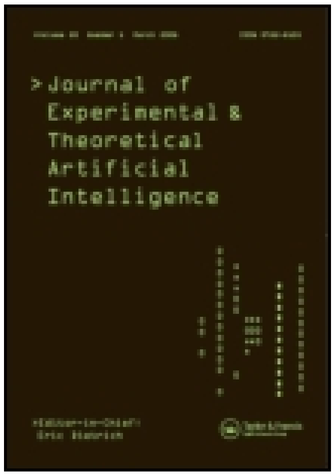

Journal of Experimental & Theoretical Artificial Intelligence
Publication details, including instructions for authors and subscription information:
http://www.tandfonline.com/loi/teta20

Weakening the language bias in LINUS

NADA LAVRAČ [a] & SAŠO DŽEROSKI [a]

[a] Jožef Stefan Institute , Jamova 39, Ljubljana, 61000, Slovenia Phone: tel.: (+386)(61) 1259 199 Fax: tel.: (+386)(61) 1259 199 E-mail:

Published online: 27 Apr 2007.

To cite this article: NADA LAVRAČ & SAŠO DŽEROSKI (1994) Weakening the language bias in LINUS, Journal of Experimental & Theoretical Artificial Intelligence, 6:1, 95-119, DOI: 10.1080/09528139408953783

To link to this article: http://dx.doi.org/10.1080/09528139408953783

J. Expt. Theor. Artif. Intell. 6(1994)95–119.

Weakening the language bias in LINUS

NADA LAVRAČ and SAŠO DŽEROSKI

Jožef Stefan Institute, Jamova 39, 61000 Ljubljana, Slovenia
tel.: (+386)(61) 1259 199, fax: (+386)(61) 219 385
email: {nada.lavrac,saso.dzeroski}@ijs.si

Abstract. The two main limitations of propositional inductive learning algorithms are the limited capability of taking into account available background knowledge and the limited expressiveness of the knowledge representation formalism used for describing examples, background knowledge and concepts. The paper presents a method for using background knowledge effectively in learning both propositional and relational descriptions. The method, implemented in the system LINUS, uses propositional learners in a more expressive logic programming framework. This allows for learning of logic programs in the form of constrained deductive hierarchical database clauses and determinate deductive database clauses.

1. Introduction

Given a set of training examples and background knowledge, the task of inductive learning is to find a hypothesis which explains the examples. In *inductive concept learning*, hypotheses are concept definitions, usually expressed in some logic-based language; examples are descriptions of instances and non-instances of the concept to be learned; and while background knowledge provides additional information about the examples and the domain under study.

For the representation of training examples, background knowledge and induced hypotheses, many inductive learning algorithms use an attribute-value language which has the same expressive power as the language of propositional logic. This language is limited and does not allow for representing complex structured objects and relations among objects or components of objects. Therefore, the background knowledge that can be used in the learning process is of a very restricted form (Nunez 1991). As a consequence, many learning tasks cannot be solved by propositional learning algorithms such as the members of the AQ (Michalski *et al.* 1986) and TDIDT (top down induction of decision trees, Quinlan 1986) families of inductive learning programs.

There are several ways to overcome the problem of a limited description language:

- Methods for *constructive induction* enable a learning system to generate new terms, which do not appear in the descriptions of the examples, and use them in the induced hypothesis (Fu and Buchanan 1985, Muggleton 1987), thus extending the initial vocabulary.
- New terms can be proposed by the expert based on his/her domain knowledge.

This *background knowledge* can take the form of functions of attribute values used to describe the training examples, or relations among attribute values which reflect relations among objects in the problem domain (Michalski 1983).
- A language more expressive than a propositional language can be selected for describing concepts and richer background knowledge used in the learning process. In *inductive logic programming* (ILP, Muggleton 1991, Muggleton 1992), the selected first-order concept description language is the language of *logic programs*.

The paper presents a method which implements the latter two approaches, i.e. which can effectively use background knowledge for learning in a logic programming framework. The presented approach builds an integrated environment for learning both attribute and relational descriptions using propositional inductive learning algorithms. The method is implemented in the system LINUS which was successfully applied to the problem of learning diagnostic rules in rheumatology (Lavrač *et al.* 1991b, 1993), to the problem of learning relational descriptions in several domains known from the machine learning literature (Lavrač *et al.* 1991a) and to the problem of learning rules for finite element mesh design in CAD (Džeroski 1991). Using propositional attribute-value learning algorithms, LINUS allows for recent advances in handling imperfect data in these algorithms to be applied easily to real-life, imperfect data. Experiments in a chess endgame domain with a controlled amount of noise (Džeroski and Lavrač 1991, Lavrač and Džeroski 1992) show that LINUS performs well on imperfect, noisy data of relational nature.

The common logical framework into which LINUS integrates various attribute-value learners is, in fact, the ILP framework. More specifically, in the current implementation, LINUS uses the deductive hierarchical database (DHDB) formalism (Lloyd 1987). LINUS is a descendant of the learning module of QuMAS (qualitative model acquisition system, Mozetič 1987) which was used to learn functions of components of a qualitative model of the heart in the KARDIO expert system for diagnosing cardiac arrhytmias (Bratko *et al.* 1989).

The paper discusses the potential of the LINUS approach in the following dimensions. First, in Section 2, it gives the intuition of how background knowledge can be used in the propositional and in the ILP learning framework. Section 3 discusses the declarative language bias in LINUS which is initially set to constrained deductive hierarchical database clauses, and shows how the approach can be extended to solve more complex ILP problems by weakening the language bias to determinate deductive database clauses. The extended LINUS algorithm is given in Section 4. Finally, in Section 5 we briefly compare the extended LINUS approach to other ILP approaches and conclude.

2. Using background knowledge in learning

This section shows how background knowledge can be effectively used to induce compact hypotheses in a propositional learning setting (Section 2.1), and illustrates how this same approach can be used within the ILP framework (Section 2.2). It also addresses the issue of the complexity of the learning task for the ILP case.

2.1. *Attribute-value learning*

In a propositional concept learning setting, the examples are tuples of attribute values labelled with a concept name (or ⊕ for positive instances of the concept and ⊖ for negative instances), and the induced hypotheses typically have the form of if–then rules or decision trees.

2.1.1. *An example learning problem.* Suppose that the learning task is to find a description of friendly and unfriendly robots (Wnek *et al.* 1990) from a given set of examples. In the original problem, there are 432 examples, described by 6 attributes. In our simplified problem, given in Table 1, robots are described by five attributes: *Is_smiling* ∈ {*no, yes*}, *Holding* ∈ {*sword, balloon, flag*}, *Has_* tie∈ {*no, yes*}, *Head_shape* ∈ {*round, square, octagon*}, and *Body_shape* ∈ {*round, square, octagon*}.

From the examples in Table 1, an algorithm from the AQ family induces the following if–then rules:

$$\begin{aligned} Class = \ & friendly \ \textbf{if} \ [Is\_smiling = yes] \wedge \\ & \qquad [Holding = balloon \vee flag] \\ Class = \ & unfriendly \ \textbf{if} \ [Is\_smiling = no] \\ Class = \ & unfriendly \ \textbf{if} \ [Is\_smiling = yes] \wedge \\ & \qquad [Holding = sword] \end{aligned}$$

Using ASSISTANT (Cestnik *et al.* 1987), a member of the TDIDT family, the decision tree in Figure 1 is induced. When transcribed into if–then rules, the tree produces exactly the same rules as above.

2.1.2. *Using background knowledge in attribute-value learning.* Background knowledge can be expressed in the form of *functions* of attribute values or *relations* among attribute values. In learning attribute descriptions, these functions/relations give rise to new attributes which are considered in the learning process. If the background knowledge is represented in the form of functions, the value of a new attribute is computed as a function of the values of existing attributes, in turn for each training example. The range of values for the function is either a finite set of discrete values or an interval of real numbers. On the other hand, if the background knowledge has the form of relations, the only values the new

Table 1. Examples of friendly and unfriendly robots

Class	*Attributes and values*				
	Is_smiling	*Holding*	*Has_tie*	*Head_shape*	*Body_shape*
friendly	yes	balloon	yes	square	square
friendly	yes	flag	yes	octagon	octagon
unfriendly	yes	sword	yes	round	octagon
unfriendly	yes	sword	no	square	octagon
unfriendly	no	sword	no	octagon	round
unfriendly	no	flag	no	round	octagon

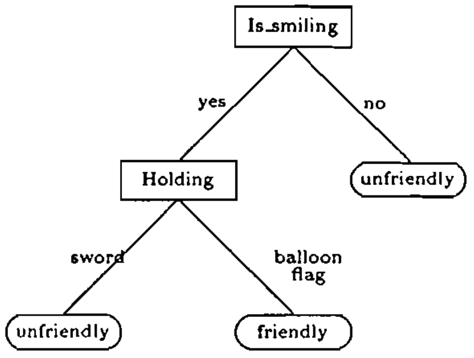

Figure 1. Decision tree for the robot world.

attribute can have are *true* and *false* (if the values of the corresponding attributes of the example do/do not satisfy the relation). In other words, relations are Boolean functions.

