistent - any negative example would be covered by this clause. When taking in account the type definition, the case (2) may be split into two cases (2a) and (2b)

(2a) `member(X,Y) :- list(L1,X,L2)`
(2b) `member(X,Y) :- list(L1,U,L2)`

where U $\neq$ X. In (2a), as at most 1 free variable can be introduced, one of L1,L2 must be equal to Y or L1=L2. It means that only following 4 clauses remain

```
member(X,Y) :- list(Y,X,L1)
member(X,Y) :- list(L1,X,L1)
member(X,Y) :- list(L1,X,Y)
member(X,Y) :- list(Y,X,Y)
```

For (2b), as U $\neq$ X is a free variable, both L1 and L2 have to be identical to Y, and just the single clause `member(X,Y) :- list(Y,U,Y)` remains. It means that the search space consists of 5 clauses only - this compares well to the number of 54 clauses estimated in the beginning[6].

To summarize, the search space shrinks considerably when we do exploit knowledge about the maximal length of a body of the clause, the maximal number of free variables allowed and type declarations of arguments.

Based on the idea of narrowing search space displayed in this section we implemented top-down ILP system *WiM* which is described in Chapter 5. We will show in Chapter 6 that this way of narrowing search space is sufficient enough for a class of list processing predicates.

[6]Knowing more about semantics of the `list/3` predicate we can also delete clauses `member(X,Y) :- list(L1,X,L1)` and `member(X,Y):-list(Y,X,Y)`.

Chapter 4

ILP systems

We introduce general-to-specific ILP systems that are as predecessors of WiM system. The basic information on MIS is followed by description of $Markus$ system. We displays also other systems for exact learning – $FILP$, $CRUSTACEAN$, $SKILit$, and also $Progol$ system, that were used for comparison with WiM system.

4.1 MIS

4.1.1 Overview

MIS - *Model Inference System* [67] is the first system for logic program synthesis from examples. It is an interactive multiple predicate learner that employs the general-to-specific strategy. The process is incremental, i.e. the clause that has been rejected as inconsistent will not be given again into the solution. It needs information about learned predicates, namely modes and types of arguments and names of domain knowledge predicates that may appear in the learned predicate definition. Each predicate may be declared as total (for each input there exists at least one output) and/or determinate (for each input there is at most one output). Then MIS process examples one-by-one resulting in a logic program that is complete and consistent with respect the known examples. During the program synthesis MIS may ask queries about intended interpretation of subgoals. The alorithm of MIS is based on error diagnosis in logic programs. For a ground goal (new example) e and a program P (that has been synthesised) some of the following errors can be detected:

1. INCONSISTENCE.
   The program P covers a negative example : $e \in E^- \wedge P \vdash e$. The clause that was detected as incorrect is deleted and MIS looks for next candidate clause.

2. INCOMPLETENESS.
   The program P does not cover a positive example : $e \in E^+ \wedge P \nvdash e$. MIS looks for a new clause that covers the example e, does not cover any negative example and is the most general from all possible clauses that have not been tested (Section 4.1.3). That clause is appended to the solution.

3. DIVERGENCE.
   The program P is looping. The metainterpreter built-in in MIS is given a maximal depth of computation of a goal. If that maximal depth is exceeded the algorithm continues as in the case of inconsistence.

4.1.2 Algorithm

Input: B, E^+, E^-, *bias*
$QH := \{\}$; The set of marked clauses is empty;
repeat
  read an example;
  repeat
    if program QH fails for a positive example
      find a new clause C so that $QH \cup \{C\}$ covers the example;
      $QH := QH \cup \{C\}$;
    if program QH suceeds for a negative example
      find the incorrect clause C of program QH; $QH := QH - \{C\}$;
      mark clause C ;
  until QH is complete and consistent
  write QH
until all examples from $E^+ \cup E^-$ have been read
Output: the sequence of programs $QH_1, QH_2, ...$ that are complete and consistent with respect known examples

Figure 4.1: Schema of MIS algorithm

In the algorithm in Fig. 4.1 marked clauses are the clauses that have already been found inconsistent.

4.1.3 Refinement operator

The construction of a new clause starts with the most general clause – a head of the clause contains distinct variables, and the body is *true* – which is further specialised. We will demonstrate the whole process on the example of synthesis of predicate *reverse*/2 (Fig. 4.2). Both arguments are declared as lists, the first one is an input, the second one is an output. The domain knowledge contains only predicate $concat(E, L1, L2)$ that appends the element E to the list $L1$ resulting in the list $L2$. Let us suppose that the base clause `reverse([],[])` has been already found. The training set contains following examples

```
reverse([1,2],[2,1]), true
reverse([1,2],[1]), false
```

The root of the refinement graph is the most general clause `reverse(X,Y)`. It can be specialised by **unification of an output variable with an input variable in the head of the clause**. However `reverse(X,X)` does not cover a new positive example. The next refinement operation is **substitution of input variable from the head to a constant**. The only constant for the type of list is `[]` but `reverse([],X)` is rejected for the same reason as above. Neither `reverse(X,[])` can be accepted which arises from **substitution output variable from the head to a constant**. Next rule that **replaces a variable with the most complex general term**[1] results in `reverse([H|T],Y)`. This clause is acceptable because it covers the uncovered positive example. However, it need to be further specialised because of inconsistency.

The next refinement rule **adds a subgoal to the clause body** . This rule is applied twice resulting in `reverse([H|T],Y) :- reverse(T,S), concat(H,S,I)`. Input arguments of that subgoal are selected from the input variables(arguments) that already exist. New variables, i.e. output variables of that subgoal, are added to the list of input variables and also into the list of free variables. They may be used as input variables in next subgoals. If a free variable is used as an input one, it is deleted from the list of free variables.

The rest of free variables can be deleted from the list only with the refinement operation of **closing a clause**. If an output variable is unified with a free

[1] Let us notice that the above rules can be applied only to clauses without subgoals.

```
reverse(X,Y).
input variables  = <X, list>
output variables = <Y, list>
                   |
                   |   X /[H|T]
                   V
reverse([H|T],Y).
input variables  = <H, integer>, <T, list>
output variables = <Y, list>
                   |
                   |   add_subgoal reverse(T,S)
                   V
reverse([H|T],Y) :- reverse(T,S).
input variables  = <H, integer>, <T, list>,
                    <S, list>
output variables = <Y, list>
volne variables   = <S, list>
                   |
                   |   add_subgoal concat(H,S,I)
                   V
reverse([H|T],Y) :- reverse(T,S), concat(H,S,I).
input variables  = <H, integer>, <T, list>,
                    <S, list>, <I,list>
output variables = <Y, list>
free variables   = <I, list>
                   |
                   |  Y/I (close clause)
                   V
reverse([H|T],I) :- reverse(T,S), concat(H,S,I).
```

Figure 4.2: MIS : A new clause synthesis

variable, the free variable is deleted from the list. The operation of closing a clause may be applied only in the case that it results in exhausting of the list of free variables.

The whole synthesis of the recursive clause for *reverse*/2 in in Fig. 4.2. As the base clause `reverse([],[])` has been already synthesised, the clause `reverse([H|T],Y) :- reverse(T,S), concat(H,S,I)` is accepted even if example `reverse([2],[2])` is not in the example set. The final clause is complete and consistent with respect to the example set.

4.1.4 Discussion

The main problem of *MIS* is the right choice of examples and even their order. User actually has to know the intended program and even to understand well *MIS* algorithm. The implemented refinement operator does not prevent from duplicate nodes in the refinement graph. Mat Huntbach [31] partially solved that problem by ordering the refinement operations. However, duplicate clauses may still appear within one refinement operation. The problem was fully solved in *Markus* [26] (Section 4.2). Extension of *MIS* for synthesis of inpure Prolog programs is suggested in [58].

4.2 *Markus*

4.2.1 Overview

Markus [25, 26, 27] extends Shapiro's Model Inference System in some directions. In noiseless domains it is still competitive with younger systems. Following [26], the main features of Markus can be summarised as follows:

- optimal generation of a refinement graph (no duplicate nodes) (continuing the work [31]),
- use of the iterative deepening search of the refinement graph,
- controllability of the search space with several parameters,
- the covering paradigm, and
- learning in the batch (non-interactive) mode.

4.2.2 Algorithm

It is non-interactive and employs **covering paradigm**. It means that it looks for partition of positive example set on mutually disjunctive parts. Each of those part will be then described by one clause. The learning algorithm finds a first clause that covers at least 1 positive example and uncover no negative one. The covered positive examples are then deleted from the training set and the learning algorithm continues in covering that smaller example set. The algorithm stops when all positive examples and no negative examples are covered, or no solution was found in the search space.

A synthesis of a new clause processes in 3 steps. A newly constructed clause is first tested whether it is **promising** , i.e. it covers at least 1 positive example from the current example set. After closing the clause (see closing refinement operator in *MIS*), the clause is tested whether it is still promising and in the same time whether it does not cover any negative example. Such clause, said as **good** one, is appended to the end of current result. The result is then tested to be complete and consistent.

4.2.3 Refinement operator

The refinement operator used in Markus is based on MIS' one. In addition it can introduce negative goal in the clause body in the case that all variables are input ones. By further ordering of modifications within one refinement operation it solves eventually the problem of duplicate nodes. In this sense the *Markus* refinement operator is optimal.

4.2.4 Parameters

Parameters of *Markus* may be split into three groups. The first group corresponds to language bias mention earlier. The second one parametrises search of hypotheses space and third one concerns inputs and outputs of *Markus* . Language bias may be defined with following parameters:

- maximal number of goals in clause body,
- maximal depth of arguments in the head of a clause.
- maximal number of free variables (newly introduced, but still unused variables in the body of a clause),
- maximal number of clauses in the result.

Besides those, user may choose a kind of refinement operator – the operator for the language of definite clause grammars or the refinement for normal programs.

4.2.5 Discussion

When settings of parameters of bias are good, *Markus* can learn simple list processing predicates as well as Peano's arithmetic operations from no

more than 4 examples, mainly due to strong focus on modes of arguments. However, the optimal choice of bias is very difficult to find without knowing the right solution in advance.

4.3 Other systems

In the following sections we introduce theree systems that are used for comparison with *WiM* (Chapter 6). After a brief description the most important faults of those systems are displayed.

4.3.1 *CRUSTACEAN*

CRUSTACEAN [2] learns recursive programs from a small number of examples. The goal of that project was to develop efficient ILP system with a strong language bias and without any need background knowledge. The learning algorithm is based on analysis of argument structure in positive examples and generalisation of found similarities. It starts in finding all possible **subterms** of the arguments. We will explain it on the example `last(a,[c,a])`. The first argument has no subterms except itself but `[c,a]` can be decomposed into `[c,a]`, `c`, `[a]`, `a`, `[]`. Each subterm can be obtained by applying a **generating term** to the particular argument. A generating term is a sequence of decomposition operators. For the domain of lists we need only one decomposition operator `[_|_]` that splits a list into its head and its body. E.g. for the term `[c,a]` the generating term for the subterm `[a]` is `[_|X]` as `[a]` can obtained from `[c,a]` after evaluation of the goal `?-[c,a]=[_|X]`. The **depth** of the generating term is 1. The generating term of `a` is `[_|[X|_]]`. because we need to compute `?- [c,a]=[_|[X|_]]`. (depth 2). The programs learnable with *CRUSTACEAN* have the form

$$P(A_1, ..., A_n). \qquad \text{(B)}$$
$$P(B_1, ..., B_n) :- P(C_1, ..., C_n). \qquad \text{(R)}$$

where A_i, B_i, C_i are terms and (1) there exists at least one i such that for $B_i \neq C_i$ C_i is not ground and C_i is a subterm of B_i, or (2) $B_i = C_i$.

CRUSTACEAN needs two positive examples P_1, P_2. Each of them can be proven by resolving a specialisation B_1 of the base clause B and the recursive clause R repeatedly. First, all subterms and corresponding generating terms for all positive examples are computed, The base clause is induced as the

least general generalisation(lgg) [53] of atoms that arise from aplication of the generating terms to positive examples. Then we apply the generating terms to the examples 0, 1, ..., n-1 times where n is a depth of generating terms. The head of the recursive clause is obtained as lgg of atoms obtained by this way. The recursive literal in the body is received by applying generating terms to the head.

CRUSTACEAN , as mentioned in the begining, is not capable to exploit any domain knowledge predicates. It implies that examples sometimes must contain unusual terms, like `reverse([1,2],append(append([],[2]),[1]))` or `factorial(s(s(s(0))),s(s(s(0)))*s(s(0))*s(0))`. The language of hypotheses is very restrictive. Obviously, *CRUSTACEAN* returns more than one solution.

4.3.2 *FILP*

FILP [7] is an interactive system for learning functional logic programs from positive examples. A logic program is functional if for each sequence of input arguments there exists just one sequence of output arguments. The queries of *FILP* are existential queries [3, 67] with unbound output variables. As the learned predicate is always functional, there is at most one answer to every query. *FILP* needs only positive examples. Negative ones are those that have the same input values as positive examples but different outputs.

The class of learned programs is a subset of logic programs. Any recursive clause must namely match the schema

$$P(X_1, ..., X_i, ..., X_n) : -..., Q(X_i, Y), P(X_1, ..., Y, ..., X_n),$$

where $Q(X_i, Y)$ defines a well ordering between Y and X, i.e. $Y < X$. User has to choose one initial example that is complex enough. *FILP* then asks for intended interpretation of all needed subgoals with "smaller" arguments (with respect that ordering). E.g. for `union([a,b],[a,c],[b,a,c])` the system asks for `union([b],[a,c],X)` and for `union([],[a,c],X)`.

The main advantages of *FILP* are two: it allways finds a program consistent with examples if such a program exists; and that program computes only correct outputs on inputs of given examples. The drawback of *FILP* is its need of extensional definition of domain predicates and its incapability to work with unflatten clauses.

4.3.3 *SKILit*

SKILit (SKetch-based Inductive Learner with ITerative extension) [32, 33, 34] builds each clauses by searching for relation link between input and output positive examples. It employs **aglorithm sketches** and **clause structure grammar**. E.g. for $reverse/2$ an algorithm sketch may be

$$\begin{aligned} reverse(+[3,2,1],-[1,2,3]) \leftarrow\ & \$P1(+[3,2,1],-3,-[2,1]),\\ & reverse(+[2,1],-[1,2]),\\ & \$P2(+3,+[1,2],-[1,2,3]). \end{aligned}$$

where $\$P1, \$P2$ are predicate variables. The clause structure grammar enables to define a schema – divide-and-conquer, generate-and-test – that a result of *SKILit* has to match. *SKILit* is capable to process integrity constraints, e.g.

$$reverse(A,B), length(A,N) \rightarrow length(B,N)$$

using Monte Carlo strategy. Randomly chosen facts that are consequences of the learned program are tested whether they violate an integrity constraint.

To be able to learn recursive definitions, *SKILit* implements an **iterative induction**. In the first iteration a nonrecursive program $P1$ is synthesised by generalisation of examples. The found clauses are in the second step used as **properties** (nonground partial specifications of a program, e.g. $reverse([X,Y],[Y,X])$) [22] for generation of new examples. Then a new clause is learned and added to program P_1 resulting in P_2 etc.

SKILit is not yet a usefull tool for practical logic program synthesis. [34]. Here we mention only some of its drawbacks. It is unefficient for larger sets of domain knowledge predicates if user is not capable to write enough strong sketches and/or to limit language bias using rules of the structure clause grammar. *SKILit* seems to be dependent on presentation order of positive examples. Negative examples have to be carefuly chosen to prune the search space.

4.3.4 *Progol*

Progol [54] is a bottom-up learner that can learn from noisy data. It chooses one or more positive examples from a training set and constructs their least

general generalisations with respect domain knowledge. Then each of generalisations is further generalised and that one is choosen that maximaly compress other positive examples. *Progol* employs a covering paradigm. It means that all the process repeats until all (but a small fraction of) positive examples are covered and none (but a small fraction of negative) examples are not covered. The degree of incorrectnes and inconsistence is driven by *Progol* parameters. As *Progol* employs a heuristic search for a space of clauses driven by a compression measure, it is not too convenient for logic program synthesis tasks. However, *Progol* is one of the best empirical ILP systems. Thanks to rich set of parameters it can be easily adapted to solving various tasks. *Progol*'s ability of constant introduction in clauses needs to be mentioned here. It allows to limit a number of domain knowledge predicatesis and it increases *Progol* efficiency.

Chapter 5

Assumption-based ILP and WiM system

We introduce a new paradigm of assumption-based learning. Then we describe the general-to-specific learner WiM as an instance of this paradigm which can efficiently learn logic programs from a small example set.

5.1 Introduction

Considering top-down exact learners in the context of automatic logic programming [23], four main drawbacks are being observed:

1. Too many positive examples are needed
2. The usefulness of the negative examples depends on the particular learning strategy
3. Generate (a hypothesis) and test (on the example set) strategy is too unefficient
4. Too many queries to the user are asked

We will show that even with a very small example set (less or equal to 4 positive examples) MIS-like [67] top-down learners are capable to learn most of the predicates which have been mentioned in ILP literature.

We here focus to top-down exact learners that employ the generate-and-test

strategy. They generate a clause (for given bias) that need to be tested afterwards on the example set. It means that a number of tests during one learning session is proportional to a number of generated clauses multiplied by a cardinality of the example set. However, the number of clauses can be limited by a declaration of argument modes and types and by exploitation of programmer's knowledge, as shown in Section 3.6.2. This knowledge can specify e.g. the maximal complexity of the learned logic program, like a maximal number of clauses in the program or a maximal number of literals in a clause body. We strongly believe that such knowledge is available very often. By this way we lower cardinality of the set of negative example, and such a modification of 'brute force' top-down learning algorithm is becoming quite efficient.

Necessary negative examples are always dependent on the particular learning strategy and that is why it is difficult for the user to find the most appropriate ones. Our approach tries to find negative examples by itself. A near-miss to one of the positive examples is considered as a candidate for that purpose. Such a negative example is found useful if after adding that example to the current learning set, the learner is able to suggest a new definition of the target predicate. Only in such a case the user is asked for a confirmation of that particular candidate for the negative example.

Ideas on an assumption-based framework inspiring our methodology may be found in [10, 15, 35, 36]. We developed a new method called **assumption-based learning** (ABL) based on those ideas. A generic scenario of assumption-based learning consists of three parts, an *inductive synthesiser*, a *generator of assumptions* which generates extensions of the input information and an *acceptability module* which evaluates acceptability of assumptions. That module is allowed to pose queries to the teacher. It may happen that the inductive synthesiser have failed for any reason to find a correct solution (e.g. because of missing examples, insufficient domain knowledge or because of too strong bias). Then ABL system is looking for such a minimal extension of the input - called assumption - which allows to find a solution. The solution has to be correct and consistent with the input extended by the new assumption. If an assumption is needed, it must be confirmed by the acceptability module. It is true that the query to the user is necessary to confirm the assumption generated by the system. However, the number of queries, in general, is smaller comparing to the other interactive systems [7, 67].

5.2 Assumption-based learning

5.2.1 Inspiration

Assumption-based reasoning [10, 15, 35, 36] is a technique for solving problems that deal with partial (uncertain, incomplete, incorrect) information. An **assumption** is a logic formula that expresses knowledge which is uncertain but potentially true. E.g. we have a hypothesis P

$$\begin{array}{l}\% \ P: \\ p(X,[X|Y]). \\ p(X,[Y|Z]) :- p(X,Z).\end{array}$$

that should define $last(X,L)$ predicate with intended meaning "X is the last member of the list L". The hypothesis P is incorrect because it covers e.g. `p(a,[a,b])`. Let a new assumption A appear defined as

$$A: \ \forall X : p(X,[X|Y]) \leftrightarrow Y = [].$$

In some cases we could add the assumption to the hypothesis P. However, it is not reasonable to add a new information directly because it may result in inconsistency or inefficiency of the new hypothesis. In our example a new hypothesis P' should be found

$$\begin{array}{l}\% \ P': \\ p(X,[X|Y]) :- Y = []. \\ p(X,[Y|Z]) :- p(X,Z).\end{array}$$

that does not contradict the assumption A.

5.2.2 Inductive inference with assumptions

We are now looking for such an extension of ILP systems which would enable to find an assumption (and consequently a modification of overgeneral program) by inductive inference. Let now a top-down ILP system look for $last/2$ predicate definition and let the example set contain only positive examples

```
p(a,[a]).   p(b,[c,b]).
```

Let a set of possible solutions contain only recursive logic programs. We suppose that the refinement operator is locally complete, i.e. it generates maximal specialisations. In top-down exact learning systems that learn from positive examples it is always the problem of overgenerality of the found clause that arises. Such a system traverses a refinement graph starting from the most general clause in a root of the graph and stops in the first node (clause) that covers a positive example. Thus it will find first the overgeneral program P from the previous page. The goal now is to find such an assumption that enables to induce the correct clauses.

In MIS it is user who has to choose the right negative example that makes the system to find another (more specific) clause. In general we could modify any of the input information – domain knowledge, the example set or constraints of bias. Extension of domain knowledge seems be solvable by a similar way as in [10, 35]. Extension (strengthness) of bias would be useful e.g. if there are more possible solutions for the given input. Here we focus on an extension of an example set. Similarly like in MIS we need some kind of negative example that prevents overgenerality. In general we look for an **integrity constraint**. Integrity constraint IC is an arbitrary formula for which $BK \cup P \nvdash IC$ holds. In terms of ILP it is actually a generalised negative example. It is often uneasy to find the right negative example. As a rule, the right choice strongly depends on the particular inductive engine. It would be welcome if the system itself helped to find it. It of course means that the system must be given an additional knowledge that is needed for such an assistance. We are now looking for minimal knowledge that enables the system to solve that task.

If we know the correct solution P' it is easy for us to find e.g. negative examples `p(c,[c,b])`, `p(e,[f,e,g,h])` for P' that are covered by P. Between these examples, `p(c,[c,b])` is the simplest one that can be used. Moreover this example is actually **near-miss** to `p(b,[c,b])` because it is a negative example and it differs from `p(c,[c,b])` as little as possible (in syntactic sense). Therefore it would be sufficient to have a function f that computes such potential near-misses from the known positive examples. Let the function be just a renaming of a constant. Then we obtain also `p(c,[c,c])` that is a positive instance of $last/2$. In general, a system cannot know – without knowing a model of $last/2$ – whether the generated example is actually negative one. Therefore, we need an acceptability module that verifies that an example computed with f is a negative example.

5.2.3 Generic algorithm

The ideas discussed above are expressed more precisely in the schema of assumption-based learning and mainly in a generic algorithm. In the following sections, WiM system is introduced. Particular parts and functions of

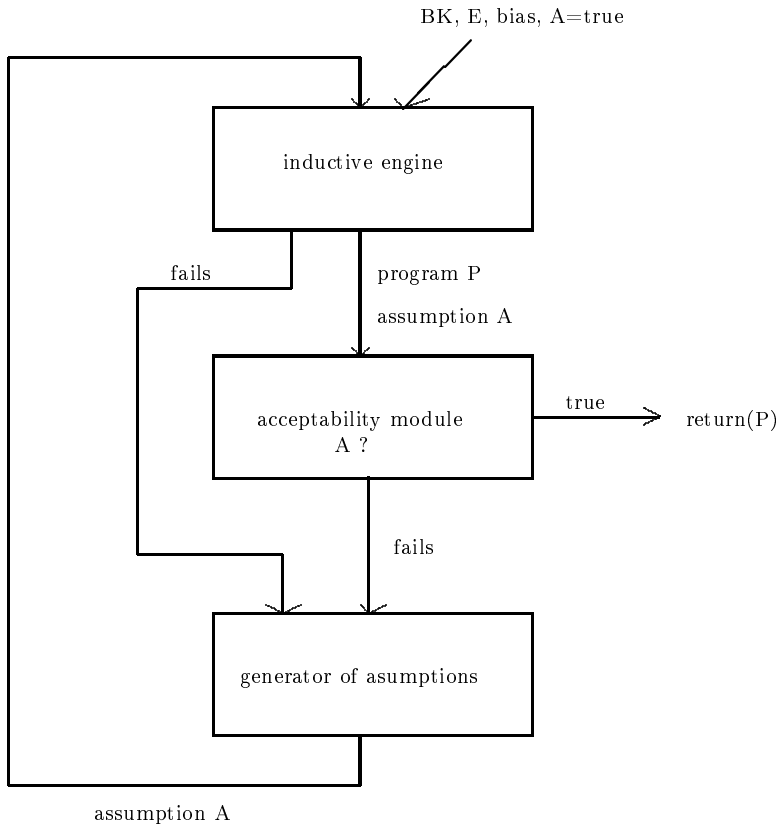

Figure 5.1: Basic schema of assumption-based learning

ABL are also explained there. Basic schema of assumption-based learning is in Fig. 5.1. A generic algorithm of assumption-based learning is in Fig.5.2.

Given:

$BK, E, bias$, $assumptionA = true$
inductive engine I, overgeneral program P
C, F (set of constants and functions that appears in E)
function f, that computes an assumption A
acceptability module AM

1. Call I on $BK \cup P, E \cup A, bias$.
   - **if** I succeeds resulting in program P'
     then call AM to validate the assumption A.
     if A is accepted **then** return(P') **else go to** (2).
   - **else go to** (2).
2. Call f to generate a new assumption A. If it fails, return(fail) and stop else go to (1).

Figure 5.2: A generic algorithm of assumption-based learning

5.3 Basic WiM algorithm

WiM [24, 59, 44] is a program for synthesis of normal logic programs from a small example set. It further elaborates the approach of MIS [67] and $Markus$ [26, 27]. It works in top-down manner and uses shifting of bias and second-order constraints. WiM is an instance of assumption-based ILP. Assumptions are ground negative examples generated one-by-one. In every moment, maximally one assumption is added into an example set. WiM consists of three modules, inductive engine $Markus^+$ (Section 5.4), a generator of assumptions (Section 5.5), and an acceptability module (Section 5.6). The basic WiM algorithm is in Fig. 5.3 In the next sections we describe those three modules in detail.

5.4 Inductive synthesiser $Markus^+$

We implemented $Markus^+$ [24] that is based on $Markus$ system [26, 27]. $Markus^+$ is MIS-like [67] top-down synthesiser applying breath-first search in

Given:

- Specification of the target predicate P : types and modes of its arguments, names and arity of background knowledge predicates.
- Example set E
- Definitions of background knowledge predicates
- 2nd order schema of the target predicate P. P must be an instance of the schema.
- bias: maximal length of clauses, maximal number of free variables in the target predicate, maximal depth of arguments in a clause head, maximal number of clauses

Algorithm:

Call (1). If fails, call (2).
(1)
Init bias.
loop

Call $Markus^+$ to learn predicate P.
if succeeded **then**
Call *acceptability module* to accept P.
if accepted **then return**(P)**, exit**.
else shift bias.
if limit of bias is reached **then return**(false)**, exit** .

pool
(2)
if (1) exited with false {no hypothesis was found within the limits of specified bias}
loop

Generate assumption A.
if no more assumptions **then return**(false)**, exit**.
Add the assumption A to the learning set.
Call (1) with the extended example set $E \cup A$.
if (1) succeeded **then return**(P)**, exit**.
else delete the assumption A from the learning set.

pool

Figure 5.3: WiM algorithm

a refinement graph and controlling a search space with different parameters. $Markus^+$ employs only a subset of $Markus$' parameters. Those parameters concerns only language bias. We wanted to make the work with WiM as easy as possible. More advanced users, of course, can tune also other parameters of $Markus$. However, for most of task is not necessary. New features of $Markus^+$ are described in the next paragraphs. We will there refer to WiM parameters (see Appendix B for full description) that have a form `wim_set(Parameter, ListOfValues)`.

5.4.1 Shifting of bias

$Markus^+$ employs shifting of bias. Four parameters are used for shifting — the maximal number of free variables in a clause, the maximal number of goals in the body of a clause, the maximal head argument depth ($X, [X|Y], [X, Y|Z]$, etc. are of depths 0, 1, 2, respectively), and the maximal number of clauses in a solution. The user defines the minimal and the maximal value of the parameters. $Markus^+$ starts with the minimal values of these parameters. If no acceptable result has been found, a value of one of the parameters is increased by 1 and $Markus^+$ is called again. In such a way all variations are being tried gradually. That choice of parameters implies that $Markus^+$ finds a simpler clause first. E.g. for $member/2$ predicate, positive example `member(a,[a,b])` and the maximal argument depth varying from 1 to 2, `wim_set(mx_arg_depth,[1,2])`, the first found clause is `member(X,[X|Y])` (depth 1) and not `member(X,[X,Z])` (= `member(X, [X|[Z|[]]])`, depth 2).

In the current version of WiM the algorithm that shifts a bias is not too efficient. The inductive engine may generate clauses that were already built for the previous bias settings. But the main goal here was to enable the user to set bias more easily than in $Markus$ and in other ILP systems without narrowing a class of learnable programs. This goal seems be fulfiled.

5.4.2 Multiple predicate learning

In some situations the domain knowledge predicates may be defined extensionally. However, it is not realistic to assume that those definitions are complete enough. It would be appreciated if such extensional definition would be replaced by intensional one. We implemented an algorithm which is sufficient

in most situations for solving this task. All predicates that are to be learned must be declared in a fact

```
wim_set(learn,ListOfPredicates).
```

E.g. for *reverse*/2 it may be `wim_set(learn, [reverse/2, concat/3])`. WiM processes this information together with declaration of these predicates. WiM actually build a dependency tree from predicate definitions where nodes are the names of predicates defined in `wim_set(learn, ListOfPreds)`. Two nodes P_1, P_2 are connected by an edge if P_2 appears in the declaration of predicate P_1 as a needed domain knowledge predicate. After building the dependency tree, the predicates in leaves are learned because they do not need any domain knowledge predicate that is defined extensionally. When the intensional definition of predicate P is learned, it is added into the set of domain knowledge predicates, all leaves are pruned and the whole process continues.

E.g we want to learn *reverse*/2 predicate and predicate *concat*/3 is defined by examples. WiM starts with following information *concat*/3 can be im-

```
wim_set(learn, [reverse/2, concat/3]).

pred_def( reverse/2,[+xl, -xl],[concat/3, reverse/2],[]).
ex( reverse([], []), true).
ex( reverse( [ a, b, c ], [ c, b, a ] ), true ).

pred_def( concat/3, [+xl, +x, -xl], [concat/3], []).
ex( conc([],a,[a]), true).
ex( conc([b],c,[b,c]), true).
ex( conc([b,c],d,[b,c,d]), true).
```

Figure 5.4: Input of WiM

mediately learned because in its list of domain knowledge predicates there is none that appears in list `L` in `wim_set(learn, L)`. After learning the definition and after adding it into background knowledge WiM is called again to learn *reverse*/2.

This method works well in most of practical situation. Some drawback arises

if some of example sets is not good enough and WiM fails to find the intensional definition. In such a situation the extensional definition of the predicate is added into the background knowledge. The method fails only in the rare case if two mutually dependent predicates are to be learned (e.g. $even/2$ that call $odd/2$ and vice versa).