For example, background knowledge can check for the equality of attribute values for pairs of attributes of the same type (i.e. attributes with the same set of values). In the world of robots, this would lead to two new attributes that test the equalities *Is_smiling* = *Has_tie* and *Head_shape* = *Body_shape*. For simplicity, let us consider only the new attribute *Head_shape* = *Body_shape* and name it *Same_shape*, the values of which are *true* and *false*. Using this idea, initially introduced by Mozetič (1987), an extended set of tuples is generated and used in learning. The set of attribute-value tuples for the world of robots is given in Table 2.

Since its two values *true* and *false* completely distinguish between the friendly and unfriendly robots, the new attribute *Same_shape* is the only attribute in the decision tree induced by ASSISTANT. The tree is shown in Figure 2. The example shows that new attributes, expressing functions of (relations among) the original attributes that describe the examples, can be more informative than the original attributes.

Table 2. Examples of friendly and unfriendly robots. The last column gives the values of the new attribute *Same_shape* (i.e. *Head_shape* = *Body_shape*)

Class	*Attributes and values*					
	Is_smiling	*Holding*	*Has_tie*	*Head_shape*	*Body_shape*	*Same_shape*
friendly	yes	balloon	yes	square	square	true
friendly	yes	flag	yes	octagon	octagon	true
unfriendly	yes	sword	yes	round	octagon	false
unfriendly	yes	sword	no	square	octagon	false
unfriendly	no	sword	no	octagon	round	false
unfriendly	no	flag	no	round	octagon	false

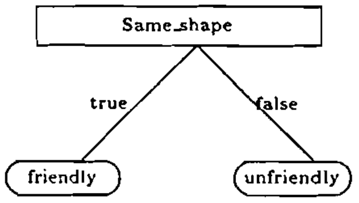

Figure 2. Decision tree built by using background knowledge.

2.2. *Inductive logic programming*

Background knowledge plays a central role in inductive logic programming (ILP) where the task is to define, from given examples, an unknown relation in terms of (itself and) known relations from the background knowledge. In this section, we first introduce some logic programming and deductive database terminology and then define the task of inductive logic programming. We proceed with a description of the LINUS algorithm that transforms the ILP learning task to a propositional learning task and conclude with an analysis of the complexity of the generated propositional task.

2.2.1. *Logic programming and deductive database terminology.* This section briefly introduces the basic logic programming and deductive database terminology, which will be used throughout the rest of the paper. For a comprehensive introduction to logic programming and the programming language Prolog, we refer the reader to Bratko (1990). An introduction to deductive databases can be found in Ullman (1988). A detailed theoretical treatment of logic programming and deductive databases can be found in Lloyd (1987). The following definitions are abridged from Lloyd (1987), and take into account the Prolog syntax.

A *first-order alphabet* consists of variables, predicate symbols and function symbols (which include constants). A *variable* is represented by an upper case letter followed by a string of lower case letters and/or digits. A *function symbol* is a lower case letter followed by a string of lower case letters and/or digits. A *predicate symbol* is a lower case letter followed by a string of lower case letters and/or digits.

A variable is a *term*, and a function symbol immediately followed by a bracketed n-tuple of terms is a term. Thus $f(g(X),h)$ is a term when f, g and h are function symbols and X is a variable. A predicate symbol immediately followed by a bracketed n-tuple of terms is called an *atomic formula*, or *atom*. Both L and its *negation* $\overline{L}$ are *literals* whenever L is an atomic formula. In this case L is called a *positive literal* and $\overline{L}$ is called a *negative literal*.

A *clause* is a formula of the form

$$\forall X_1 \forall X_2 \ldots \forall X_s (L_1 \vee L_2 \vee \ldots L_m)$$

where each L_i is a literal and $X_1, X_2, \ldots, X_s$ are all the variables occurring in $L_1 \vee L_2 \vee \ldots L_m$. A clause can also be represented as a finite set (possibly empty) of literals. Thus the clause $(L_1 \vee L_2 \vee \ldots \overline{L_i} \vee \overline{L_{i+1}} \vee \ldots)$ is equivalently represented as $\{L_1, L_2, \ldots, \overline{L_i}, \overline{L_{i+1}}, \ldots\}$ or most commonly as $L_1, L_2, \ldots \leftarrow L_i$, $L_{i+1}, \ldots$ A set of clauses is called a *clausal theory* and represents the conjunction of its clauses.

Literals, clauses and clausal theories are all *well-formed-formulae* (wff's). Let E be a wff or term. Let *vars*(E) denote the set of variables in E. E is said to be *ground* if and only if *vars*$(E) = \emptyset$.

Horn clause: A Horn clause is a clause which contains at most one positive literal.

Definite program clause: A definite program clause is a clause which contains exactly one positive literal. It has the form

$$T \leftarrow L_1, \ldots, L_m$$

where T, L_1, ..., L_m are atoms.

The positive literal (T) in a definite program clause is called the *head* of the clause while the negative literals (L_1, ..., L_m) are collectively called the *body* of the clause. A Horn clause with no positive literal is a *definite goal.* A *positive unit clause* is a clause of the form $L \leftarrow$, that is, a definite program clause with an empty body. In Prolog terminology, such a clause is called a *fact* and is denoted simply by L.

Definite program: A set of definite program clauses is called a definite logic program.

Only atoms are allowed in the body of definite program clauses. In Prolog, however, literals of the form *not* L, where L is an atom, are allowed, where *not* is interpreted under the *negation-as-failure* rule.

Program clause: A program clause is a clause of the form

$$T \leftarrow L_1, \ldots, L_m \tag{1}$$

where T is an atom, and each of L_1, ..., L_m is of the form L or not L, where L is an atom.

Normal program: A normal program is a finite set of program clauses.

Predicate definition: A predicate definition is a set of program clauses with the same predicate symbol (and arity) in their heads.

Let us now illustrate the above definitions by some simple examples. The clause

$$daughter(X,Y) \leftarrow female(X), mother(Y,X)$$

is a definite program clause, while the clause

$$daughter(X,Y) \leftarrow not\ male(X), father(Y,X)$$

is a normal program clause. Together, the two clauses constitute a predicate definition of the two-place predicate *daughter*, which is also a normal logic program. The first clause is an abbreviated representation of the formula

$$\forall X \forall Y : daughter(X,Y) \vee \overline{female(X)} \vee \overline{mother(Y,X)}$$

and can also be written in set notation as

$$\{daughter(X,Y), \overline{female(X)}, \overline{mother(Y,X)}\}$$

Among the important developments in relational databases are *deductive databases* (Lloyd 1987, Ullman 1988) which allow for both extensional and intensional definitions of relations. The logic programming school in deductive databases (Lloyd 1987) argues that deductive databases can be effectively represented and implemented using logic and logic programming. The definitions below are adapted from Ullman (1988) and Lloyd (1987).

Relation: A n-ary relation is a set of tuples, i.e. a subset of the Cartesian product of n domains $D_1 \times D_2 \times \ldots \times D_n$, where a *domain* (or a *type*) is a set of values. It is assumed that a relation is finite unless stated otherwise.

Relational database: A relational database (RDB) is a set of relations.

Database clause: A database clause is a typed program clause of the form:

$$T \leftarrow L_1, \ldots, L_m \tag{2}$$

where T is an atom and $L_1, \ldots, L_m$ are literals.

Deductive database: A deductive database (DDB) is a set of database clauses.