5.4.3 Constraint of a program schema

A second-order schema can be defined which the learned program has to match. This schema definition can significantly increase an efficiency of the learning process because only the synthesised programs which match the schema are verified on the learning set. In the current version of WiM a schema of recursive programs

$$(P \ :- \ Q^*)^+ \ ;$$
$$(P \ :- \ R^*, P^+, S^*.)^+$$

where P, Q, R, S are predicate variables, is built-in. It can be switch on by `wim_set(schema, [recursive])` setting. As default, WiM searches for any logic program that do not has to be necessarily recursive.

5.5 Generator of assumptions

An assumption is generated in the moment when the current example set is not complete enough to ensure that the inductive synthesizer is capable to find a definition of the target predicate. As an assumption, a **near-miss** to a chosen positive example is generated [24]. The whole process of generation of assumptions consists of two steps:

Algorithm of assumption generation:

repeat

1. Find the preferable positive example in the example set E.
2. Generate its near-miss.

until a correct and complete program was found
or no more assumptions exist.

5.5.1 Ordering on positive examples

A preference relation on the set of examples is defined based on measure of complexity for atomic formulas. It enables to generate near-misses of less complex examples first.

We define a **complexity of example** as a sum of complexities of its arguments. A complexity of an arbitrary term is computed as follows.

1. Complexity of an atom is equal to 1.
2. For an unary function term $f/1$, a complexity of term $f(X)$ is computed as a complexity of $X + 1$.
3. Complexity of a n-ary term is equal to a sum of complexities of its arguments +1.

E.g. if learned `last/2` predicate from `{last(a,[a]), last(b,[c,b])}` , a complexity of the first example is 3

complexity of `last(a,[a])` =

1 (complexity of `a`)
+ 2 (complexity of [a|[ ]])
= 3

A complexity of the second one is 1+3=4

A preference relation is induced by this function of complexity. An **example e_1 is prefered to an example** e_2 if the complexity of e_1 is smaller than the complexity of e_2.

The relation of preference is an ordering on a set of examples. Thus it has a minimal elements. Now the preferable example can be computed. First a complexity is computed for every positive example in the learning set. Then arbitrary example with a minimal complexity is chosen as preferable for computing near-misses. In our example, `last(a,[a])` is chosen because it has the minimal complexity.

5.5.2 Generator of near-misses

A syntactic approach is used for computing near-misses. WiM program allows to learn predicates in two domains - lists and integers. For each of those

domains the particular generator of assumptions is implemented exploiting the same preference relation. Whenever a new near-miss has been built, it is added into the example set as the negative example and learning algorithm is called. If no solution is found then the near-miss is replaced by another near-miss of the same positive example. If no near-miss of the example enables to learn a correct definition, next positive example (following the ordering given by the preference relation) is chosen for generation of near-misses.

Domain of lists

A set C of individual constants that appear in the preferable example is built. Then the set is extended by a new constant(of correct type) that does not appear in the particular example. This extended constant set is further used for building near-mises. For a given positive example e and a set of individual constants C, a near-miss is computed by one or more operations below.

1. For a list in e, add an arbitrary constant $c \in C$ at the begining of the list.
2. For a list in e, add an arbitrary constant $c \in C$ at the end of the list.
3. Delete an arbitrary element from a list in e.
4. Replace an individual constant c' appearing in e by another constant $c \in C$.

The set of constants that appears in the preferable example usually contains not more than 1 element. However, it does not allow to generate a rich set of near-misses. E.g. for `last(a,[a])` we would obtain `last(a,[])`, `last(a,[a,a])`, `last(a,[a,a,a])`, ... In our example, the constant `new` has been added so that the set of constants contains two constants, `{new, a}` and the near-misses

```
not last(a,[new,a])
not last(a,[a,new])
not last(a,[])
not last(new,[a])
not last(a,[new])
...
```

are generated one-by-one. The addition of one new constants is enough for learning a rich set of predicates.

Domain of integers

For a given positive example e and an integer argument X, near-misses are computed employing operations below.

1. If $X > 0$, replace X with its predecessor.
2. Replace X with successor $s(X)$.

5.6 Acceptability module

For each assumption which has lead to a new predicate definition, **acceptability module** asks an oracle for a confirmation/rejection of the assumption. As WiM works with ground assumptions, a membership oracle is employed in WiM. The oracle answers $true$ if the ground assumption is in the intended interpretation. Otherwise it answers $false$. Other possibility is to use equivalence oracle [3, 67] where a query is a formula P. The equivalence oracle answers either $true$, if P is complete and consistent with respect to the intended interpretation. Otherwise it answers a counterexample for P.

5.7 Sample session with WiM

A learning session for a predicate p starts from a learning set L, a second-order schema SP of the predicate P, a definition of modes and types of arguments D, and a set of background knowledge predicates BK given by the teacher. We will demonstrate it by learning $last(Elem, List)$ predicate ($Elem$ is the last element of the list $List$) using WiM with input knowledge as follows:

$L = \{p(a,[a]),\ p(b,[c,b])\}$
$SP = P :- Q^*\ .\ (\ P :- R^*, P^*, R^*)^*.$
$D = \{-x, +list(x)\}$
$BK = \{\}$

Q, R, S may be a call of any background knowledge predicates but P, i.e. recursive programs containing at least two clauses are acceptable. The set BK of background knowledge predicates is empty so that only `p/2` predicates may appear in the learned program.

```
?- wim(p/2).

Examples of p(X,Y)
   p(a,[a]), true
   p(c,[b,c]), true
Found clause after searching 1 clauses:
    p(X,[X|Y]):-true.
Found clause after searching 3 clauses:
    p(X,[Y|Z]):-p(X,Z).

Found predicate after searching 4 clauses (Total = 4) :
    p(X,[Y|Z]):-p(X,Z).
    p(U,[U|V]):-true.
with no need of assumption.
OK (yes. / no. / continue. / new example) ?  no.
```

The user rejected the found solution. Therefore WiM looks for an assumption that would result in finding another solution.

```
The preferable example is   p(a,[a]) .
An extended constant set is   [a,new] .

New assumptions are generated ...

After a replacement of an individual constant: p(new,[a])
Examples of p(X,Y)
   p(a,[a]), true
   p(c,[b,c]), true
   p(new,[a]), false
Found clause after searching 1 clauses:
    p(X,[X|Y]):-true.
Found clause after searching 3 clauses:
    p(X,[Y|Z]):-p(X,Z).
```

```
After deleting a list element: p(a,[])
Examples of p(X,Y)
   p(a,[a]), true
   p(c,[b,c]), true
   p(a,[]), false
Found clause after searching 1 clauses:
    p(X,[X|Y]):-true.
Found clause after searching 3 clauses:
    p(X,[Y|Z]):-p(X,Z).

After adding a list element: p(a,[a,a])
Examples of p(X,Y)
   p(a,[a]), true
   p(c,[b,c]), true
   p(a,[a,a]), false
Found clause after searching 2 clauses:
    p(X,[X]):-true.

After adding a list element: p(a,[a,a])
Examples of p(X,Y)
   p(a,[a]), true
   p(c,[b,c]), true
   p(a,[a,a]), false
Found clause after searching 2 clauses:
    p(X,[X]):-true.

After adding a list element: p(a,[new,a])
Examples of p(X,Y)
   p(a,[a]), true
   p(c,[b,c]), true
   p(a,[new,a]), false
Found clause after searching 1 clauses:
    p(X,[X|Y]):-true.

After adding a list element: p(a,[a,new])
Examples of p(X,Y)
   p(a,[a]), true
   p(c,[b,c]), true
```

```
   p(a,[a,new]), false
Found clause after searching 2 clauses:
    p(X,[X]):-true.
Found clause after searching 4 clauses:
    p(X,[Y|Z]):-p(X,Z).

p(a,[a,new]) assumed to be false.
```

The assumption `p(a,[a,new])` has resulted in a new solution. *WiM* now poses a query to the user. After the user's confirmation the new solution is displayed.

```
O.K. ? (yes. /no. / unknown) yes.
Found predicate after searching 4 clauses (Total = 23) :
    p(X,[Y|Z]):-p(X,Z).
    p(U,[U]):-true.
under assumption that
    p(a,[a,new]) = false
```

5.8 Related works

Many research papers and presentations that appeared in the last years less or more influenced our work. First of all, it was the seminal work of Ehud Shapiro [67] (see Section 4.1), followed by works of Pierre Flener [22] and Luc de Raedt [16]. In [22, 23] a strategy for stepwise synthesis of logic programs from specifications by examples and properties is presented. It is based on a methodology described in [18]. The process of synthesis is guided by Divide-and- Conquer schema, features non-incremental presentation of examples. It is proposed to be interactive. For example, as a set of examples is usualy incomplete, some generalization has to be performed. To accept its result user is asked for confirmation. In general, user is asked whenever precisions are needed or a choice is to be made. The system is both inductive and deductive. It starts with inductive reasoning from examples and then deduction is used, whenever appropriate. Luc de Raedt's *CLINT* system [16] was designed to be a user-friendly interactive concept-learner , i.e. to require only information from the user that is easy to formulate/provide and to use as much knowledge as possible. A concept is a definite clause (i.e. Horn clause without negation), and examples both positive and negative are

ground facts. Besides ground examples, the oracle can find an answer to a first-order logic formula. As mentioned before, the ideas about assumption-based learning were inspired by works of Kakas, Mancarela, Kowalski and Toni [35, 36], Concerning bias in ILP, the most of ideas can be found in [6, 11, 56]. We have to mention other work on machine learning and ILP, like [2, 7, 32, 34, 4, 30, 45, 55] that directly or indirectly influenced this work.

Chapter 6

Experimental results

We present experimental results obtained with WiM. We used both carefully chosen learning sets and learning sets generated randomly. The learned definitions of predicates were tested on randomly chosen example sets. WiM outperforms Markus and has higher efficiency of learning as well as smaller dependency on the quality of the example set than other exact learners. Assumption-based framework enables to minimise the number of negative examples.

6.1 Learned predicates

WiM was examined on the following predicates:

List processing predicates

- $member(E, L)$ iff the element E appears in the list L;
- $concat(L1, E, L2)$ iff the list $L2$ is equal to the list $L1$ appended by the element E;
- $append(L1, L2, L3)$ iff the list $L3$ is equal to the list $L1$ appended by the list $L3$;
- $delete(E, L1, L2)$ iff the list $L2$ is the non-empty list $L1$ without its first (existing) occurrence of E;

- $reverse(L1, L2)$ iff the list $L2$ has the same elements as the list $L1$ but in the reverse order. It uses $concat(L1, E, L2)$ predicate which appends the element E to the list $L1$;
- $reverseDL(L1, L2)$ is the same as $reverse(L1, L2)$ but using difference lists;
- $last(E, L)$ iff the element E is the last element of the list L;
- $split(L1, L2, L3)$ iff the lists $L2$ and $L3$ contain only odd and even elements, respectively, of the list $L1$.
- $sublist(L1, L2)$ iff the list $L1$ is a compact subsequence of the list $L2$;
- $insert(X, L1, L2)$ iff X is inserted into sorted list $L1$ resulting in sorted list $L2$;
- $isort(L1, L2)$ is insertion sort.

Peano arithmetics

- $plus(I1, I2, I3)$ iff for integers $I1, I2, I3$ $I3 = I1 + I2$ holds;
- $leq(I1, I2)$ iff for integers $I1, I2$ $I1$ is less or equal I2;

Other predicates

- $length(N, L)$ iff N is the length of the list L;
- $extractNth(N, L, E)$ iff E is the Nth element of the list L.

All integers are assumed to be expressed using a successor function $s(X)$[1] as $s^n(0)$. Training sets as well as predicate declarations can be found in Appendix C.

6.2 Carefully chosen example sets

Carefully chosen examples are usually defined by a user. We suggest a way of generation of this kind of examples that is – for most of predicates from the list above – user-independent. Moreover, it enables to find the smallest example set for which WiM can learn a correct logic program.

For a given predicate, a given depth of a recursive data structure[2] and a universe of constants (`a..z` for lists, `0` for integers) all positive examples were generated. Then a couple of positive examples was chosen randomly and WiM was run on it. If WiM failed on this couple, another couple was tried. If WiM failed on all couples, then triples (quadruples) of positive examples were generated. First tuple in which WiM found the intended definition of a predicate is a set of carefully chosen examples for the given predicate.

For most of predicates no constant might appear in more than one example. It prevents to choose a tuple of positive examples that are on the same resolution path as `member(a,[a])`, `member(a,[b,a])`[3]

There are predicates like $last/2$ that are not learnable from positive examples. WiM always finds an over-general definition ($member/2$ in this case) when learning from positive examples. For those predicates one negative example was added into a training set. The example was constructed as near-miss to some of positive example appearing in the training set.

The method should be further developed. For most of predicates mentioned above it allows to find the learning set. But e.g. $union/3$ is not learnable from such a training set . The method also does not prevent generation a learning set containing `member(a,[a])`, `member(b,[b])` that would be hardly built by a user.

6.3 Randomly chosen example set

6.3.1 Overview

In [2] a method for testing of learners using randomly chosen examples has been introduced. Here we briefly overview the method. More precise description of generation of examples can be found Section 6.3.2. First the domain D of input arguments of a limited depth was generated. Then values of arguments were chosen randomly from a uniform distribution on this domain. As there is a dependency between arguments, output arguments were derived

[1] $s(0) = 1, s(s(X)) = 1 + s(X)$

[2] lists not longer than 4 elements

[3] WiM of course can learn from such training data. The reason to omit it is that learning from examples lying in the same resolution path is in general easier.

(e.g. the first argument of *member*/2, the first argument of *sublist*/2 etc.). Then 2,3 and 5 positive examples were chosen as a learning set, sometimes extended by some negative examples. The learned definitions were tested on 50 positive and 50 negative examples generated randomly. A strategy of testing is described in Section 6.3.3.

6.3.2 Example set generation

Positive examples are generated as follows

1. Generate **input** arguments randomly as terms of depth 0..4 over a domain of constants (`{a,b,c, ... ,z}` for lists, `0` for integers).

2. Compute the value of the **output** argument using the same domain.

3. If the depth of output argument is not greater than 4, and the example does not appear in the example set, add it there.

E.g. for `append/3` predicate with mode declarations `append(+,+,-)`, each example has to match one of the following literals:

```
append([],[],_),  append([],[_],_),  append([],[_,_],_),
    ... , append([],[_,_,_,_]],_),
append([_],[],_), append([_],[_],_), append([_],[_,_],_),
    ... , append([_],[_,_,_,_]],_),
...
append([_,_,_,_],[],_), append([_,_,_,_],[_],_),
    ... , append([_,_,_,_],[_,_,_,_]],_),
```

The input arguments are installed and then the output argument is computed, e.g. for `append([_],[_,_],_)` we may receive ([r],[g,b],[r,g,b]). As the length of [r,g,b] is not greater than 4, the example append([r],[g,b],[r,g,b]) is added into the example set.

Negative examples are chosen in two steps.

1. Generate **all** arguments randomly as terms of depth 0..4 over a domain of constants (`{a,b,c, ... ,z}` for lists, `0` for integers).

2. If it is a negative example, add it to the example set.

6.3.3 Method of testing

We use two criteria for evaluation, a success rate and a fraction of test perfect programs over N learning sessions. For a given logic program and a test set we define **success rate** as a sum of covered positive examples and uncovered negative examples divided by a total number of examples. The quality of ILP systems maybe better characterised by a frequence of test perfect solutions learned by the system. **Test perfect solution** is a logic program which for a test set reaches the success rate 1, i.e. the program covers all positive and no negative examples in a test set,

We generated a learning set as 2,3 and 5 positive examples and 10 negative examples. Then WiM was called to learn a program P. 50 positive and 50 negative examples were generated as a test set and a success rate was computed for P on this testing set. The whole process was repeated 10-times.

6.4 Parameter settings

The system was set to learn only recursive definitions. Shift of bias was enabled. Meanings and thresholds of parameters are in Tab 6.1. For most

Parameter	Meaning	Default
`schema`	schema of a target predicate	recursive
`bias`	enables a shift of bias	shift
`mx_clause_length`	maximal number of subgoals	1
`mx_free_vars`	maximal number of free variables	1
`mx_arg_depth`	maximal depth of terms	4
`mx_clauses`	maximal number of clauses in the target	2
`maxNumOfRefGood`	maximal number of generated clauses	50

Table 6.1: WiM parameters settings

of predicates these default settings were used. For $reverse/2$, $union/3$ the number of subgoals was set to 2. For $union/3$ a maximal number of clauses was increased to 3.

6.5 Overview of experiments

To show that WiM performs well, several questions need to be answered, namely

1. What is a minimal learning set for ILP benchmark predicates ?
2. Does assumption-based learning allow to find missing negative examples?
3. How does WiM performance depend on a given bias?
4. What happens if the number of examples increases?
5. How good is WiM in comparison with its predecessor $Markus$?
6. How good is WiM in comparison with other ILP systems ?

We first evaluate capabilities of WiM itself both on carefully chosen examples (Section 6.6.1) and on randomly chosen examples (Section 6.6.2). We use two different criteria for evaluation of WiM performance on a test set: a success rate of the learned program (Section 6.6.3) and a number of test perfect programs (Section 6.6.4). We also show how WiM performance depends on bias settings (Section 6.6.3) and on changing number of positive examples (Section 6.6.5).

In Section 6.7 attention is paid to a new paradigm of learning with assumptions. Results for carefully chosen positive example are displayed in Section 6.7.1. Then we compare learning without assumptions and with an assumption in Section 6.7.1.

Comparison of WiM with $Markus$, a predecessor of WiM is displayed and discussed in Section 6.8.1. We also compare WiM with other ILP systems $CRUSTACEAN$ and $FILP$ (Section 6.8.2), $SKILit$ (Section 6.8.3) and $Progol$ (Section 6.8.4). We conclude with a discussion of results.

6.6 Learning without assumptions

6.6.1 Carefully chosen examples

Number of carefully chosen examples needed for learning predicates from Section 6.1 are in Tab 6.2. In M/N M(N) means a number of positive (negative)

examples. For most of predicates WiM needs 2 positive examples. WiM never needs more than 4 examples for the class of predicates and more than one negative example. WiM is quite fast. For all predicates CPU time was smaller than 5 seconds on SUN Sparc. The table displays a minimal number

	+/-	CPU time		+/-	CPU time
member	2/0	0.283	last	2/1	0.460
append	2/0	1.950	delete	2/1	0.813
reverse	2/1	1.213	split	2/1	2.313
reverseDL	2/0	2.457	subset	4/0	1.537
plus	2/0	2.090	leq	2/1	7.230
isort	2/1	0.907	extractNth	3/0	0.807

Table 6.2: Results of WiM on carefully chosen examples

of examples needed by WiM for learning a correct predicate definition. Average CPU time from 5 runs is displayed, too.

WiM can learn most of predicates from positive examples. However, there are some predicates from the list in Section 6.1 that WiM cannot learn, for different reasons, in absence of negative examples. E.g. $last/2$ cannot be found from only positive examples because it is actually specialisation of $member/2$. As WiM generates clauses starting from the most general one, it always finds as a first solution the over-general definition of $member$. The similar situation appears for other predicates as $delete/3$, $leq/2$ etc. It must be stressed that in these cases only one negative example is needed.

6.6.2 Evaluation on randomly chosen examples

Following the method described in Section 6.3 we tested WiM performance for the case when examples are not chosen by human. Each learning set contained 2, 3 or 5 positive examples and 10 negative ones. Test sets contained 50 positive and 50 negative randomly generated examples. In Tab. 6.3 we display an average success rate in 10 runs. WiM can learn from 5 examples with an accuracy at least 94 %.

number of positive examples	2	3	5
member	0.80	0.97	0.97
last	0.76	0.89	0.94
append	0.77	0.95	0.95
delete	0.85	0.88	0.97
reverse	0.85	0.95	0.99
extractNth	0.74	0.80	0.98
plus	0.82	0.92	0.96

Table 6.3: Results for randomly chosen examples

6.6.3 Dependence on bias settings

We show in Tab 6.4 how an average success rate changes depending on the maximum argument depth. Learning sets and testing sets was constructed by the same way as in the previous section. Each experiment was usually repeated 10-times. Both the maximum number of free variables and the maximum number of goals was set to 1. For a majority of predicates settings

num.of examples	2				3				5	
max. arg. depth	1	2	3	4	1	2	3	4	1	2
member	0.59	0.80	0.80	0.80	0.90	0.97	0.97	0.97	0.90	0.97
last	0.32	0.59	0.70	0.76	0.40	0.82	0.82	0.89	0.59	0.94
append	0.64	0.77	0.77	0.77	0.89	0.95	0.95	0.95	0.95	0.95
delete	0.61	0.85	0.85	0.85	0.80	0.88	0.88	0.88	0.97	0.97
reverse	0.50	0.85	0.85	0.85	0.95	0.95	0.95	0.95	0.99	0.99
extractNth	0.52	0.68	0.74	0.74	0.80	0.80	0.80	0.80	0.98	0.98
plus	0.75	0.82	0.82	0.82	0.92	0.92	0.92	0.92	0.96	0.96

Table 6.4: Dependence on a maximal argument depth

of maximal argument depth parameter to 2 is enough to reach a success rate greater than 80%. Further increasing of this parameter does not increase WiM accuracy.

6.6.4 Number of test perfect solutions

A number of test perfect solution found in 10 learning sessions is another indicator of quality of ILP systems. Test perfect solution covers all positive

examples and uncovers no negative example in an example set. We again performed 10 runs for 2,3, and 5 positive examples and 10 negative ones randomly chosen. Test sets again contained 50 positive and 50 negative randomly chosen examples. Results are displayed in Tab. 6.5. It is important

	2	3	5
append	2	5	9
delete	1	3	4
reverse	1	4	5
extractNth	2	2	2
plus	3	7	7

Table 6.5: Number of test perfect solutions in 10 learning sessions

that all test perfect solutions were also equal to a correct predicate definition.

6.6.5 CPU time

	2	3	5	10
member	**0.283**	0.307	0.357	0.817
append	**1.950**	2.077	2.500	10.130
delete	-	**0.813**	1.073	2.903
reverse	-	**1.213**	3.420	9.723
last	-	**0.460**	0.600	0.840
plus	**2.090**	4.953	13.203	36.496
split	-	**2.313**	3.657	45.469

Table 6.6: Average CPU time for a different number of examples

In all experiments carefully chosen example sets were taken and extended with randomly chosen examples. If we want to test WiM on learning $member/2$ from 5 examples we must to add 3 examples because the set of carefully chosen examples for $member/2$ contains two positive examples. Those examples was again generated using the same method as in previous experiments. Average times from 5 runs are in Tab 6.6. The results for carefully chosen examples are in the 2nd (for $member/2$, $append/3$ and $plus/3$) and the 3rd columns (for the rest of predicates) and they are printed in bold. The same

bias as before was used for all predicates but *split*. In the case of *split*/3 *WiM* was not able to learn it from 10 examples. That is why a maximal number of clauses was enlarged to 3. The enormous need of CPU time for learning from 10 examples for *plus*/3 and *split*/3 was observed. It happened when *WiM* found 2 clauses that do not cover all positive examples. As *WiM* reached a limit for a maximal number of clauses, it backtracks and is looking for another solution. E.g. for 10 positive randomly chosen examples *plus*/3 *WiM* found the following clauses

```
plus(0,X,X):-true.              (1)
plus(X,Y,s(X)):-true.           (2)
plus(X,Y,s(Y)):-true.           (3)
plus(X,Y,s(s(X))):-true.        (4a)
```

As none of those clauses can be removed as redundant and as the threshold `mx_clause_length=3` was exceeded, the last clause was not added into the solution and another clause

```
plus(s(X),Y,Z):-plus(Y,X,Z).    (4b)
```

was found. For the same reason the clause was rejected, and eventually the clause

```
plus(s(X),Y,s(Z)):-plus(X,Y,Z). (4c)
```

was found. As in the definition containing clauses `{1,2,3,4c}` clauses `2,3` are redundant, they are removed and the target definition is

```
plus(0,X,X):-true.
plus(s(Y),Z,s(U)):-plus(Y,Z,U).
```

6.7 Learning with assumptions

6.7.1 Carefully chosen examples

Tab. 6.7 contains the results of *WiM* if one assumption was generated. The example sets consist of 2 or 3 positive examples carefully chosen by the user and one negative example generated by the system as the assumption (near-miss to a positive example) and afterwards verified by the user. The last column contains average CPU time from 5 runs. Assumption-based learning

	Number of positive examples	CPU time
delete	2	2.717
last	2	3.774
leq	2	32.136
length	3	3.645

Table 6.7: Carefully chosen examples: Learning with assumptions

is more time-consuming for predicates of Peano arithmetics. The reason is the way how near-misses are generated. For a given argument only its predecessor and its succesor is generated.

6.7.2 Randomly chosen examples

# pos.	2			3			5		
	without	with	TP	without	with	TP	without	with	TP
last	0.885	0.896	6	0.906	0.934	7	0.932	0.971	8
delete	0.882	0.962	8	0.857	0.937	7	0.874	0.943	7
leq	0.380	0.703	0	0.527	0.795	4	0.572	0.932	9
length	0.540	0.659	0	0.692	0.816	1	0.728	0.956	4

Table 6.8: Randomly chosen examples: Learning with assumption

We generated randomly N positive examples. Then we compared results reached with WiM in interactive and non-interactive regime. Results are in Tab. 6.8. without(with) means the regime without(with) generation of assumptions. TP means a number of test perfect solutions.

6.8 Comparison with other systems

6.8.1 Comparison with *Markus*

	Markus	*WiM*
member	2/1	2/0
append	2/1	2/0
reverse	2/2	2/1
reverseDL	2/2	2/0
plus	2/1	2/0
insert	3/2	3/2
isort	2/2	2/1

Table 6.9: Comparison with *Markus*

Let us compare *WiM* with its predecessor *Markus* . A summary of results on carefully chosen examples is in Tab 6.2. *WiM* as well as *Markus* needs at least N positive examples for learning a predicate definition which contains N clauses. It means that for all of predicates with 2 clauses *Markus* as well as *WiM* needs not less than 2 positive examples. Concerning a number of negative examples, *WiM* outperforms its predecessor *Markus* significantly. For four of seven predicates in Tab 6.2 *WiM* needs no negative example to learn them. For *isort*/2 one example is needed; *Markus* needs 2 negative examples. For *insert*/3 the number of positive and negative examples are the same for both systems. This small number of negative examples is caused by shifting of bias and a program schema that the target definition must match. *WiM* looks for all solutions inside the strongest bias. If there is no recursive program that is complete and consistent, only then bias is shifted. On the other hand *Markus* finds first a nonrecursive definition. To prevent it *Markus* needs negative examples.

It must be said that there are predicates that *Markus* can learn [25] and *WiM* cannot. The reason is that *WiM* does not allow to set some of parameters, e.g. a number of literals to be added onto a clause body in one refinement step. Moreover, *Markus* employs iterative deepening search but *WiM* breath-first search. This allows to *Markus* to learn, e.g. *multiply*/3, *partition*/4 and *qsort*/2. The reason for a poorer parameter set of *WiM* was

following. We aimed at a system that would be easy to drive, even for user who is not an expert in ILP. However, an extension of the parameter set is a challenge for future research.

6.8.2 *CRUSTACEAN* and *FILP*

Carefully chosen examples

	WiM	*FILP*	*CRUSTACEAN*
member	2/0	4	2/2
last	2/1	?	2/1
append	2/1	4	2/2
delete	2/1	?	2/1
reverse	2/0	4	?
reverseA	3/0	?	2/1
split	2/1	?	2/4
extractNth	3/0	?	2/1
plus	2/0	?	2/3
exponential	no	4	?
factorial	3/0	4	2/1
noneIsZero	2/0	?	2/1
union	4/0	3	no
intersection	no	3	no
subset	4/0	3	?
partition	no	7	no
qsort	no	6	no

Table 6.10: Comparison with *FILP* and *CRUSTACEAN*

Number of examples needed by *WiM* , *FILP* and *CRUSTACEAN* are in Tab. 6.10. *no* means that a system is not able to learn the predicate, ? sings that it is not known.

For all predicates *WiM* needs at worst as many positive examples as *CRUSTACEAN* and less then *FILP* [7] with at most 1 negative example. This example is found by *WiM* itself. *WiM* can learn all the predicates learnable with *CRUSTACEAN* [2]. Moreover, *CRUSTACEAN* generates, as a rule, more than one solution. It also needs more negative examples. *WiM* asks the user whenever a new assumption allows to find a new solution. In our experiments *WiM* stops in that moment. It means that only one membership

query is asked. Even with this limitation to one membership query WiM outperforms $FILP$. We can claim that the number of queries to the user is smaller than in $FILP$.

Comparison with $CRUSTACEAN$ on randomly chosen examples

A comparison of $CRUSTACEAN$ and WiM on randomly chosen examples is in Tab. 6.11. The table gives experiments with or 2, 3, and 5 positive examples, and 10 negative examples, randomly chosen by the way described above, an average of success rates from 10 runs. WiM reaches a higher success rate for all predicates but $noneIsZero$.

	$CRUSTACEAN$		WiM		
	2	3	2	3	5
member	0.65	0.76	0.80	0.97	0.97
last	0.74	0.89	0.76	0.89	0.94
append	0.63	0.74	0.77	0.95	0.95
delete	0.62	0.71	0.85	0.88	0.97
reverse	0.80	0.86	0.85	0.95	0.99
split	0.78	0.86	0.80	0.88	0.79
extractNth	0.60	0.78	0.74	0.80	0.98
plus	0.64	0.86	0.82	0.92	0.96
noneIsZero	0.73	0.79	0.72	0.46	0.58

Table 6.11: Comparison with $CRUSTACEAN$ on randomly chosen examples

6.8.3 $SKILit$

We refer here to experiments described in [34] p.153. $SKILit$ training set contained 2, 3 or 5 positive examples and 10 negative ones. Positive examples were again chosen randomly like in other experiments but negative examples were now generated as near-misses to these positive examples. Success rates obtained with $SKILit$ and with WiM are summarised in Tab 6.12. WiM training set was built by a similar way but negative examples were chosen randomly from the universe of all negative examples (Section 6.3). It implies that training sets for WiM were potentially worse than those for $SKILit$. In spite of that fact, WiM reached higher success rates on testing data than

	SKILit			*WiM*		
	2	3	5	2	3	5
member	0.70	0.89	0.95	0.80	0.97	0.97
last	0.71	0.72	0.94	0.76	0.89	0.94
append	0.76	0.80	0.89	0.77	0.95	0.95
delete	0.75	0.88	1.00	0.85	0.88	0.97
reverse	0.66	0.85	0.87	0.85	0.95	0.99

Table 6.12: Comparison with *SKILit*

SKILit namely for 2 and 3 positive examples. For the case of 5 examples *SKILit* outperforms *WiM* once (*delete*), *WiM* wins twice (*append*, *reverse*), and other results are comparable.