A relation in a deductive database is essentially the same as a predicate in a logic program. The basic difference between program clauses (1) and database clauses (2) is in the use of types. In typed clauses, a type is associated with each variable appearing in a clause. The type of a variable specifies the range of values which the variable can take. For example, in the relation *lives_in*(X,Y), we may want to specify that X is of type *person* and Y is of type *city*.

It should be noted that types provide a simple syntactic way of specifying semantic information and can increase the deductive efficiency by eliminating useless branches of the search space. In logic, the terms *sorts* and *many-sorted logic* are used, whereas in logic programming the term *type* is used instead of *sort* (Lloyd 1987). We will adopt the term *type*.

Database clauses use variables and function symbols in predicate arguments. Although recursive types and recursive predicate definitions are allowed, the language is substantially more expressive than the language of relational databases.

Deductive hierarchical database: A deductive hierarchical database (DHDB) is a deductive database restricted to nonrecursive predicate definitions and to nonrecursive types.

Nonrecursive types determine finite sets of values which are constants or structured terms with constant arguments.

2.2.2. *Definition of ILP.* One of the early systems that made use of relational background knowledge in the process of learning structural descriptions was

INDUCE (Michalski 1980). Recent inductive learning systems learn descriptions of relations in the form of logic programs and are named inductive logic programming (ILP) systems (Muggleton 1991, Muggleton 1992). Shapiro's work on model inference (Shapiro 1983), Plotkin's work on inductive generalization (Plotkin 1969) and the work of Sammut and Banerji (1986) have inspired most of the current work in the field of ILP.

Systems that learn inductively from examples (including ILP systems) can be divided along several dimensions. First of all, they can learn either a single concept or multiple concepts (predicates). They may require all the training examples to be given before the learning process (batch learners) or may accept examples one by one (incremental learners). During the learning process, a learner may rely on an oracle to verify the validity of generalizations and/or classify examples generated by the learner. The learner is called interactive in this case and non-interactive otherwise. Finally, a learner may try to learn a concept from scratch or can accept an initial hypothesis (theory) which is then revised in the learning process. The latter systems are called theory revisors.

Although these dimensions are in principle independent, existing ILP systems are situated at two ends of the spectrum. At one end are batch non-interactive systems that learn single predicates from scratch, while at the other are interactive and incremental theory revisors that learn multiple predicates. Following De Raedt (1992) we call the first type of ILP systems *empirical ILP* systems and the second type *interactive ILP* systems.

Examples of empirical ILP systems are GOLEM (Muggleton and Feng 1990), FOIL (Quinlan 1990), LINUS (Lavrač *et al.* 1991a) and MOBAL (Wrobel 1988, Kietz and Wrobel 1992). However, MOBAL learns multiple predicate definitions. Interactive ILP systems include MIS (Shapiro 1983), CLINT (De Raedt 1992) and CIGOL (Muggleton and Buntine 1988).

The task of empirical inductive logic programming can be now formulated as follows:

Given:

- a set of training examples $\mathscr{E}$, consisting of true $\mathscr{E}^+$ and false $\mathscr{E}^-$ ground facts of an unknown predicate p,
- a description language $\mathscr{L}$, specifying syntactic restrictions on the definition of predicate p,
- background knowledge $\mathscr{B}$, defining predicates q_i (other than p) which may be used in the definition of p and which provide additional information about the arguments of the examples of predicate p

Find:

- a definition $\mathscr{H}$ for p, expressed in $\mathscr{L}$, such that $\mathscr{H}$ is complete, i.e., $\forall e \in \mathscr{E}^+$: $\mathscr{B} \cup \mathscr{H} \models e$, and consistent with respect to the examples and background knowledge, i.e. $\forall e \in \mathscr{E}^-$: $\mathscr{B} \cup \mathscr{H} \not\models e$.

In the following, we refer to the true facts $\mathscr{E}^+$ as *positive examples*, the false facts $\mathscr{E}^-$ as *negative examples* and the definition of p as the definition of the *target* relation. When learning from noisy examples, the completeness and consistency criteria need to be relaxed in order to avoid overly specific hypotheses (Lavrač and Džeroski 1992).

2.2.3. *Transforming ILP problems to propositional form.* The method of using background knowledge, outlined in Section 2.1.2, is based on the idea that the use of background knowledge can introduce new attributes for learning. This same method can be used within the ILP framework.

In ILP, various restrictions on the complexity of the hypothesis language $\mathcal{L}$ and various kinds of information about the background knowledge predicates can be used to further constrain the hypothesis space, which is initially determined by the predicates from background knowledge $\mathcal{B}$. For example, some of the current empirical ILP systems constrain $\mathcal{L}$ to function-free program clauses, as is the case in FOIL, or determinate clauses as in GOLEM. LINUS is limited to constrained DHDB clauses. MOBAL uses rule models (also called program schemata) which define the form of induced predicate definitions.

To illustrate our method, let us currently assume the following restrictions on the hypothesis language $\mathcal{L}$:

Datalog clauses: A hypothesis is a set of Datalog clauses (Ullman 1988), i.e. program clauses with no function symbols in the arguments. Datalog allows only variables and constants as predicate arguments.

Typed clauses: Clauses are typed, i.e. each variable appearing in arguments of literals is associated with a set of values.

Constrained clauses: Clauses are constrained, i.e. all variables in the body also appear in the head.

Nonrecursive clauses: Clauses are nonrecursive, i.e. the predicate symbol in the head does not appear in any of the literals in the body.

Let us further assume that background knowledge $\mathcal{B}$ is represented by typed Datalog clauses (with no further restrictions) and training examples have the form of ground facts.

In this setting, the algorithm which solves ILP problems by transforming them into propositional form consists of the following three steps:

- The learning problem is transformed from relational to attribute-value form.
- The transformed learning problem is solved by an attribute-value learner.
- The induced hypothesis is transformed back into relational form.

The outlined algorithm allows for a variety of approaches developed for propositional problems, including noise-handling techniques in attribute-value algorithms such as ASSISTANT (Cestnik *et al.* 1987) or CN2 (Clark and Niblett 1989), to be used for learning relations.

The algorithm is illustrated on a simple ILP problem of learning family relationships. The task is to define the target relation *daughter*(X,Y), which states that person X is a daughter of person Y, in terms of the background knowledge relations *female, male* and *parent*. These relations are given in Table 3,

Table 3. A simple ILP problem: learning the *daughter* relationship

Training examples		*Background knowledge*		
daughter(sue,eve)	⊕	parent(eve,sue)	female(ann)	male(pat)
daughter(ann,pat)	⊕	parent(ann,tom)	female(sue)	male(tom)
daughter(tom,ann)	⊖	parent(pat,ann)	female(eve)	
daughter(eve,ann)	⊖	parent(tom,sue)		

where all variables are of type *person*. The type *person* is defined as *person* = {*ann, eve, pat, sue, tom*}. There are two positive and two negative examples of the target relation.

The first step of the algorithm, i.e. the transformation of the ILP problem into attribute-value form, is performed as follows. The possible applications of the background predicates on the arguments of the target relation are determined, taking into account argument types. Each such application introduces a new attribute. In our example, all variables are of the same type *person*. The corresponding attribute-value learning problem is given in Table 4, where *f* stands for *female*, *m* for *male* and *p* for *parent*. The attribute-value tuples are generalizations (relative to the given background knowledge) of the individual facts about the target relation.

In Table 4, *variables* stand for the arguments of the target relation, and *propositional features* denote the newly constructed attributes of the propositional learning task. When learning function-free Datalog clauses, only the new attributes are considered for learning. If we remove the function-free restriction, the arguments of the target relation are used as attributes in the propositional task as well (see Sections 3.1.2 and 3.1.3 for a more detailed explanation).

In the second step, an attribute-value learning program induces the following if–then rule from the tuples in Table 4:

$$Class = \oplus \textbf{ if } [female(X) = true] \wedge [parent(Y,X) = true]$$

In the last step, the induced if-then rules are transformed into Datalog clauses. In our example, we get the following clause:

$$daughter(X,Y) \leftarrow female(X), parent(Y,X)$$

Note that the same result can be obtained on a similar learning problem,

Table 4. Propositional form of the *daughter* relationship problem

Class	*Variables*		*Propositional features*							
	X	*Y*	*f*(*X*)	*f*(*Y*)	*m*(*X*)	*m*(*Y*)	*p*(*X*,*X*)	*p*(*X*,*Y*)	*p*(*Y*,*X*)	*p*(*Y*,*Y*)
⊕	sue	eve	true	true	false	false	false	false	true	false
⊕	ann	pat	true	false	false	true	false	false	true	false
⊖	tom	ann	false	true	true	false	false	false	true	false
⊖	eve	ann	true	true	false	false	false	false	false	false

where the target relation *daughter*(X, Y) is to be defined in terms of the relations *female*, *male* and *parent*, given the background knowledge from Table 5. It illustrates that non-ground background knowledge containing rules in addition to facts can be used within our approach.