6.8.4 *Progol*

Randomly chosen examples

For a comparison of *WiM* with *Progol* we used again randomly chosen examples. We focus only on *append*/3 predicate because the distribution package of *Progol* contains a training set for this predicate The set contains 17 positive and 8 negative examples. We also display results on this example set obtained with both systems (last columns - distr). Tab 6.13 contains the average success rate for 10 runs. *Progol* never found the correct solution for

Progol					*WiM*				
2	3	5	7	distr	2	3	5	7	distr
0.68	0.81	0.89	0.97	0.89	0.77	0.95	0.95	1.00	1.00

Table 6.13: Comparison with *Progol*

2, 3, 5 and 7 examples neither any recursive solution. On the (distr) data and 5 runs, *Progol* did not stop once and 4-times it found an over-general definition

```
append([A|B],C,[A|D]).
append(A,B,B).
```

with the success rates varying between 0.96 and 0.99.

Progol **distribution example set**

From the distribution set for `append/3` (17 positive and 8 negative examples) 2, 3, and 5 positive examples were chosen randomly and all 8 negative examples was added to it. Results for 10 runs in the form (average success

Progol			*WiM*		
2	3	5	2	3	5
0.546/0	0.571/0	0.672/0	0.778/0	0.85/3	1.00/5

Table 6.14: Results on *Progol* distribution data

rate / number of test perfect solutions) are given in Tab 6.14.

6.9 Summary of results

WiM can learn most of ILP benchmark predicates mentioned in Section 6.1 from 2 positive examples, sometimes extended with one negative example. It never needs more than 4 examples for that class of predicates and more than one negative example. If the negative example is needed it can be generated with very good accuracy by *WiM* itself. We showed that the accuracy of the target definition increases with a number of positive example in the training set as well as with weakening of bias. *WiM* outperforms *Markus* and has higher efficiency of learning as well as smaller dependency on the quality of the example set than other exact learners. *WiM* is quite fast. CPU time needed for learning without assumptions was smaller than 8 seconds on SUN Sparc. Assumption-based learning is of course more time-consuming. The maximal CPU time was smaller than 4 seconds for list-processing predicates and smaller than 33 seconds for Peano arithmetics ($leq/2$). Comparing *WiM* with *MIS* , *MIS* is not able to learn from only positive examples. FOIL [66] needs much more positive examples to succeed. Also the close world assumption employed by FOIL is not appropriate for learning from a small example set.

Chapter 7

WiM for applications

In this chapter we address possibilities of WiM in solving some tasks in database technology. We introduce a concept of basic domain knowledge and then we describe a unified way of extracting reusable domain knowledge from object-oriented data description.

7.1 Application challenges

In the previous chapter we presented experimental results obtained with WiM on both carefully chosen learning sets and learning sets generated randomly. Here we show how to use this kind of ILP techniques to simplify some tasks which have to be usually solved by human. As we focus in this thesis on exact ILP, we looked for such application areas where exact ILP can help exploiting advantages of WiM system.

The most natural area is **automatic program synthesis**. We showed in the previous chapter that WiM can be applied for such tasks. Earlier [23] we argued that the automatic program synthesis is not the only field where exact ILP can be employed and that some tasks in e.g. database technology seem solvable by means of inductive techniques. We will show in the next chapters that the other promising areas are inductive redesign of database schema and some tasks in data mining.

Inductive redesign of database schema. If there are instances of database classes, we can, in most cases, easily design the schema of those classes, namely attributes and their types as well as super/subclasses of the class. In

deductive databases this knowledge can be expressed by means of deductive rules. We will show how to learn those rules employing ILP .These rules can be afterwards used to redesign the database schema.

Data mining. The last rank of applications that we focus here is data mining (DM) [19]. The work on inductive redesign of database schema mentioned above can be considered as DM, too. However, most of DM tasks are inappropriate for exact ILP namely because of noise in data. Fortunately, not all data are necessarily noisy. Geographic data that serve as the background for map drawing is an example of that case. We show how to apply exact ILP in mining of such kind of spatial data.

In the rest of this chapter we first explain a concept of basic domain knowledge and we demonstrate a way of automatic synthesis of basic domain knowledge from object-oriented database schema. Then we describe the whole process more formally.

7.2 Reusable domain knowledge

The use of domain knowledge that is difficult to express in propositional framework is one of main advantages of ILP. That knowledge very often concerns of a structure of the data and can be, as a rule, extracted from database schema or from description of the data without being an expert in the domain. We will call it **basic domain knowledge**. This knowledge is usually extended by knowledge that concerns the task that we solve in the moment. We will call the second kind of knowledge as **expert domain knowledge**. We here focus on the basic knowledge and its (semi)automatic synthesis. This knowledge is independent on a particular task because it depends only on the data itself. When working with a database, a principal part of such knowledge is contained in the database schema.

Domain knowledge for those two tasks which we try to solve here – inductive redesign of database schema and data mining – can be described by some sort of first-order logic – typed, object, or their variants. Here we have chosen **object-oriented paradigm** as a way of data description that is general enough. Object-oriented databases(OODB) [1, 5] enrich, in essence, the relational database model with attributes as complex objects, inheritance and object identity. To prevent dependency on a particular database schema we have chosen object-oriented F-logic [37] as a tool for both class and ob-

ject description. Such kind of object-oriented data description is easy to transform to first-order logic. This transformation from object-oriented description into first-order predicate calculus is always possible. It enables to employ ILP. We show in the next section how to exploit knowledge contained in object-oriented database schema for automatic building of domain knowledge predicates. This **domain knowledge** is then **reusable** in any other learning task for the given database.

7.3 Building domain knowledge: Example

Data description in F-logic

Let us have a simple object-oriented database schema (Fig. 7.1). As an instance of the `CAR` class, we have the object `car1` with the identification number `id = 'BZA-0882'` and with the producer of the class `FACTORY` with the `name = 'Honda'`. The town where the factory is placed is `tname = 'Osaka'` `country = 'Japan'`. The name of the factory director is `pname = 'Uko Jeschita'`. The database schema above is expressed in F-logic as follows.

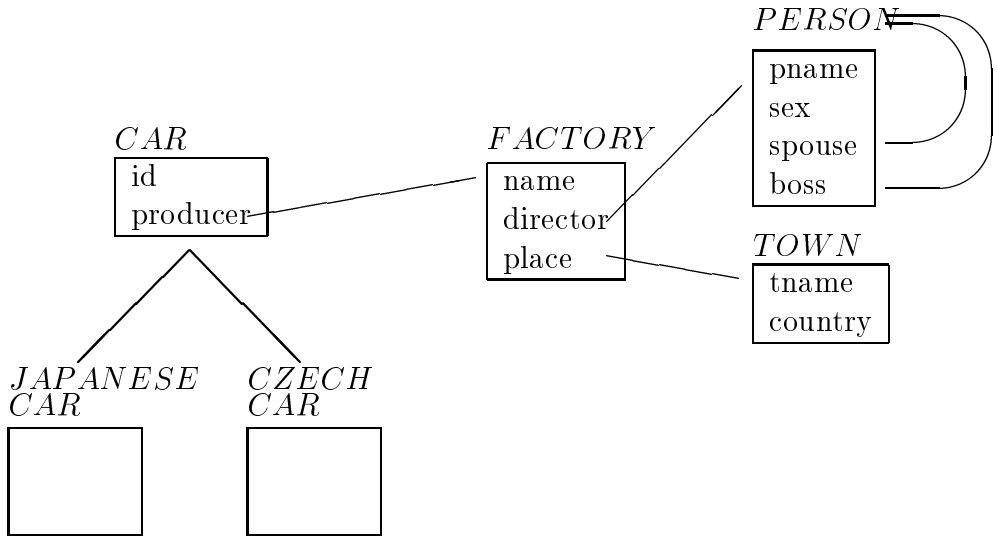

Figure 7.1: Object-oriented database schema

```
car[id=>string;                factory[name=>string;
    producer=>factory].                place=>town;
                                       director=>person].
person[pname=>string ;         town[tname=>string;
    sex=>'Sex'].                     country=>string].
```

Figure 7.2: Description in F-logic

where `A => T` means declaration of attribute `A` of type `T`. The `car1` object definition can be described as F-logic term where `A -> V` means that for a

```
car->car1
    [id->'BZA-0882',
     producer->factory1
                        [name-> 'Honda',
                         place-> place1
                                      [tname->'Osaka',
                                       country->'Japan']
                         director->person1
                                   [pname -> 'Uko Jeschita',
                                    sex -> male] ] ]
```

Figure 7.3: Example of an object description

particular object the attribute `A` has value `V`.

Here we use only a subset of F-logic. In the chosen subset no set-valued attributes are allowed, with the exception of those defined by deductive rules. Extensions to richer subset of F-logic are mentioned in the next chapter.

Building domain knowledge

Domain knowledge consists of predicate definitions and declarations. For each class, the unary predicate of the same name is introduced. We have

```
car(car1). car(car2).
factory(factory1). factory(factory2).
town(place1). town(place2).
```

For each attribute there is an equivalent domain knowledge predicate. In addition, for each value of each attribute a new predicate is built[1] of the form

isAttributeValue(X) :- Attribute(X,Value)

E.g. F-term `country->'Japan'` gives rise to two predicates definitions `country/2`, `isCountryJapan/1`

```
country(place1,'Japan').
isCountryJapan(X) :-country(X,'Japan').
```

A predicate declaration consists of the name and the arity of the predicate, modes of arguments (+x for input, -x for output) and a list of domain knowledge predicates allowed to appear in the learned predicate. For each domain knowledge predicate, the predicate declaration is built. E.g. for `id/2` we will have

```
id/2, [+x,-x], []
```

and for `isCountryJapan/1` we will have

```
isCountryJapan/1, [+x], [country/2]
```

Than the declaration of `japaneseCar/1` predicate looks as follows.

```
japaneseCar/1,
[+x],
[car/1, factory/1, town/1,
 id/2, producer/2, place/2, tname/2, country/2,
 'isIdBZA-0882'/1, ... , isCountryJapan/1,
 ... , isCountryCzechia/1]
```

[1] This is necessary for *WiM*. Otherwise, there is no possibility to introduce constants into clauses. For systems that can introduce constants in the clause body (like *Progol*) it is not necessary

7.4 Unified approach to building domain knowledge

7.4.1 General schema

Algorithm for building reusable domain knowledge and for extraction of learning set from object-oriented database follows. The general schema exploiting WiM system is in Fig. 7.4.

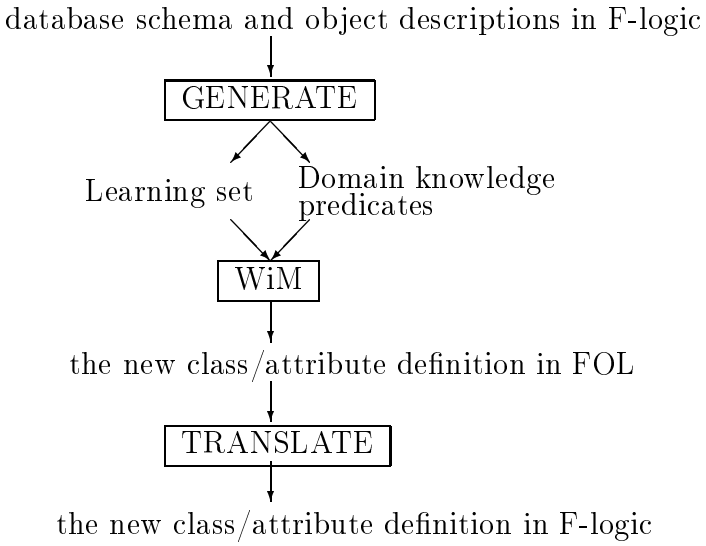

Figure 7.4: $DWiM$ schema

7.4.2 Algorithm $GENERATE$

1. **Select objects from the database.** Build set OL (object-oriented learning set) of objects of a class and assign them truth value *true* (positive examples) or $false$ (negative examples).

2. **Build type definitions.** For each F-term `A => V`, add fact A_V(List), where List contains all values of attribute `A` that appear in OL.

3. **Build domain knowledge.**

(a) Build predicate definitions.

i. For each object $O \in OL$ with object identifier OID, build fact `O(OID)`.

ii. For each attribute A and each value of this attribute that appear in OL, add two clauses `A(OID,V).` `isAV(X):-A(X,V).`

(b) Build predicate declarations.

i. Modes: For each unary predicate, set input mode. For each binary predicate, set mode of the first argument to input and the mode of the second one to output.

ii. Types: Using database schema, set the appropriate type.

4. **Build learning set.**
For each object $O \in OL$ with oid OID, add fact `O(OID,TruthValue)` where `TruthValue` is either `true` of `false`.

It is easy to show that all information included in set OL and relevant information from the particular database schema is contained also in facts and clauses generated by algorithm $GENERATE$. Now WiM can be run on the generated learning set and with the generated domain knowledge.

7.4.3 Translation between first-order logic and F-logic

So far we have not addressed the question about translation between first-order logic(FOL) and F-logic, module $TRANSLATE$ in Fig. 7.4. It is easy to prove that each formula in the defined subset of F-logic can be translated into FOL. The opposite direction, i.e. translation from FOL into F-logic, doesn't have to succeed in general. In our example we have actually learned the rule

```
japaneseCar(X):-producer(X,Y),place(Y,Z),japaneseTown(Z).
```

which cannot be translated into the correct F-logic formula because the information about the parent class `CAR`, i.e. `car(X)` literal, is missing in the rule body. This drawback is solved by adding all information about classes and/or complex attributes so that each variable in the body of rule becomes linked with a logical oid introduced in a class definition. We have got the needed information from the database schema. Let us assume that all class and attribute names are unique. We have to find a class which contains

attribute `producer/2`, and add the appropriate literal to the clause body. After that we have

```
japaneseCar(X):-
      car(X), producer(X,Y), place(Y,Z),japaneseTown(Z)
```

This formula can be afterwards translated into F-logic rule

```
japaneseCar:X  <-
      car:X [producer->F[place->P[country->'Japan']]].
```

In this chapter we described a way of building domain knowledge and a learning set from data described with object-oriented F-logic. In the next chapter we will show how to exploit this approach for building reusable domain knowledge for database schema redesign in deductive object-oriented databases. In Chapter 9 we employ the same method in the process of data mining in geographic data.

Chapter 8

Inductive redesign of a database schema

System DWiM program for deductive object-oriented database schema redesign is introduced based on the method described in the previous chapter. Experimental results obtained with DWiM are discussed.

8.1 Rules in deductive object-oriented databases

In databases there can be two kinds of rules [50]. Active rules define a database system reaction to some actions like update or deletion of a record. Deductive rules define a new dataset. Here we focus on the deductive rules in object-oriented databases. Basically, deductive rules are useful for definition of derived classes and/or for values of derived attributes. We briefly summarize this topic, following [50].

Class definitions by rules

In general, a new class can be created by specializing of an existing class or by generalizing several classes, or by generating new objects. first one is the class `JAPANESE CAR`. The F-logic formula which defines the new class (a subclass of the class `car`) has a form

```
japaneseCar:X <-car:X[producer->F[place->P[country->'Japan']]]
```

where X,F and P are variables. (Here, as in the following text, we will skip type declarations of attributes).

Generalization of classes `CAR FACTORY` and `AIRCRAFT FACTORY` gives rise of the class `FACTORY`

```
factory : X <- carFactory : X
factory : X <- aircraftFactory : X
```

A new class can be created by specialization followed by generalization as demonstrated in the following example. Let us have classes `CAR` and `PUBLIC TRANSPORT VEHICLE`, both with the attribute `power`. Let the attribute has values from the list `(petrol, gasoline, electricity, horse)`. We want to define a new class `E-VEHICLE` of cars and vehicles with electrical power. Then the definition of that class will be

```
eVehicle:X <- car:X[power->electricity]
eVehicle:X <- publicTransportVehicle:X[power->electricity]
```

In all the above examples, the existing objects have been only reclassified (they have not changed their class) and no new objects were generated. However, the derived class can rise from several existing classes resulting in a new class of objects. E.g. you can create the class `family` from pairs of objects of the `PERSON` class.

```
family : F[husband->H, wife ->W] <-  person:H [spouse->W] ,
                                     person:W [spouse->H]
```

Attribute definitions by rules

Rules can be used also for definition of new attributes. E.g. we want to add the new attribute `managed` into the class `PERSON`. The new attribute contains all people who manage the given person:

```
person:X[managed->Y] <- person:X[boss->Y]
person:X[managed->Y] <- person:X[boss->Z],person:Z[managed->Y]
```

8.2 *DWiM*

DWiM system has been implemented following the strategy described in the previous chapter. In the first step, the domain knowledge is being extracted

from an object-oriented database schema. Then positive examples are chosen by user from the database. Negative examples can be generated automatically as assumptions (see Chapter 5), using the closed world assumption, or can be assigned also by user.

Limits of bias are generated automatically, too. The maximum complexity of head is set on 1 as well as number of free variables. Maximum clause length is equal to the number of attribute names and values which have appeared in input objects. Now *WiM* is run with the collected example set and background knowledge.

8.3 Results

```
japaneseCar:X          <-car:X[producer->F[place->
                           P[country->'Japan']

isMother:M             <-person:M [son->S],
                           not person:S[father->X]

factory:X              <-carFactory:X
factory:X              <-aircraftFactory:X

person:X               <-child:X
person:X               <-adult:X

eVehicle:X             <-car:X[power->electricity]
eVehicle:X             <-pubTranVehicle:X[power->electricity]

family:F[hu->H,wi->W]  <-person:H[spouse->W],
                           person:W[spouse->H]

person:X[managed->Y]   <-person:X[boss->Y]
person:X[managed->Y]   <-person:X[boss->Z],
                           person:Z[managed->Y]

person:X[mother->M]    <-not(person:X[father->M]),
                           person:M[son->X]
person:X[mother->M]    <-not(person:X[father->M]),
                           person:M[daughter->X]
```

Figure 8.1: Class and attribute definitions

DWiM was examined on the class and attribute definitions in Fig 8.1. In Tab. 8.1 there are numbers of both positive and negative examples needed for each new class/attribute described above. Negative examples were generated using closed world assumption ('cwa' in the 2nd column) or were found as an assumption with *WiM* itself ('a'). *DWiM* needed from 1 to 5 positive instances(objects) of classes. The number of the needed examples is small enough for user to be able to choose them. *DWiM* program is quite fast so that it can be used in interactive design of deductive object-oriented databases.

	No. of positive examples	No. of negative examples
japaneseCar	1	2(cwa)
isMother	3	9(cwa)
factory	2	1(a)
person	2	1(a)
eVehicle	2	2(cwa)
family	2	4(cwa)
managed	4	6(cwa)
mother	5	7(cwa)

Table 8.1: *DWiM*: Summary of results

8.4 Extension to full F-logic

Our method can be extended to manage full F-logic description of a database schema. In this section we explain how to manage structured types like *set* or *list*, and how to manage complex value attributes, i.e. those referencing another object. We also show how to learn methods.

Structural types

Besides the elementary types like `string`, `integer` , and the reference to another class, there is a structured type `set`. E.g. the definition of the class

car with the attribute passengers and an instance of that class may looks as

```
car[id=>string;
    producer=>factory;.
    passangers=>>person].

car1[id->'BMZ-1234';
    producer->p1;.
    passangers->>{eva, tom, jan}].
```

To work with sets, we have to define the new type for WiM and to add to the background knowledge appropriate predicates, namely member/2 and those for the basic set operations like union/3.

The module $GENERATE$ need to be extended, too. E.g. the above instance car1 is translated into

```
car(car1). producer(car1,p1).
passangers(car1,[eva, tom, jan]).
```

Inheritance

A hierarchy of classes allows to build hierarchical background knowledge starting with the most general class(es). If a system failed to learn with the current background knowledge, then (some of) subclasses are added to the background knowledge. From other point of view, we can see it as the modification of the refinement operator. We add a new refinement operation which adds to the rule body a subclass. We alows to add only subclasses of some class which already appears in the body.

References to complex objects

Another information in the database schema are attributes which reference objects from another class, e.g. the attribute director in the class FACTORY is of type PERSON. It seems be natural to modify the refinement operator by following way. Informally, let us start with the background knowledge containing only predicates transformed from "flat" attributes, e.g. id, color in case of car. Whenever the learning fails with that background knowledge, let us add (some of) predicates transformed from referenced objects.

How to learn methods

As methods in F-logic can be expressed by deductive rules, we may use the same mechanism as described earlier to learn methods. In this task, however, human assistance is also needed. In contrast to learning virtual classes or attributes where the examples has been got from the database itself, we need to know the behavior of the learned method. It is user who has to prepare the example set of input-output behavior of the method.

Example. Let us extend the definition of the `car` class by adding a method `country_of_origin`

```
country_of_origin@ factory => string
```

which may look as

```
X[country_of_origin@ Y -> Z] <-
         X:car, X[producer->Y[place->P[country->Z].
```

8.5 Related works

ILP methods were applied for first-order logic rule synthesis in relational and/or deductive databases close to our approach. ILP system $LINUS$ [43] can learn relations expressed in the language of deductive hierarchical database clauses [48] with the restriction that no new variables may be introduced in the body of a clause. The main disadvantage of $LINUS$, in comparison with $DWiM$ is that $LINUS$ cannot learn recursive rules. $FOIL$ [66] is able to find also recursive definitions of relations. However, as it needs a large learning set, $FOIL$ seems to be more useful for knowledge discovery in databases than for software engineering, like the database schema design is. Both $LINUS$ and $FOIL$ can induce logical definitions of relations from noisy data. It is what $DWiM$ cannot.

The first system for semi-automatic modification of relational database schema exploiting ILP was $INDEX$ [21]. $INDEX$ finds dependent attributes and allows interactive decomposition of relations. Our improvement of that system can be found in [41]. $CLAUDINE$ [14] finds dependency and integrity constraints for a given relational database. Interactive system that provides support for inductive database design is presented in [9].

Chapter 9

KDD in geographic data

We show that the technique described in Chapter 7 and used for database schema design in deductive object-oriented databases is fully usable for spatial mining. An inductive query language is proposed and three kinds of inductive queries are described. Description of GWiM mining system as well as results reached with the system are given.

9.1 Mining in spatial data

Mining in geographic data is challenging and very important. However, the classical KDD, either statistical or based on machine learning in propositional calculus are not convenient for the task. The spatial data have to be managed with means that respect(and exploit) their structural nature. Moreover, non-spatial data need to be used, too, e.g. to find a region with some non-spatial characteristics. Main tasks when mining geographic data [38, 65] are, among others, understanding data, discovering relationship as well as (re)organising geographic databases. We show that inductive logic programming (ILP) is a powerful tool for solving these tasks.

In Chap 7 we introduced the method for building domain knowledge. Exploiting that method we further develop the direction started (or symbolised) with *GeoMiner* [29]. We show that a technique similar to that used for database schema redesign (Chapter 8) is fully usable for spatial mining.

This chapter is organised as follows. In the next section, we demonstrate the

use of $GWiM$ system for solving a simple task. In Section 9.3 $GWiM$ inductive query language for mining in spatial data is described. Results obtained with $GWiM$ are displayed in Section 9.4. We conclude with discussion of results and mainly the weaknesses of the method.

9.2 $GWiM$

The general schema of $GWiM$ system is a modification of $DWiM$ described in Chap. 8. The TRANSLATE module is replaced by a module that compiles a rule in the spatial mining language into input of WiM . Then WiM is called.

We will first demonstrate performance of $GWiM$ on a simple mining task using the database in Fig.9.2. The $BRIDGE$ class consists of all road bridges over rivers. Each bridge has two attributes – `Object1` (a road) and `Object2` (a river). Each river (as well as roads and railways) inherits an attribute $Geometry$ (a sequence of (x,y) coordinates) from the class $LINEAR$. Objects of a class $RIVER$ has no more attributes but `Name` of the river. In a class $ROAD$, the attribute `state` says whether the road is under construction (state=0) or not. The `importance` defines a kind of the road: `1` stands for highways, `2` for other traffic roads, and `3` for the rest(e.g. private ones).

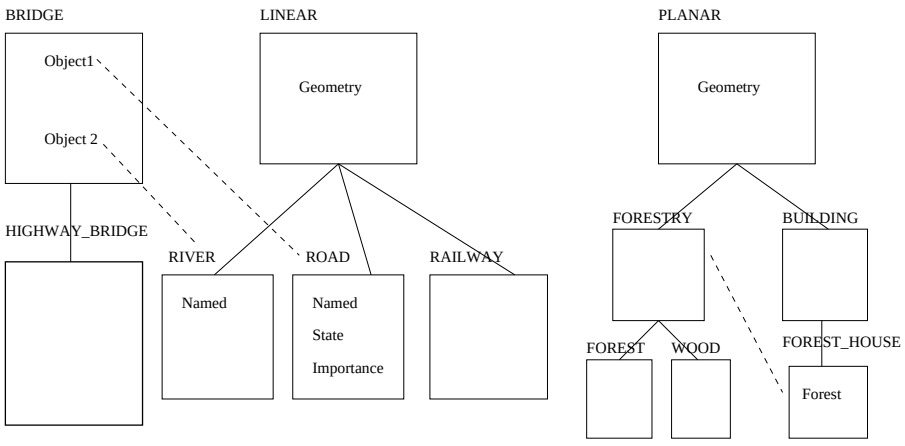

Figure 9.1: Spatial database schema

Our goal is to find a description of the class $HIGHWAY\_BRIDGE$ in terms of other classes in the given database. Let us have 2 rivers with object identifiers `river1,river2`, named `Svratka` and `Svitava` and two roads (with object identifiers `road1, road2`) a highway `'D1'` and a state road `'E7'` that cross those rivers. It can be expressed in first order logic(FOL) as Arguments

```
road(road1) :-
   name(road1,'D1'), geometry(road1,[ ... ]),
   state(road1,1),importance(road1,1).
road(road2) :-
   name(road2,'E7'), geometry(road2, [ ... ]),
   state(road2,1),importance(road2,2),

river(river1) :-
   geometry(river1,[ ... ]),
   name(river1,'Svratka').
river(river2) :-
   geometry(river2, [ ... ]),
   name(river2,'Svitava').

bridge(bridge1) :-
   object1(bridge1,road1),
   object2(bridge1,river1).
bridge(bridg2) :-
   object1(bridge2,road1),
   object2(bridge2,river2)
bridge(bridge3) :-
   object1(bridge3,road2),
   object2(bridge3,river1).
```

Figure 9.2: Schema description in first-order logic

of unary predicates stands for object identifiers(oid's) of corresponding objects. For binary predicates, the first arguments is again oid, the second one is a value of a particular attribute. Let the first two examples be assigned as instances of the class $HIGHWAY\ BRIDGE$. A learning set contains all instances of the class $HIGHWAY\ BRIDGE$ as positive examples. The rest of members of the class $BRIDGE$ (i.e. bridge3) serves as negative examples.

GWiM starts with a minimal language which consists of `object1,object2` attributes of the class *HIGHWAY BRIDGE* itself. The best clause being found

```
highway_bridge(X) :- bridge(X), object1((X,Y), object2(X,Z).
```

is over-general as it covers even the negative example. In that case when no solution is found, the language is extended by adding attributes from neighboring classes(either super/subclasses or referenced classes). If there is a referenced class, the most general superclass is added first. In our case, an attribute `geometry` of the class *LINEAR* (both for *RIVER* and *ROAD*) has been added. As it does not help, the language has been further enriched by attributes of classes *RIVER*, *ROAD*, i.e. `state, importance` and `name`. For this language, *GWiM* has eventually found a logic formula

```
highway_bridge(X) :- bridge(X),object1(X,Y),importance(Y,1).
```

that successfully discriminates between positive and negative examples.

9.3 Inductive query language

In this section we present three kinds of inductive queries. Two of them, that ask for characteristic and discriminate rules, are adaptation of *GeoMiner* [28] rules. The dependency rules add a new quality to the inductive query language. The general syntactic form, adapted from *GeoMiner* of the language is as follows. Semantics of those rule differs from that of *GeoMiner*. Namely

extract < KindOfRule > **rule**
for < NameOfTarget >
[**from** < ListOfClasses >]
[< Constraints >]
[**from point of view** < ExplicitDomainKnowledge >] **.**

Figure 9.3: General form of rules

< ExplicitDomainKnowledge > is a list of predicates and/or hierarchy of predicates. At least one of them have to appear in the answer to the query. The answer to those inductive queries is a first-order logic formula which characterizes the subset of the database which is specified by the rule.

Characteristic rule

Characteristic rules serve for a description of a class which exists in the database or for a description of a subset of a database.

The result of **characteristic** rule is a predicate < NameOfClass > of arity equal to the number of attributes of that class. The predicate is built using other classes and/or attributes in the given database as well as using < ExplicitDomainKnowledge >. Instances of the class introduced in the clause **for** that satisfy **where** condition serve as positive examples.

extract characteristic rule
for < NameOfClass >
where < ConstraintOnListOfClasses >
from point of view < DomainKnowledge > **.**

E.g. the concept of bridge in Section 9.2 can be expressed by this kind of rule.

Discriminate rule

Discriminate rules find a difference between two classes which exist in the database, or between two subsets of the database.

extract discriminate rule
for < NameOfClass >
[**where** < ConstraintOnClass >]
in contrast to < ClassOfCounterExamples >
[**where** < ConstraintOnCounterExamples >]
[**from point of view** < DomainKnowledge >] **.**

Positive examples of the concept < NameOfTarget > are described by

for < NameOfClass >
where < ConstraintOnListOfClasses >

negative examples are described by

from < ListOfClasses >
in contrast to < ClassOfCounterExamples >
where < ConstraintOnCounterExamples >

The discriminate rules allows to find a quantitative description of a class in contrast to another one. E.g. forests have an area greater than 100 hectares. Woods serve as counterexamples there.

Dependency rule

Dependency rules aim at finding dependency between different classes. In opposite to discriminate rules, dependency rules look for a qualitative characterization of a difference between two classes. The clause **from point of view** specifies explicit background knowledge which can be used to build the target predicate < NameOfTarget >. In fact it is a criterion of interestingness.

extract dependency rule
for < NameOfClass >
from < ListOfClasses >
[**where** < ConstraintOnClasses >]
[**from point of view** < DomainKnowledge >] **.**

The objects are defined by the **from ... where ... from point of view ...** formula. The target predicate < NameOfTarget > is of arity equal to a number of classes in < ListOfClasses >. E.g. for forests and woods, an area of a forest is always greater than an area of a wood.

9.4 Results

The geographic data used can be seen in Fig. 9.4. The thick lines are rivers. The data set contains 31 roads, 4 rails, 7 forest/woods, and 59 buildings. Particular mining task are described in the following paragraphs.