2.2.4. *The complexity of learning constrained logic programs.* To show the complexity of the ILP learning task, obtained by using background knowledge in the learning process, let us consider the number of attributes in the transformed learning task (Lavrač *et al.* 1991a). For simplicity, only background knowledge predicates q_i are considered in this analysis; a more detailed analysis can be found in (Džeroski and Lavrač 1992).

The total number of attributes k_{Attr} to be considered by a propositional learner equals:

$$k_{Attr} = k_{Arg} + \sum_{i=1}^{l} k_{New,q_i}$$

where $k_{Arg} = n$ is the number of arguments of the target relation p (i.e. *variables* in Table 4), and k_{New,q_i} is the number of new attributes resulting from the possible applications of the background predicate q_i on the arguments of the target relation.

Under the function-free restriction, k_{Attr} is equal to the total number of new attributes resulting only from the possible applications of the l background predicates on the arguments of the target relation (i.e. *propositional features* in Table 4):

$$k_{Attr} = \sum_{i=1}^{l} k_{New,q_i} \tag{4}$$

Suppose that u is the number of distinct types of arguments of the target predicate p, u_i is the number of distinct types of arguments of the background predicate q_i, $n_{i,s}$ is the number of arguments of q_i that are of type $\mathcal{T}_s$ and k_{ArgT_s} is the number of arguments of target predicate p that are type $\mathcal{T}_s$. Then k_{New,q_i} is computed by the following formula:

$$k_{New,q_i} = \prod_{s=1}^{u_i} (k_{ArgT_s})^{n_{i,s}} \tag{5}$$

The $n_{i,s}$ places for arguments of type $\mathcal{T}_s$ can be, namely, filled in $(k_{ArgT_s})^{n_{i,s}}$ ways independently from choosing the arguments of q_i which are of different types.

Since a relation where two arguments are identical can be represented by a

Table 5. Intensional background knowledge for learning the *daughter* relationship

Background knowledge			
mother(eve,sue)	parent(X,Y) ←	female(ann)	male(pat)
mother(ann,tom)	mother(X,Y)	female(sue)	male(tom)
father(pat,ann)	parent(X,Y) ←	female(eve)	
father(tom,sue)	father(X,Y)		

relation of a smaller arity, the arguments of a background predicate can be restricted to be distinct. In this case the following formula holds:

$$k_{New,q_i} = \prod_{s=1}^{u_i} \binom{k_{ArgT_s}}{n_{i,s}} \times n_{i,s}! \qquad (6)$$

To illustrate the above formulae on the simple example from Tables 3 and 4, observe that q_1 = *female,* q_2 = *male* and q_3 = *parent.* As all arguments are of the same type $\mathcal{T}_1$ (*person*), $u_1 = u_2 = u_3 = 1$ and $k_{ArgT_1} = 2$. Since there is only one type, n_i can be used instead of $n_{i,s}$. In this notation, $n_1 = n_2 = 1$ and $n_3 = 2$. Thus, according to equation (5), $k_{New,q_1} = k_{New,q_2} = (k_{ArgT_1})^{n_1} = (k_{ArgT_1})^{n_2} = 2^1 = 2$. This means that there are two applications of each of the predicates *female* and *male*, namely *female*(X), *female*(Y) and *male*(X), *male*(Y), respectively. Similarly, $k_{New,q_3} = (k_{ArgT_1})^{n_3} = 2^2 = 4$, the four applications of *parent*/2 being *parent*(X,X), *parent*(X,Y), *parent*(Y,X) and *parent*(Y,Y). Finally, $k_{Attr} = k_{New,q_1} + k_{New,q_2} + k_{New,q_3} = 2 + 2 + 4 = 8$.

If the built-in background predicate *equality* (=/2) is used, the number of its possible applications equals to:

$$k_{New_{1=}} = \sum_{s=1}^{u} \binom{k_{ArgT_s}}{2} = \sum_{s=1}^{u} \frac{k_{ArgT_s} \times (k_{ArgT_s} - 1)}{2} \quad .$$

where k_{ArgT_s} denotes the number of arguments of the same type $\mathcal{T}_s$, and u is the number of distinct types of arguments of the target relation; from the arguments of type $\mathcal{T}_s$, $(k_{ArgT_s} \times (k_{ArgT_2} - 1)/2$ different attributes of the form $X = Y$ can be constructed.

Assuming a constant upper bound j on the arity of background knowledge predicates, the largest number of attributes that can be obtained in case that all variables are of the same type is of the order $O(ln^j)$, where l is the number of predicates in the background knowledge and n is the arity of the target predicate. In other words, the size of the transformed learning task is polynomial in the arity n of the target predicate p and the number l of background knowledge predicates. This allows us to use PAC-learnability results for the propositional case in the ILP framework (Džeroski *et al.* 1992a).

3. Declarative language bias in LINUS

Any mechanism employed by a learning system to constrain the search for concept descriptions is named *bias* (Utgoff and Mitchell 1982). *Declarative bias* denotes explicit, user-specified bias which can preferably be formulated as a modifiable parameter of the system. Bias can either determine how the hypothesis space is searched (*search bias*) or determine the hypothesis space itself (*language bias*). This paper is only concerned with the language bias. By selecting a stronger language bias (a less expressive hypothesis language) the search space becomes smaller, and learning more efficient; however, this may prevent the system from finding a solution which is not contained in the less expressive language. This expressiveness/tractability tradeoff underlies much of the current research in inductive learning.

The term *expressiveness* captures two different dimensions. The first dimension is the expressive power of a formalism; in its standard meaning, *stronger expressive power* means that there are concepts that can be represented in the stronger formalism which cannot be represented in the weaker formalism. The other dimension is the *length* of the concept representation. For example, it is possible to state that two Boolean attributes have the same value in both an attribute-value language and a first-order language. But, whereas in the former we have to say $(A = 0 \wedge B = 0) \vee (A = 1 \wedge B = 1)$, we can simply say $A = B$ in the latter.

Various forms of language bias have been employed by existing ILP learners, including types and symmetry of predicates in pairs of arguments (Lavrač *et al.* 1991a), input/output modes and *ij*-determination (Muggleton and Feng 1990), program schemata (Wrobel 1988), predicate sets (Bergadano *et al.* 1989), parametrized languages (De Raedt and Bruynooghe 1990), integrity constraints (De Raedt and Bruynooghe 1992) and determinations (Russell 1989).

In this section we describe the syntactic language bias which is currently used in LINUS. We then present a weaker bias of determinate deductive database clauses that can be used within the LINUS framework, extending LINUS to solve more complex ILP problems that require a more expressive hypothesis language.

3.1. *Language bias in LINUS*

The learning method which transforms an ILP problem to a propositional learning problem (Section 2.2.3) is implemented in the system LINUS (Lavrač *et al.* 1991a). Before giving the restrictions imposed by the hypothesis language $\mathcal{L}$ of deductive hierarchical database clauses in LINUS, it is important to consider the form of training examples $\mathcal{E}$ and, in particular, predicates in $\mathcal{B}$ which actually determine the hypothesis space. Training examples have the form of ground facts (which may contain structured, but nonrecursive terms) and background knowledge has the form of deductive database clauses (possibly recursive). Variables are typed.

3.1.1. *Background knowledge.*

In LINUS, predicate definitions in the background knowledge $\mathcal{B}$ are of two types:

Utility functions: Utility functions f_j/n_j are annotated predicates; mode declarations specify the *input* and *output* arguments, similar to mode declarations in GOLEM and FOIL2.0 (Quinlan 1991). When applied to ground input arguments from the training examples, utility functions compute the unique ground values of their output arguments. When used in an induced clause, output variables must be bound to constants.

Utility predicates: Utility predicates q_i/n_i have only *input* arguments and can be regarded as Boolean utility functions having values *true* or *false* only.

A reduction of the hypothesis space is achieved by taking into account the pre-specified types of predicate arguments and by exploiting the fact that some utility predicates are symmetric:

Symmetric predicates: Utility predicates can be declared *symmetric* in a set of arguments of the same type. For example, a binary predicate $q_i(X,Y)$ is symmetric in $\{X,Y\}$ if X and Y are of the same type, and $q_i(X,Y) = q_i(Y,X)$ for every value of X and Y. A built-in symmetric utility predicate *equality* (=/2) is defined by default on arguments of the same type.

Note that the complexity analysis in Section 2.2.4 (equation (5)) does not take into account symmetry and the use of utility functions f_j/n_j.

3.1.2. *Hypothesis language.* In the current implementation of LINUS, the selected hypothesis language $\mathcal{L}$ is restricted to constrained deductive hierarchical database (DHDB) clauses. In DHDB, variables are typed and recursive predicate definitions are not allowed. In addition, all variables that appear in the body of a clause have to appear in the head as well, i.e. only constrained clauses are induced.

To be specific, the body of an induced clause in LINUS is a conjunction of literals, each having one of the following four forms:

(1) a binding of a variable to a value, e.g. $X = a$;
(2) an equality of pairs of variables, e.g. $X = Y$;
(3) an atom with a predicate symbol (utility predicate) and input arguments which are variables occurring in the head of the clause, e.g. $q_i(X,Y)$; and
(4) an atom with a predicate symbol (utility function) having as input arguments variables which occur in the head of the clause, and output arguments with an instantiated (computed) variable value, e.g. $f_i(X,Y,Z)$, $Z = a$.

In the above, X and Y are variables from the head of the clause, and a is a constant of the appropriate type. Literals of form (2) and (3) can be either positive or negative. Literals of the form $X = a$ under items (1) and (4) may also have the form $X > a$ and/or $X < a$, where a is a real-valued constant.

The attributes given to propositional learners are (1) the arguments of the target predicate, (2)–(3) binary valued attributes resulting from applications of utility predicates and (4) output arguments of utility functions. Attributes under (1) and (4) may be either discrete or real-valued. For cases (2) and (3) an attribute-value learning system will use conditions of the form A = *true* or A = *false* in the induced rules, where A is an attribute (cf. example in Section 2.2.3). These are transcribed to literals A and *not* A, respectively, in the DHDB clauses. For case (1) an attribute–value system will use conditions of the form $X = a$, $X > a$ or $X < a$, which can be immediately used in DHDB clauses. For case (4), in addition to the conditions $Z = a$, $Z > a$ and $Z < a$, the literal $f_i(X,Y,Z)$ has to be added to the DHDB clause so that the value of Z can be computed from the arguments of the target predicate.

To guide induction, LINUS uses meta-level knowledge which can exclude any of the above four cases, thus reducing the search space. For example, if only case (1) is retained, the hypothesis language is restricted to an attribute-value language. This can be achieved by using only the arguments of the target relation as attributes for learning. Cases (2)–(4), on the other hand, result from the use of predicates/functions from $\mathcal{B}$ as attributes for learning. The *equality* predicate (=/2), which generates literals of the form (2), is built-in in LINUS.

The complexity of learning with LINUS is analysed in Section 2.2.4 and shows the number of new attributes leading to literals of form (1)–(3) above. A detailed analysis of the actual hypothesis space, including the number of attributes which result in literals of form (4), can be found in (Džeroski and Lavrač 1992).

3.1.3. *Language biases for two typical learning tasks.* As mentioned above, meta-level knowledge can be used to adjust the language bias in LINUS, which is done by the user according to the type of learning task at hand. Below we list two types of tasks with the appropriate settings of the language bias:

Class learning mode: As an attribute-value learner, LINUS is used to induce descriptions of individual classes. In this case, the classes can be different from $\oplus$ and $\ominus$ and are determined by the values of a selected argument (or a set of arguments) of the target relation. The induced clauses have the form

$$class(Class) \leftarrow L_1, \dots, L_m$$

where *Class* is a class name and literals L_i can take any of the four forms outlined in Section 3.1.2.

Relation learning mode: As an ILP learner, LINUS induces constrained DHDB clauses of the form

$$p(X_1, \dots, X_n) \leftarrow L_1, \dots, L_m$$

where p is the name of the target predicate and literals L_i have any of the forms (1)–(4) from Section 3.1.2. In most of the experiments reported elsewhere (Lavrač *et al.* 1991a, Lavrač and Džeroski 1992) LINUS was used to learn function-free clauses; thus only literals of form (2) and (3) were actually used.

To summarize, the language bias in LINUS is declarative. The user can set the language bias to any of the above forms, depending on the learning task at hand.

3.2. *Weakening the language bias in LINUS*

In this section, the language bias of i-determinacy is described and illustrated by a simple example.

3.2.1. *The* i*-determinacy bias.* To weaken the language bias in LINUS and allow for more expressiveness, an idea from GOLEM is borrowed, also used in FOIL2.0 (Quinlan 1991). While LINUS allows only 'old' variables from the head of a clause, the idea of determinacy allows for a restricted form of 'new' variables to be introduced into the learned clauses. The very type of restriction (determinacy) allows the LINUS transformation approach to be used in this case as well.

It is assumed that an integer constant j is given. Only the class of ILP problems where all predicates in $\mathcal{B}$ are of arity at most j will be considered. The notion of *variable depth* is first defined.

Variable depth: Consider a clause $p(X_1,X_2, \dots,X_n) \leftarrow L_1,L_2,\dots,L_r, \dots$ Variables that appear in the head of the clause have *depth* zero. Let a variable V appear first in literal L_r. Let d be the maximum depth of any of the other variables in

L_r that appear in the clause $p(X_1, X_2, \ldots, X_n) \leftarrow L_1, L_2, \ldots, L_{r-1}$. Then the *depth* of variable V is $d + 1$.

By setting a maximum variable depth i, the syntactic complexity of clauses in the hypothesis language is restricted.

The following definition of *determinacy*, adapted to the case of function-free clauses, is taken from (Džeroski *et al.* 1992b).

Determinacy: A predicate definition is *determinate* if all of its clauses are determinate. A clause is *determinate* if each of its literals is determinate. A literal is *determinate* if each of its variables that do not appear in preceding literals has only one possible binding given the bindings of its variables that appear in preceding literals.

i-determinacy: The determinacy restriction is called *i-determinacy* for a given maximum variable depth i. Given the arity bound j, *i-determinacy* implies *ij-determination* as defined by Muggleton and Feng (1990).

Note that while i and j restrict the syntactic complexity of clauses, the notion of determinacy is essentially a semantic restriction, as it depends on the given training examples and background knowledge.

If we set the hypothesis language to the language of i-determinate clauses, the transformation approach of LINUS is still applicable, as demonstrated in Section 4. Determinate literals are a natural extension of the utility functions used presently in LINUS. In the light of the above definitions we can say that, in the current implementation of LINUS, only variables of depth 0 and 1 are allowed and that variables of depth 1 may not be used in other literals, except as described in Section 3.1.2.

The arity bound j is fixed. On the other hand, the parameter i can be increased to increase the expressiveness of the hypothesis language. For a fixed j, the user can consider the following series of languages:

$$\mathcal{L}_0 - \mathcal{L}_1 - \mathcal{L}_2 - \ldots$$

where the expressiveness of $\mathcal{L}_i$ (and thus the complexity of the search space) grows with i. The language $\mathcal{L}_0$ is the language of constrained function-free DHDB clauses. If a solution for the ILP problem cannot be found in the selected language, the next language in the series may be chosen, and the learning process repeated until a complete and consistent solution is found. Along these lines, LINUS could, in principle, shift its bias dynamically, similarly to CLINT (De Raedt 1992) and NINA (Adé and Bruynooghe 1992).

However, there are several problems with shifting bias dynamically. First of all, the series of languages considered has to be in some sense complete. Consider, for example, the case where the target definition is not determinate (is not in any of the languages in the series). In this case the system may continue to shift its bias forever without finding a satisfactory solution. Next, the learning process is repeated for each language in the series, up to the appropriate one. This leads to less efficient learning, although some improvements are possible. The most important problem, however, is when and how to shift the bias if learning data is imperfect, which can easily be the case in empirical ILP. Therefore, we will rather assume a fixed language bias $\mathcal{L}_i$.