Characteristics of *bridge*

Find a description of bridge in terms of attributes of classes *road*, *river*, using the implicit domain knowledge for domains *nom*, *ordinal*, *geometry*, i.e. generic predicates =/2 (for all 3 domains), </2 (for the domain of ordinals, and member/2 for geometry.

extract characteristic rule
for bridge
from road, river.

```
bridge(X,Y):-
     road(X),roadGeometry(X,Z),
     river(Y),riverGeometry(Y,U),
     member(V,Z),member(W,U),W=V.
```

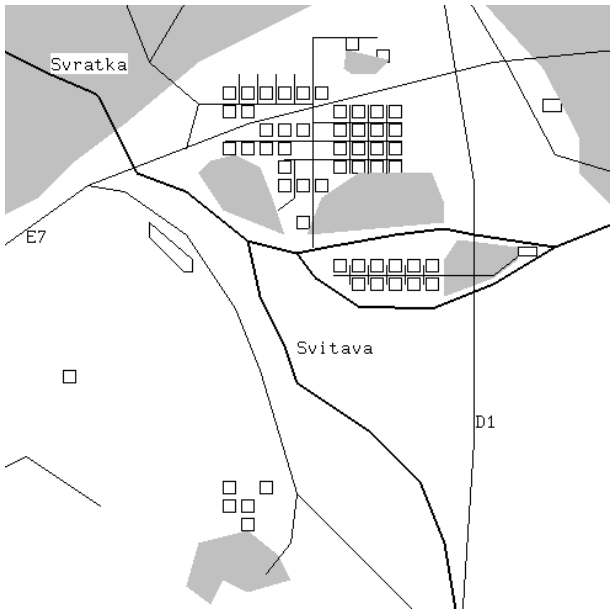

Figure 9.4: Geographic data

Bridge with additional domain knowledge

Find a description of bridge in terms of attributes of classes *road*, *river*, using additional domain knowledge of the predicate $commonPoint(Path1, Path2)$.

extract characteristic rule
for bridge
from road, river
from point of view commonPoint.

```
bridge(X,Y):-
  object1(X,Z),geometry(Z,G1),
  object2(Y,U),geometry(U,G2),
  commonPoint(G1,G2).
```

Discrimination of forests and woods

Find a difference between forests and woods from the point of view of area. *area* is the name of set of predicates like $area(Geometry, Area)$.

extract discriminate rule
for forest
in contrast to wood
from point of view area.

```
forest(F) :-
      geometry(F,GForest),
      area(GForest,Area),
      100 < Area.
```

Relation between forest and wood

Find a relation between forests and woods from the point of view of area. *area* is the name of set of predicates like $area(Geometry, Area)$.

extract dependency rule
for forestOrWood
from forest, wood
from point of view area.

```
forestOrWood(F,W) :-
      geometry(F,GF),area(GF,FA),
      geometry(W,GW), area(GW,WA),
      WA<GA.
```

The number of examples, both positive and negative, needed for a particular concept to be learned are in Tab. 9.4. The cardinality of the learning sets is

	# positive	# negative
bridge	2	1
forest	3	4
forestOrWood	7	7

Table 9.1: $GWiM$: Summary of results

small enough for the user who has to choose the learning examples. $GWiM$ outperforms in some aspects $GeoMiner$. Namely $GWiM$ can mine a richer class of knowledge, Horn clauses. Background knowledge used in $GeoMiner$ may be expressed only in the form of hierarchies. $GWiM$ accepts any background knowledge that is expressible in a subset of object-oriented F-logic [37].

9.5 Discussion

9.5.1 On the inductive query language

The query language is quite powerfull. However, some of queries looks little tricky. Let us look for the relation between forest houses (i.e. those that are near a forest) and other buildings. In the current vesrion of $GWiM$, the inductive query looks as follows.

extract dependency rule
for differentHouses
from forestHouse, forest, building
where building(B, GB),
  not forestHouse(B, F)
from point of view distance, less

```
differentHouses(FH,F,H) :-
      distance(FH,F,D1),
      distance(H,F,D2),
      D1<D2.
```

It seems to be quite difficult (or at least incomfortable) for a user to write such a **where** clause. In this case, it can be improved by allowing

where building not in forestHouse

9.5.2 On mining in real databases

In the current implementation all the data are first imported from a database (PostgreSQL or Oracle) in the form of Prolog facts. One weakness of WiM is its incapability to process large learning sets. Even in the case of the example above (`differentHouses`) some kind of sampling was necessary to reach a result. However, there is a way how to manage such data and we will explain it below.

As mentioned earlier, the general-to-specific learning splits into two steps. In the first step a promising hypothesis is generated that is, in the second step, tested on the example set. The generation step does not depend on the number of examples. If the cardinality of the learning set is increasing, a relative price of the generation step is even decreasing. It seems be straightforward to employ the database management system itself in the testing step. Actually we only need to know the number of examples (both positive and negative) that are covered by the current hypothesis. It means that we need to implement a communication channel connecting an ILP system, with the database system. The answer itself will be received after evaluation of the query using the database management system.

The current version of WiM can work in interactive mode. Oracles have been implemented that allow to ask any external device and even evaluate a given hypothesis on external data. The hypothesis is first translated into a (sequence of) SQL queries. The answers are then evaluated and the hypothesis is either specialized or accepted as the result of the particular inductive query.

We will demonstrate it on the same example as in Section 9.2. We have 2 positive examples and 1 negative example. The minimal success rate for a hypothesis to be accepted is set to 1 – all positive examples need to be covered and none of negative ones does. An initial portion of examples is randomly chosen from the database employing **random oracle**. Let we ask for just 1 positive example. The learned hypothesis

```
highway_bridge(X) :- bridge(X), object1(X,Y), object2(X,Z).
```

has to be verified using the full database. A **success rate oracle** is called that returns the success rate for the hypothesis. Thus the hypothesis is overgeneral and needs to be further specialized. The process continues until the limit for a minimal success rate is reached.

9.6 Related works

An improved version of the query language described in this thesis can be found in [39, 40]. Most of spatial mining algorithms must employ some kind of neighbourhood relationships. In [20] an extension of spatial database management system is proposed for processing of spatial neighbourhood relations that is based on the notion of neighbourhood graphs. The query language described in [39, 40] follows this direction.

GeoMiner [29, 38], the spatial data mining system has been developed in Simon Fraser University, British Columbia, Canada. It follows the line started with the relational data mining system *DBMiner*. The user interface of *GeoMiner* is implemented on the top of MapInfo Professional 4.1 Geographic Information System. Current version of *GeoMiner* can find three kinds of rules, characteristic rules, comparison rules, and association rules. In [8] a fuzzy spatial object query language FuSOQL is introduced for selecting data. The FuSOQL interpreter is built upon the O_2 OODBMS [5] and GeO_2 . A fuzzy decision tree is afterwards built that describes that data.

Chapter 10

Conclusion

10.1 Main contributions

10.1.1 Novel ILP architecture

In Chapter 5 we introduced the new ILP paradigm called *assumption-based learning* motivated by [35, 36]. This novel ILP architecture consists of an *inductive synthesiser*, a *generator of assumptions* which generates extensions of the learning set, and an *acceptability module* which evaluates acceptability of both the found solution and the assumptions. The acceptability module is allowed to ask queries to a teacher. The number of queries is much smaller than in other interactive ILP systems. We experimentally proved that the implementation of the assumption-based paradigm, the system WiM, is less dependent on the quality of the learning set than other ILP systems.

10.1.2 Efficient program synthesis from small learning sets

In Section 3.6.2 we showed how to decrease complexity of the search space in ILP setting. WiM , described in Chapter 5, extends $Markus$ [26] by shifting bias, generating negative examples and employing second-order schema. We showed in Chapter 6 that even with a very small example set (less or equal to 4 positive examples, see Appendix C for particular learning sets) WiM is capable to learn most of the predicates which have been mentioned in ILP literature. We showed that WiM is feasible for solving real-world tasks

(Chapters 8 and 9). Our ssumption-based approach can be combined with any existing ILP system.

10.1.3 Automatic generation of negative examples

When solving tasks with an ILP system, as a rule the needed negative examples are more dependent on the particular system then on the solved problem. In our approach, the system WiM finds negative examples itself (Section 5.5). A near-miss to one of the positive examples is considered as a candidate for that purpose. Such a negative example is found useful if after adding that example to the current learning set, the learner is able to suggest a new definition of the target predicate. Only in such a case the user is asked for a confirmation of that particular candidate for the negative example.

10.1.4 Building of reusable domain knowledge

In Chapter 7 we described a new method for automatic building of domain knowledge. We suppose that logic description of domain in some kind of first-order logic exists or at least can be easily obtained (e.g. by transformation of object-oriented database schema into object-oriented logic). We showed how to exploit such a description for automatic building of a *basic domain knowledge*. We only wanted to make the domain knowledge construction easier and to exploit knowledge which is usually known (in some form) to user. Another advantage of the basic domain knowledge is that it is reusable for other tasks because it depends only on that logic description. Our method can be applied as the first step in domain knowledge building. The next step – completing the basic set of domain predicates with other predicates – is up to user and his experience.

10.1.5 Inductive redesign of object-oriented database

In Chapter 8 we addressed the possibilities of ILP methods in object-oriented database schema modelling, i.e. in database schema design and restructuring. We showed that inductive logic programming could help in synthesis of those rules to support the database schema design and modification. New approach to the object-oriented database modelling by means of ILP was introduced. Experimental results obtained by $DWiM$ system, a variant of WiM , were discussed.

10.1.6 Mining in spatial data

We showed in Chapter 9 that inductive logic programming is a powerful tool for spatial data mining [38]. We proceed in the direction started(or symbolised) by *GeoMiner* [29]. We showed that the technique developed for database schema design in deductive object-oriented databases is fully usable for spatial mining. Mining system $GWiM$ was implemented based on WiM. $GWiM$ overcomes, in expressive power, some other mining methods. Results reached with the system have been reviewed. An inductive query language, with richer semantics, was proposed and three kinds of inductive queries were described. Two of them are improved versions of *GeoMiner* [28] rules. We introduced a new kind of rules, dependency rules, that allow to compare two or more subsets. We are convinced that ILP allows significantly extent an expressive power of inductive query languages in the domain of geographic data.

10.2 Future work

There are at least two research directions that should be followed in future. The first one concerns search strategies of the hypotheses space, the second one aims at (inter)active learning. We will discuss them briefly below.

Markus, the predecessor of WiM, employed iterative deepening search of the hypothesis space. WiM, to be easy to drive, does not support it[1] and sets this parameter to 0. As a consequence, WiM cannot learn e.g. `quicksort/2` predicate. However, it is easy to extend the shift of bias mechanism in WiM by adding the parameter for iterative deepening.

Interactive learning is really a challenge. In many application tasks, like in natural language processing or data mining, there is a huge amount of data. However, those data sets cannot be used directly for learning because of the extreme computational complexity of ILP systems, including WiM. An alternative to sampling techniques may be active learning. ILP system itself looks for examples that are useful to accept/reject a hypothesis.

In the current version of WiM, several oracles have been implemented (see Fig. 10.2). Moreover, a user can easily implement its own oracle. Following oracles have been implemented (Fig. 10.2): `existential` [67], `membership`

[1] It is, of course, possible to use the internal settings of *Markus* even for WiM.

[3, 67], `weak_subset` and `weak_ existential`. The weak subset oracle re-

```
oracle1(weak_subset,Cl,Answer) ... Answer == true iff
              at least 1 new positive example is covered
              and any negative one is not
oracle1(exists,     Cl,Answer) ... Answer
              == set of all instantiations of the clause Cl
oracle1(membership, Cl,Answer) ... Answer == true|false
oracle1(weak_exists,Cl,Answer) ... Answer == true|false
```

Figure 10.1: List of oracles implemented in WiM

turns true iff at least 1 new positive example is covered and any negative one is not. The weak existential oracle returns true if at least one positive example is covered by the tested hypothesis, otherwise it returns false. Using those oracles, a bridge between WiM and Oracle 7 DBMS has been implemented by Petr Chocholáč. This interface enables WiM to extract learning examples directly from user relations saved in Oracle.

Index

Bibliography

[1] Abiteboul S. Hull R. Vianu V.: Foundations of databases. Addison-Wesley Publ. 1995.

[2] Aha D.W., Lapointe S., Ling C.X., and Matwin S.: Inverting implication with small training sets. In Bergadano F., De Raedt L. (Eds.) Proc. of ECML'94, Catania, LNCS 784, pp. 31–48, Springer Verlag 1994.

[3] Angluin D.: Queries and Concept Learning. Machine Learning 2, 4, April 1988, 319-342

[4] Arima J.: Automatic Logic Programming under Highly Redundant Background Knowledge. De Raedt, L.(ed.): Proceedings of the 5th International Workshop on Inductive Logic Programming pp. 355-372, 1995.

[5] Bancilhon F., Delobel C., Kanellakis P.: Building an Object-Oriented Databases Systems: The story of O_2. Morgan Kaufmann 1992.

[6] Bergadano F.: Towards an Inductive Logic Programming Language (manuscript)

[7] Bergadano F. and Gunetti D.: An interactive system to learn functional logic programs. *Proc. of IJCAI'93*, Chambéry, pp. 1044–1049.

[8] Bigolin N.M., Marsala C.: Fuzzy Spatial OQL for Fuzzy Knowledge Discovery in Databases. In Żytkow J.M., Quafafaou M.(eds.): Proc. PKDD'98, Nantes, France. LNCS 1510, Springer Verlag 1998.

[9] H. Blockeel H., De Raedt L.: Inductive Database Design. Proceedings of ISMIS-96.

[10] Bondarenko A., Toni F., and Kowalski R.A.: An assumption-based framework for non-monotonic reasoning. In Perreira L.M., Nerode A. (Eds.) Proc. of the 2nd International Workshop on Logic Programming and Non-Monotonic Reasoning, Lisbon, 1993, pp. 171–189, MIT Press, 1993.

[11] Cohen W.: Rapid prototyping of ILP systems using explicit bias. Proceedings of 1993 IJCAI Workhop on ILP.

[12] Cohen W.: Pac-learning recursive logic programs: Efficient algorithms. Journal of Artificial Intelligence Research, Volume 2, pages 501-539, 1995.

[13] Cohen W.: Pac-learning recursive logic programs: Negative results. Journal of Artificial Intelligence Research, Volume 2, pages 541-573, 1995.

[14] Dehaspe L., Van Laer W., De Raedt L.: Applications of a logical discovery engine. In: Wrobel S.(ed.): Proc. of 4th Workshop on Inductive Logic Programming ILP'94, Bonn Germany, 1994.

[15] DeKleer J.: An Assumption-Based TMS. Artificial Intelligence 18, 1986.

[16] De Raedt, L.: Interactive Theory Revision: An Inductive Logic Programming Approach. Academic Press, 1992. (see also De Raedt, L.:Interactive Concept-Learning. PhD Thesis, Catholic University Leuven, Belgium 1991.)

[17] De Raedt L., Lavrač N., Džeroski S.: Multiple predicate learning. In Proc. IJCAI'93. Morgan Kaufmann, San Mateo, CA.

[18] Deville Y.: Logic Programming: Systematic Program Development. Addison Wesley, 1990.

[19] Džeroski S., Lavrač N.(eds.): Inductive Logic Programming in KDD. A Special Issue of Data Mining & Knowledge Discovery, Vol 3., No. 1, Feb. 1998.

[20] Ester M., Kriegel H.-P., Sander J.: Spatial Data Mining: A Database Approach. In: Proc. of the 5th Int. Symposium on Large Spatial Databases(SSD'97), Berlin. LNCS Vol.1262, pp.47-66, Springer Verlag 1997.

[21] Flach P.: Predicate invention in inductive data engineering. Proceedings of ECML'93, LNAI 667, Springer-Verlag 1993.

[22] Flener P.: Logic Program Synthesis from Incomplete Information. Kluwer Academic Publishers, 1995. (see also Flener P.: Logic Algorithm Synthesis from Examples and Properties. PhD. Thesis, Universitè Catholique de Louvain 1993.)

[23] Flener P., Popelínský L.: On the use of inductive reasoning in program synthesis: Prejudice and prospects. Proc. of the 4th Int'l Workshop on Logic Program Synthesis and Transformation (LOPSTR'94), Pisa, Italy, 1994.

[24] Flener P., Popelínský L. Štěpánková O.: ILP nad Automatic Programming: Towards three approaches. Proc. of 4th Workshop on Inductive Logic Programming (ILP'94), Bad Honeff, Germany, 1994.