3.2.2. *An example determinate definition.* Let us illustrate the notion of i-determinacy on a simple ILP problem. The task is to define the predicate *grandmother*, where *grandmother*(X,Y) states that person X is grandmother of person Y, in terms of the background knowledge predicates *father* and *mother*. The training examples and background knowledge are given in Table 6.

A correct target predicate definition is:

$$grandmother(X,Y) \leftarrow father(Z,Y), mother(X,Z)$$
$$grandmother(X,Y) \leftarrow mother(U,Y), mother(X,U)$$

This hypothesis can be induced in a language which is at least as expressive as $\mathcal{L}_1$. The clauses are determinate (but not constrained), because each occurrence of a new variable (i.e. Z in *father*(Z,Y) and U in *mother*(U,Y)) has only one possible binding given particular values of the other (old) variables in the literal (i.e. Y in this case); this is the case since each person has exactly one mother and father. The hypothesis is function-free and the maximum depth of any variable is one ($i = 1$). The arity bound j has to be at least two to allow induction of the above definition.

However, the logically equivalent hypothesis

$$grandmother(X,Y) \leftarrow mother(X,Z), father(Z,Y)$$
$$grandmother(X,Y) \leftarrow mother(X,U), mother(U,Y)$$

is not determinate, since the new variable Z in the literal *mother*(X,Z) can have more than one binding for a fixed value of X (e.g. X = *ann*, Z = *tom* or Z = *jim*); each person can namely have several children. We can thus see that the property of determinacy of a clause does not depend only on the given training examples and background knowledge, but also on the ordering of literals in the body of a clause.

4. Learning determinate clauses with LINUS

In this section, LINUS is treated exclusively as an ILP learner using only background knowledge in the form of predicates which may have both input and output arguments. We show how to weaken the language bias in LINUS and learn determinate DDB clauses. An algorithm that transforms the problem of learning nonrecursive function-free i-determinate clauses into a propositional form is first presented and then illustrated on the example from Section 3.2.2.

Table 6. A simple ILP problem: Learning the *grandmother* relationship

Training examples		*Background knowledge*		
grandmother(ann,bob)	⊕	father(zak,tom)	father(pat,ann)	father(zak,jim)
grandmother(ann,sue)	⊕	mother(ann,tom)	mother(liz,ann)	mother(ann,jim)
grandmother(bob,sue)	⊖	father(tom,sue)	father(tom,bob)	father(jim,dave)
grandmother(tom,bob)	⊖	mother(eve,sue)	mother(eve,bob)	mother(jean,dave)

The back transformation of induced propositional concept descriptions to the program clause form is then briefly described and illustrated by an example. Finally, we discuss how the algorithm can be extended to learn recursive clauses. Dealing with nonrecursive clauses is easier, as the recursive case may require querying the user (oracle) in order to complete the transformation process.

4.1. *Learning nonrecursive determinate clauses*

For simplicity, we first consider the problem of learning nonrecursive clauses. The transformation of the ILP problem of constructing an i-determinate definition for target predicate $p(X_1,X_2, \ldots,X_n)$ from examples $\mathcal{E}$ and from predicate definitions in $\mathcal{B}$ is described below.

4.1.1. *The transformation algorithm.* The algorithm consists of the following steps (the first step contains the most important differences to the original LINUS algorithm described in Section 2.2.3).

- Transform the ILP problem to propositional form.

  (1) Construct a list L of determinate literals that introduce new variables of depth at most i. Construct a list V_i of old variables and new variables introduced by determinate literals from L.

  (2) Construct a list F of all literals that use predicates from the background knowledge and variables from V_i. The determinate literals from L are excluded from this list, as they do not distinguish between positive and negative examples (since the new variables have a unique binding for each example, due to the determinacy restriction). The resulting list is the list of features (attributes) used for propositional learning.

  (3) Transform the examples to propositional form. For each example, the truth value of each of the propositional features is determined by calls to the background knowledge $\mathcal{B}$.

- Apply a propositional learning algorithm to induce a propositional concept description.
- Transform the induced propositional description to a set of determinate DDB clauses, adding the necessary determinate literals which introduced the new variables.

Algorithm 1 from Figure 3 describes the transformation of the ILP problem into propositional form, i.e. the first step of the above algorithm for learning nonrecursive determinate clauses. The algorithm assumes that the learning task is to find a predicate definition as a set of determinate DDB clauses with $p(X_1,X_2, \ldots,X_n)$ in the head. Given are the training examples $\mathcal{E}$ as ground facts defining predicate p/n, the language bias $\mathcal{L}_i$ of i-determinate DDB clauses, the definitions of background knowledge predicates q_s/n_s in $\mathcal{B}$, and the types of arguments of predicates p/n and q_s/n_s. The output of the algorithm is a set of examples $\mathcal{E}_f$ for attribute-value learning formed of tuples f labelled $\oplus$ for $p(a_1,a_2, \ldots,a_n) \in \mathcal{E}^+$ and labelled $\ominus$ for $p(a_1,a_2, \ldots,a_n) \in \mathcal{E}^-$.

```
Algorithm 1: Learning nonrecursive determinate clauses
Initialize the list of variables $V_0 := \{X_1, X_2, \dots, X_n\}$.
Initialize the list of literals $L := \emptyset$.
Initialize the set of tuples $\mathscr{C}_f := \emptyset$.

1. for $r = 1$ to $i$ do
   • $D_r := \{q_s(Y_1,Y_2,\dots,Y_{j_s}) | q_s \in \mathscr{B}, Y_1,Y_2,\dots,Y_{j_s}$ are of the appropriate
     types, $q_s(Y_1,Y_2,\dots,Y_{j_s})$ is determinate and contains at least one new
     variable not in $V_{r-1}\}$.
   • $L := L \cup D_r$.
   • $V_r := V_{r-1} \cup \{Y | Y$ appears in a literal from $D_r\}$.
   endfor
2. $F := \{q_s(Y_1,Y_2,\dots,Y_{j_s}) | q_s \in \mathscr{B}, Y_1,Y_2,\dots,Y_{j_s} \in V_i\} - L$.
3. for each $p(a_1,a_2,\dots,a_n) \in \mathscr{E}$ do
   • Determine the values of variables in $V_i$ by executing the body of the
     clause $p(X_1,X_2,\dots,X_n) \leftarrow L$ with variables $X_1,\dots,X_n$ bound to $a_1,\dots,a_n$.
   • Given the values of variables in $V_i$, determine the tuple $f$ of truth values
     of literals in $F$, by querying the background knowledge $\mathscr{B}$.
   • $\mathscr{C}_f := \mathscr{C}_f \cup \{f\}$.
   endfor
```

Figure 3. The algorithm for learning nonrecursive determinate clauses.

The knowledge base $\mathscr{B}$ of background predicates may take the form of a (normal) logic program. In step (3) of Algorithm 1, two types of queries have to be posed to this knowledge base.

- The first type are *existential* queries, which are used to determine the values of the new variables. Given a partially instantiated goal (literal containing variables that do not appear in previous literals), an existential query returns the set (possibly empty) of all bindings for the unbound variables which make the literal true. For example, the query *mother*(X,A) where $X =$ *ann* and A is a new variable would return the set of answers $\{A =$ *tom*, $A =$ *jim*$\}$. Note, however, that for the determinate literals in L, the set of answers is always a singleton.

Table 7. Propositional form of the *grandmother* learning problem

$g(X,Y)$	*Variables*		*New variables*				*Propositional features*		
	X	Y	$f(U,X)$	$f(V,Y)$	$m(W,X)$	$m(Z,Y)$	… $m(X,V)$	$m(X,Z)$	…
Class	X	Y	U	V	W	Z	A_1	A_2	
⊕	ann	bob	pat	tom	liz	eve	true	false	
⊕	ann	sue	pat	tom	liz	eve	true	false	
⊖	bob	sue	tom	tom	eve	eve	false	false	
⊖	tom	bob	zak	tom	ann	eve	false	false	

- The other type of queries are ground (*membership*) queries about background knowledge predicates, where the goal is completely bound. These are used to determine the truth values of the propositional features.

In an actual implementation of Algorithm 1, steps (1) and (3) should be interleaved, i.e. the values of the new variables and the propositional features should be calculated for each example as they are introduced. In this way, the determinacy of literals in step (1) is automatically tested when the existential query determining the values of the new variables in a literal is posed. A literal is determinate if the set of answers to the existential query is a singleton for all the examples. If the set of answers for some example is not a singleton, the literal is not determinate.

4.1.2. *The algorithm at work: an example.* For the grandmother example, given in Table 6 of Section 3.2.2, we have $i = 1$, $l = 2$, and $n = 2$. Let $j \geq 2$. The literal *father*(X,A), where X is an old and A is a new variable, is not determinate (e.g. $X = tom$, $A = sue$ or $A = bob$). However, if A is old and X is new, the literal is determinate. As the target predicate is *grandmother*(X,Y), we have $V_0 = \{X,Y\}$ and $D_1 = \{f(U,X), f(V,Y), m(W,X), m(Z,Y)\}$, where f and m stand for *father* and *mother*, respectively.

This gives $L = D_1$, $V_1 = \{X,Y,U,V,W,Z\}$. The list F includes literals such as $f(X,X)$, $f(X,Y)$, $f(Z,Y)$, $f(W,X)$ and similarly $m(Z,Z)$, $m(V,Y)$, $m(W,W)$, $m(U,X)$, $m(X,V)$, $m(X,Z)$. In fact, the pairs of arguments of f and m are all the pairs from the Cartesian product $V_1 \times V_1$, excluding the pairs that produce literals from D_1.

The transformation process for the ILP problem as defined by the training examples and background knowledge from Table 6 is illustrated in Table 7. For the two positive and the two negative examples of $g(X,Y)$ (g stands for *grandmother*), given are the values of the old variables X and Y, the values of the new variables U, V, W and Z introduced by the determinate literals from the list L, and the values of two of the propositional features from the list F, namely $m(X,V)$ and $m(X,Z)$, denoted by A_1 and A_2, respectively. Note that only the propositional features (attributes A_1,A_2, ...) are actually considered in the attribute-value learning task.

4.1.3. *Back transformation to DDB clause form.* The algorithm that converts the induced propositional concept definition to DDB clause form is fairly straightforward. First, transcribe the induced rules into DDB clauses (as in the constrained case). Then add in the necessary determinate literals, ensuring that the values of the new variables can be uniquely determined from the value of old variables. For each clause repeat the following process, until all variables in the body of the clause and not in its head are introduced by determinate literals: choose a new variable and add to the clause the literal in L that introduced the new variable. The literals are then ordered according to the maximum depth of variables appearing in them.

To illustrate this transformation to a set of DDB clauses, the grandmother example is used. Suppose that a propositional learner induces the following two rules from the examples in Table 7:

$$Class = \oplus \textbf{ if } [A_1 = true]$$

$$Class = \oplus \text{ if } [A_2 = true]$$

This description is consistent with the examples, i.e. does not cover any negative example. (Note, however, that it is not too probable that any propositional learner would actually induce this description, since the first rule would suffice for discriminating between the positive and negative examples.) The two rules are transcribed to the clauses:

$$grandmother(X,Y) \leftarrow mother(X,V)$$
$$grandmother(X,Y) \leftarrow mother(X,Z)$$

The new variables V and Z in the literals $mother(X,V)$ and $mother(X,Z)$ are not introduced by determinate literals, so the literals $father(V,Y)$ and $mother(Z,Y)$ from L have to be added to the first and second clause, respectively. The final form of the logic program would then be:

$$grandmother(X,Y) \leftarrow father(V,Y), mother(X,V)$$
$$grandmother(X,Y) \leftarrow mother(Z,Y), mother(X,Z)$$

Expressed in logic, this program stands for:

$$\forall X \forall Y{:}grandmother(X,Y) \leftrightarrow [\exists V{:}f(V,Y) \wedge m(X,V)] \vee [\exists Z{:}m(Z,Y) \wedge m(X,Z)]$$

or more precisely for:

$$\forall X \forall Y{:}grandmother(X,Y) \leftrightarrow [\exists! V{:}f(V,Y) \wedge m(X,V)] \vee [\exists! Z{:}m(Z,Y) \wedge m(X,Z)]$$

as the value of each of the new variables is uniquely determined.

4.2. Learning recursive determinate clauses

To learn recursive determinate definitions, Algorithm 1 needs to be slightly modified (Džeroski *et al.* 1992b). Let us first distinguish between the case of recursive literals which do not and recursive literals which do introduce new variables. In the first case, membership queries are needed, and in the second, both membership and existential queries about the target relation.

In the first case (when recursive literals do not introduce new variables), only steps (2) and (3) of Algorithm 1 need to be modified. In step (2) which constructs the list of literals F, we treat the target predicate p as any of the other background knowledge predicates and form all possible literals with it and the variables available, excluding the literal $p(X_1,X_2, \ldots,X_n)$. However, in step (3) we cannot evaluate features involving p by querying the background knowledge. In that case we first check whether the ground query is among the training examples for p, in which case we can determine its truth value (*true*, if the example is positive, *false* if negative). If, however, the query cannot be found among the examples, we have to resort to a membership query, that is, we must ask the user whether the answer to the query involved is true or false.

To illustrate the above modification to Algorithm 1, consider the task of learning the relation $member(X,Y)$, where X is of type *element* and Y is of type *list*. The background knowledge given includes the predicate $components(X,Y,Z)$, where X and Z are of type *list* and Y is of type *element*. This predicate is defined as $components([A|B],A,B)$ and decomposes a list $[A|B]$ into its head A and tail B. It is a flattened version of the *cons* function symbol. The equality predicate which works on arguments of the same type is also given. In this case j has to be at least 3. The maximum depth of variables is set to $i = 1$.

Table 8 gives four training examples and their transformation into the propositional form. From the table we can see that the only determinate literal is $c(Y,A,B)$ (c and m stand for *components* and *member*, respectively), and $V_1 = \{X,Y,A,B\}$, where X and A are of type *element* and Y and B are of type *list*. Taking into account the types of arguments, the modified algorithm produces the list of features in the table. The literals $c(Y,X,Y)$, $c(Y,A,Y)$, $c(B,X,B)$, $c(B,A,B)$, $c(B,X,Y)$ and $c(B,A,Y)$ are also on the list of features, but have been excluded for the sake of readability (they always have the value *false*). Consider the propositional feature A_4 which corresponds to the recursive call *member*(X,B). The value of this feature can be determined for the first example, since the ground query is in this case *member*(1,[2]), which can be found as a negative training example. However, for the other three examples, membership queries have to be posed (the answers are marked with an asterisk '*'). A similar observation holds for features A_5 and A_6.

A propositional learner might generate the following rules from the given propositional examples:

$$\textit{Class} = \oplus \textbf{ if } [A_2 = \textit{true}]$$
$$\textit{Class} = \oplus \textbf{ if } [A_4 = \textit{true}]$$

The rules would be transformed to the following definition:

$$\textit{member}(X,Y) \leftarrow \textit{components}(Y,A,B), = (X,A)$$
$$\textit{member}(X,Y) \leftarrow \textit{components}(Y,A,B), \textit{member}(X,B)$$

which is the correct definition of the concept *member*(X,Y).

The second case occurs if recursive literals with new variables are allowed, which are, for example, necessary for learning the *quicksort* program. In addition to the changes outlined above, the first substep of step (1) of Algorithm 1 has also to be changed. In this step, the target predicate p is treated exactly as the other predicates. Determining the values of the new variables in literals involving the target predicate requires in this case the use of *existential* queries about the target concept.