[25] Grobelnik M.: MARKUS: An optimized Model Inference System In Proceedings of the ECAI-92 Workshop on Logical Approaches to Machine Learning, Vienna 1992.

[26] Grobelnik M.: Induction of Prolog programs with Markus. In Deville Y.(ed.) Proceedings of LOPSTR'93. Workshops in Computing Series, pages 57-63,Springer-Verlag, 1994.

[27] Grobelnik M.: Declarative Bias in Markus ILP system. Working notes of the ECML'94 Workshop on Declarative Bias, Catania, 1994.

[28] Han J. et al.: DMQL: A Data Mining Query Language for Relational Databases. In: ACM-SIGMOD'96 Workshop on Data Mining

[29] Han J., Koperski K., and Stefanovic N.: GeoMiner: A System Prototype for Spatial Data Mining. In: Proc. 1997 ACM-SIGMOD Int'l Conf. on Management of Data(SIGMOD'97), Tucson, Arizona, May 1997.

[30] Horváth T., Turán G.: Learning logic programs with structured background knowledge. De Raedt, L.(ed.): Proceedings of the 5th International Workshop on Inductive Logic Programming ILP'95, pp.53-76, 1995.

[31] Huntbach, M.: An improved version of Shapiro's Model Inference System. In Shapiro E.(Ed.), Proceedings of Third International Conference On Logic Programming ICLP'86, London, pp.180-187, LNCS 225, Springer-Verlag 1986.

[32] Jorge A., Brazdil P.: Architecture for Iterative Learning of Recursive Definitions. In De Raedt L.(ed.): Advances in Inductive Logic Programming. IOS Press 1996.

[33] Jorge A., Brazdil P.: Integrity Constraints in ILP using a Monte Carlo approach. In Proceedings of 6th Int. Workshop on ILP ILP'96. LNAI 1314 Springer Verlag 1996.

[34] Jorge A.: Iterative Induction of Logic Programs. PhD Thesis, Departamento do Ciências de Computadores Faculdade de Ciências da Universidade do Porto 1998.

[35] Kakas A.C., Kowalski R.A., and Toni F.: Abductive logic programming. Journal of Logic and Computation 2, 6, pp. 719-770, 1992.

[36] Kakas A.C., Mancarela P.: Knowledge assimilation and abduction. Proceedings of ECAI'90 Int. Workshop on Truth Maintenance, Stockholm 1990. Martins(ed.) LNCS Springer-Verlag 1991.

[37] Kifer M., Lausen G., Wu J.: Logical Foundations of Object-Oriented and Frame-Based Languages. TR 93/06, Dept. of Comp. Sci. SUNY at Stony Brook, NY, March 1994 (accepted to Journal of ACM).

[38] Koperski K., Han J., Adhikary J.: Mining Knowledge in Geographical Data. Comm.of ACM 1998

[39] Kuba P.: Knowledge discovery in spatial data. Master thesis, Faculty of Informatics MU Brno, 2000 (in Czech).

[40] Kuba P.: Query language for knowledge discovery in spatial data. Valenta J.(ed.): Proceedings of DATASEM'2000 conference, Brno, 2000 (in Czech).

[41] Kuklová J., Popelínský L.:On Biases in Inductive Data Engineering. ECML'94 Workshop on Declarative Bias, Catania, Sicily, 1994

[42] Lavrač N., Džeroski S.: Background knowledge and declarative bias in inductive concept learning. In: Jantke K.(ed.): Proceedings 3rd International Workshop on Analogical and Inductive Inference, pp.51-71, LNCS 642 Springer Verlag 1992.

[43] Lavrač N., Džeroski S.: Inductive Logic Programming. Techniques and Applications. Ellis Horwood Ltd. 1994

[44] Lavrač N., Džeroski, Kazakov D., Štěpánková O.: ILPNET repositories on WWW: Inductive Logic Programming systems, datasets and bibliography. AI Communications Vol.9, No.4, 1996, pp. 157-206 .

[45] Le Blanc G.: BMWk Revisited. In Bergadano F., De Raedt L. (eds): *Proc. of ECML'94*, Catania, pages 183-197. LNCS 784, Springer Verlag, 1994.

[46] Ling C.X.:. Logic Program Synthesis from Good Examples. Proc of 1st Workshop on ILP, ILP'91 pp. 41–57, Viana do Castelo 1991.

[47] Ling C.X.:. Inductive learning from good examples. In Proc. of IJCAI'91, pp. 751–756, Sydney, Australia. Morgan Kaufmann.

[48] Lloyd J.W.: Foundations of Logic Programming (2nd edition). Springer-Verlag Berlin 1987.

[49] Manandhar S., Džeroski S., Erjavec T.: Learning multilingual morphology with CLOG. In Proc. of ILP'98, 1998.

[50] Manthey R.: Rules and Objects - issues in the dessign and development of DOOD systems. Summer school Object Orientation in Database World, Leysin, Switzerland 1994.

[51] Mitchell, T.M.: Machine Learning. McGraw Hill, Newy York, 1997.

[52] Muggleton S. (ed): Inductive Logic Programming. Volume APIC-38, Academic Press, 1992.

[53] Muggleton S., De Raedt L.: Inductive Logic Programming: Theory And Methods. J. Logic Programming 1994:19,20:629-679.

[54] Muggleton S.: Inverse Entailment and Progol. New Generation Computing Journal, 13:245-286, 1995.

[55] Nédellec C.: Knowledge Refinement Using Knowledge Acquisition and Machine Learning Methods. Gaines B.(ed.): Proceedings of AAAI Spring Symposium, AAAI Press 1992.

[56] Nédellec C., Rouveirol C.: Specification of the HAIKU system. Rapport de Recherche n 928, L.R.I. Université de Paris Sud, 1994.

[57] Nienhuys-Cheng S.-H., de Wolf R.: Foundations of Inductive Logic Programming. Lect. Notes in AI 1228, Springer Verlag Berlin Heidelberg 1997.

[58] Popelínský L.: Towards Synthesis of Nearly Pure Logic Programs. In: Proceedings of LOPSTR'91, Workshops Series Springer Verlag 1992.

[59] Popelínský L.: Towards Program Synthesis From A Small Example Set. Proceedings of 21st Czech-Slovak conference on Computer Science SOFSEM'94, pp.91-96 Czech Society for Comp. Sci. Brno 1993. (See also Proceedings of 10th WLP'94, Zuerich 1994, Switzerland.)

[60] Popelínský L.: Object-oriented data modelling and rules: ILP meets databases. Proceedings of Knowledge Level Modelling Workshop, ECML'95 Heraklion, Crete

[61] Popelínský L., Štěpánková O.: *WiM*: A Study on the Top-Down ILP Program . Technical report FIMU-RS-95-03, August 1995.

[62] Popelínský L.: Knowledge Discovery in Spatial Data by Means of ILP. In: Zytkow J.M., Quafafou M.(Eds.): Proc. of 2nd European Symposium PKDD'98, Nantes France 1998. LNCS 1510, Springer-Verlag 1998.

[63] Popelínský L.: Inductive inference to support object-oriented analysis and design. In: Proc. of 3rd Conf on Knowledge-Based Software Engineering, Smolenice 1998, IOS Press.

[64] Popelínský L.: Induktivní logické programování. Technical Report Gerstner Laboratory GLC-30/99, CTU Prague 1999 (in Czech)

[65] Popelínský L.: Approaches to Spatial Data Mining. In Proceedings of GIS... Ostrava'99 Conference, ISSN 1211-4855, 1999.

[66] Quinlan J.R.: FOIL: A midterm report. Proceedings of ECML'93, LNAI 667, Springer-Verlag 1993.

[67] Shapiro Y.: Algorithmic Program Debugging. MIT Press, 1983.

[68] Srinivasan A., Muggleton S., King, R.D.: Comparing the use of background knowledge by two Inductive Logic Programming systems. De Raedt, L.(ed.): Proceedings of the 5th International Workshop on Inductive Logic Programming, pp. 199-230, 1995.

[69] Stahl I.: Predicate Invention in Inductive Logic Programming. In: L. De Raedt(Ed.), Advances of Inductive Logic Programming, IOS Press, 1996.

[70] Wrobel S.: On the proper definition of minimality in specialization and theory revision. In Brazdil P.B.(Ed.): Proceedings of ECML-93 Conference. LNAI 667, Springer-Verlag 1993, pp. 65-82.

Appendix A

Number of admissible sequences of variables

A sequence of variables $\{X_1, ..., X_i\}$ is **admissible** if no variable X_{j+1} can appear before all variables $\{X_1, ..., X_j\}$ have been used.

In order to count the number of admissible sequences of variables of a given length, it is useful to introduce a function $h(P, N)$. This function specifies the exact number of those admissible sequences of variables of the length N in which just P variables appear. Obviously, this function is defined only for $P \leq N$ (all P variables have to be present in the considered sequence). This function is easy to evaluate for distinguished values of its arguments, namely

$$h(1, N) = 1$$
$$h(P, P) = 1$$

Number of admissible seguences of the length N with just 2 variables is given as a sum of cardinalities of those sets of admissible sequences which differ by the positions of the first occurence of X_2. Variable X_2 can appear first on the position 2, and then on all higher positions, i.e.

$$h(2, N) = 2^{N-2} + 2^{N-3} + ... + 1 = 2^{N-1} - 1$$

For other values of its arguments the function h can be defined recursively as follows

$$h(P, N) = P * h(P, N-1) + h(P-1, N-1).$$

The number $NC(N)$ of all admissible sequences of variables of the length N is then given as a sum

$$NC(N) = h(1,N)+h(2,N)+h(3,N)+....+h(N-1,N)+h(N,N).$$

and the number of all sequences of K variables of the length N is given as a sum

$$NC(K,N) = h(1,N) + h(2,N) + ... + h(K,N).$$

Obviously, for $N > 1$ there holds

$$NC(N) > h(2,N) + 1 = 2^{N-1}$$

The function $NC(N)$ has clearly an exponential character.

Appendix B

Parameters of WiM

General settings

`wim_set(learn,NameOfPredicate/Arity)`

name and arity of a predicate to be learned

`wim_set(verbose,[yes])`

full information about learning session is displayed

`wim_set( maxNumOfRefGood, [Max])`

maximal number of generated clauses that covers at least 1 uncovered positive example

Language bias

`wim_set(mx_free_vars,[Min,Max])`

maximal number of free varibales that may appear in a learned clause

`wim_set(mx_goals,[Min,Max])`

maximal number of goals in a clause body

`wim_set(mx_arg_depth,[Min,Max])`

maximal depth of function terms in a clause head

Search bias

`wim_set(bias,[shift])`

shift of bias allowed

`wim_set(interactive,[interactive=no,answer=continue])`

WiM looks for all solutions in the hypotheses space

Interactive mode

`wim_set(assumptions,[no])`

switchs off an assumption generation

Appendix C

Example sets

Definitions of predicates in the form of

```
pred_def( Predicate/Arity, <arguments types and modes>,
    <background knowledge predicate to use> , []).
```

and examples of the given predicate follows.

```
pred_def( member/2, [ -x, +xl ], [ member/2 ], [] ).
ex( member(a,[a]),true).
ex( member(c,[b,c]),true).

pred_def( concat/3, [+xl, +x, -xl], [concat/3], []).
ex( conc([],a,[a]), true).
ex( conc([b],c,[b,c]), true).
ex( conc([b,c],d,[b,c,d]), true).

pred_def( append/3, [ +xl, +xl, -xl ], [ append/3 ], [] ).
ex( append( [], [ a ], [ a ] ), true ).
ex( append( [  b , c], [  d , e], [ b, c, d, e ] ), true ).
ex( append( [ f ], [ g , h ], [f ,g, h ] ), true ).

pred_def( delete/3, [ +x, +xl, -xl ], [delete/3], [] ).
ex( delete(a,[b,a],[b]),true).
ex( delete(c,[d,e,c,f],[d,e,f]),true).
```

Assumption: delete(b,[b,a],[b]),false

```
pred_def( reverseConcat/2, [ +xl, -xl ],
          [ cconc/3, reverseConcat/2 ], [] ).
ex( reverseConcat([], []), true).
ex( reverseConcat( [ a, b, c ], [ c, b, a ] ), true ).

pred_def( reverseAppend/2, [ +xl, -xl ],
[ ssingleton/2, append/3, reverseAppend/2  ], [] ).
ex( reverseAppend([a,3,4],[4,3,a]), true ).
ex( reverseAppend([2,0],[0,2]),true ).
ex( reverseAppend( [], []), true).

pred_def( last/2, [-x, +xl], [last/2], []).
ex( last(a,[a]),true).
ex( last(b,[a,b]),true).
```

Assumption: last(a,[a,b]),false

```
pred_def(split/3, [+xl, -xl, -xl], [split/3], []).
ex( split([a,b], [a], [b]), true).
ex( split([c,d,e,f], [c,e], [d,f]), true).

pred_def( sublist/2, [ -xl, +xl ], [ sublist/2 ], [] ).
ex( sublist([],[]),true).
ex( sublist( [ c, d ], [ b, c, d, a ] ), true ).
ex( sublist( [ c, d ], [ c, d, b, a ] ), true ).
ex( sublist( [ a ], [ b, a]), true).

pred_def( union/3, [+xl, +xl, -xl], [member/2, union/3], [] ).
ex( union([],[1,2,3],[1,2,3]), true).
ex( union([1,3],[2,3,4],[1,2,3,4]), true).
ex( union([1,2,3,4],[2,3,5],[1,4,2,3,5]), true).
ex( union([1,2,3],[3,4,5],[1,2,3,4,5]), true).

pred_def( plus/3, [+int, +int, -int], [plus/3], []).
ex( plus(0,s(0),s(0)), true ).
ex( plus(s(s(0)),s(s(0)),s(s(s(s(0))))), true ).
ex( plus(s(0),s(s(0)),s(s(s(0)))), true ).
```

Assumption: plus(0,0,s(0)), false

```
pred_def( leq/2, [+int, +int], [leq/2], [] ).
ex( leq(0,s(0)), true ).
ex( leq(s(s(s(0))),s(s(s(s(0))))), true ).
ex( leq(s(s(0)),s(s(s(0)))), true ).
```

Assumption: leq(s(s(s(s(0)))),s(0)) , false

```
pred_def( length/2, [+xl, -int], [length/2, is0/1, ppl/3], []).
ex( length( [], 0 ), true ).
ex( length( [ b, c ], s(s(0)) ), true ).
ex( length( [ f ], s(0) ), true ).
```

Assumption: length([0],0) , false

```
pred_def(extractNth/3, [-int, +xl, -xl], [extractNth/3], [] ).
ex( extractNth( s(0), [ s(s(s(0))) ], []), true).
ex( extractNth( s(0), [ s(s(0)), s(0), (s(s(0))) ],
    [s(0), (s(s(0))) ]), true).
ex( extractNth( s(s(0)), [  s(s(0)), s(0), (s(s(0)))  ],
    [ s(s(0)), (s(s(0)))  ]), true).
```

Assumption: extractNth(s(s(s(0))),[s(s(s(0)))],[]) , false

Appendix D

Geographic data

Specification: Topographic description of rivers, roads, railways as well as woods/forests and buildings

Data complexity estimation or order of magnitude: Based on real-world data (modified to keep confidentiality)

Data format: Prolog

Description: The data contain description of rivers, roads, railways as well as woods/forests and buildings (1 fact per 1 object). Each object is decribed by its geometry in 2-dimensional space and by some characteristics (for roads and railways). The example set is based on real-world topographic data. To keep confidentiality, however, they had to be modified. The modification should keep the main characteristics that can be learned from the data (e.g. based on intersection, being parallel, the area etc.)

The provided dataset is intended for induction of the rules for identification of concepts like bridge, forest in contrast to wood, railway station in contrast to a house nearby the railway etc.