The worst-case complexity of learning determinate database clauses can be estimated as follows. Given l predicates in $\mathcal{B}$, a target predicate of arity n, an arity bound j and a constant bound i on variable depth, Algorithm 1 generates $\mathcal{O}(l((jl+1)n)^{j^{i+1}})$ features (Džeroski *et al.* 1992b). Assuming $\mathcal{B}$ is in some sense

Table 8. Propositional form of a recursive ILP problem

m(X,Y)	*Vars*		*New vars*		*Propositional features*					
	X	*Y*	*c(Y,A,B)*		*c(Y,X,B)*	*X=A*	*Y=B*	*m(X,B)*	*m(A,Y)*	*m(A,B)*
Class	*X*	*Y*	*A*	*B*	A_1	A_2	A_3	A_4	A_5	A_6
⊕	1	[1,2]	1	[2]	true	true	false	false	true	false
⊕	1	[2,1]	2	[1]	false	false	false	true*	true*	false*
⊕	2	[2]	2	[]	true	true	false	false*	true	false*
⊖	1	[2]	2	[]	false	false	false	false*	true	false*

efficient, the transformation process can be completed in polynomial time (Džeroski *et al.* 1992b). Using a suitable propositional learning algorithm and the corresponding PAC-learnability results (Li and Vitanyi 1991), it was proved that determinate function-free programs are PAC-learnable under simple distributions (Džeroski *et al.* 1992b). Simple distributions (Li and Vitanyi 1991) assign higher probabilities to simpler examples; for instance, the example *member*(1,[1]) should be much more likely than *member*(3,[7,1,4,3,2,8]) under a simple distribution.

The above upper bound of features (attributes) is fortunately not realistic for computing the actual number of attributes to be used for learning. Due to type restrictions, symmetry of predicates and determinacy, the actual number of attributes is substantially smaller, as can be seen in the *member* example above. However, one has to be aware of the drastical increase in complexity with the arity bound j and the depth i in the language bias $\mathcal{L}_i$.

5. Conclusions

The paper presents a method which allows for effective use of background knowledge in inductive concept learning. An algorithm which implements this method can incorporate state-of-the-art propositional learners. It is important to emphasize that this allows for learning from real-valued and noisy data, both in the propositional and the ILP framework. Compared to propositional attribute-value learners, the advantage is the effective use of background knowledge and learning in a more expressive logic programming framework. On the other hand, compared to ILP learners, the advantage is the ability to handle real-valued and noisy data.

An analysis of the declarative language bias imposed by the method is given. For the constrained DHDB clause bias, the method has already been implemented in the LINUS learning system. This bias can be weakened to learn concept descriptions in the more expressive language of determinate (deductive) database clauses. It is shown how to extend the LINUS algorithm to handle this case.

Compared to an another empirical ILP system FOIL, the proposed extension to LINUS has the disadvantage that the transformation approach computes the values of all possible features beforehand. On the other hand, FOIL computes the values of features dynamically, i.e. during the heuristic evaluation process, and is thus potentially more efficient. It can also use induce non-determinate logic programs, while our approach is restricted to determinate programs.

As compared to GOLEM and FOIL, our approach has the following advantages. The concept description language based on i-determinate clauses is more expressive than the one used in GOLEM. Namely, negated literals may be used in LINUS whereas only definite program clauses can be induced by GOLEM. Furthermore, unlike in FOIL and GOLEM, non-ground background knowledge may be used directly. In FOIL and GOLEM, non-ground background knowledge may be used, but has to be converted to a ground model, i.e. a set of ground unit clauses, by carrying out all h-easy derivations starting from the constant symbols in the observations. Bratko *et al.* (1992) report serious problems resulting from this requirement, due to the huge number of ground facts generated.

The presented methodology was also used to prove PAC-learnability results for constrained and determinate logic programs (Džeroski *et al.* 1992a,b). Despite negative results by Haussler (1989), which indicate that non-determinate logic programs are probably not PAC-learnable, it would be interesting to explore the limits of this methodology, i.e. to investigate whether it would allow for efficient learning of non-determinate logic programs.

Acknowledgements
This work is part of the ESPRIT III Basic Research Project No. 6020 Inductive Logic Programming. It was also supported by the Slovenian Ministry of Science and Technology and in part by the Belgian State-Science Policy Office grant, enabling Nada Lavrač her six months leave of absence at the Katholieke Universiteit Leuven, and the British Council grant enabling Sašo Džeroski his eight month leave of absence at The Turing Institute Limited, Glasgow. We are grateful to Igor Mozetič and Marko Grobelnik for their contribution to the development of LINUS. We would also like to thank Stephen Muggleton and Stuart Russell for the co-operation in establishing the PAC-learnability results. Thanks to Peter Flach and Irma Sutlič for their comments on the paper. Special thanks go to Luc De Raedt for his insightful and stimulating review.

References

Adé, H. and Bruynooghe, M. (1992) A comparative study of declarative and dynamically adjustable language bias in concept learning. In *Proceedings of Workshop on Biases in Inductive Learning, Ninth International Conference on Machine Learning,* Aberdeen.

Bergadano, F., Giordana, A. and Ponsero, S. (1989) Deduction in top-down inductive learning. In *Proceedings of Sixth International Workshop on Machine Learning* (San Mateo: Morgan Kaufmann), 23–25.

Bratko, I. (1990) *Prolog Programming for Artificial Intelligence*, 2nd edition (Wokingham: Addison-Wesley).

Bratko, I., Mozetič, I. and Lavrač, N. (1989) *KARDIO: A Study in Deep and Qualitative Knowledge for Expert Systems* (Cambridge: MIT Press).

Bratko, I., Muggleton, S. and Varšek, A. (1992) Learning qualitative models of dynamic systems. In S. Muggleton (ed.) *Inductive Logic Programming* (London: Academic Press), 437–452.

Cestnik, B., Kononenko, I. and Bratko, I. (1987) ASSISTANT 86: A knowledge elicitation tool for sophisticated users. In I. Bratko and N. Lavrač (eds) *Progress in Machine Learning* (Wilmslow: Sigma Press), 31–45.

Clark, P. and Niblett, T. (1989) The CN2 induction algorithm, *Machine Learning,* **3**(4): 261–283.

De Raedt, L. (1992) *Interactive Theory Revision: An Inductive Logic Programming Approach* (London: Academic Press).

De Raedt, L. and Bruynooghe, M. (1990) Indirect relevance and bias in inductive concept-learning, *Knowledge Acquisition,* **2**(4): 365–390.

De Raedt, L. and Bruynooghe, M. (1992) Interactive concept learning and constructive induction by analogy, *Machine Learning,* **8**(2): 107–150.

Džeroski, S. (1991) Handling noise in inductive logic programming. Master's thesis Faculty of Electrical Engineering and Computer Science, University of Ljubljana, Ljubljana, Slovenia.

Džeroski, S. and Lavrač, N. (1991) Learning relations from noisy examples: An empirical comparison of LINUS and FOIL. In *Proceedings of Eight International Workshop on Machine Learning* (San Mateo: Morgan Kaufmann), 399–402.

Džeroski, S. and Lavrač, N. (1992) Refinement graphs for FOIL and LINUS. In S. Muggleton (ed.) *Inductive Logic Programming* (London: Academic Press), 319–333.

Džeroski, S., Muggleton, S. and Russell, S. (1992a) PAC-learnability of constrained nonrecursive logic programs. In *Proceedings of Third International Workshop on Computational Learning Theory and Natural Learning Systems*, Wisconsin.

Džeroski, S., Muggleton, S. and Russell, S. (1992b) PAC-learnability of determinate logic programs. In *Proceedings of Fifth ACM Workshop on Computational Learning Theory* (Baltimore: ACM Press), 128–135.

Fu, L. M. and Buchanan, B. G. (1985) Learning intermediate concepts in constructing a hierarchical knowledge base. In *Proceedings of Ninth International Joint Conference on Artificial Intelligence* (Los Altos: Morgan Kaufmann), 659–666.

Haussler, D. (1989) Learning conjunctive concepts in structural domains, *Machine Learning,* **4**: 7–40.

Kietz, J. and Wrobel, S. (1992) Controlling the complexity of learning in logic through syntactic and task-oriented models. In S. Muggleton (ed.) *Inductive Logic Programming* (London: Academic Press), 335–359.

Lavrač, N. and Džeroski, S. (1992) Inductive learning of relations from noisy examples. In S. Muggleton (ed.) *Inductive Logic Programming* (London: Academic Press), 495–514.

Lavrač, N., Džeroski, S. and Grobelnik, M. (1991a) Learning nonrecursive definitions of relations with LINUS. In *Proceedings of Fifth European Working Session on Learning* (Berlin: Springer), 265–281.

Lavrač, N., Džeroski, S., Pirnat, V. and Križman, V. (1991b) Learning rules for early diagnosis of rheumatic diseases. In *Proceedings of Third Scandinavian Confernece on Artificial Intelligence* (Amsterdam: IOS Press), 138–149.

Lavrač, N., Džeroski, S., Pirnat, V. and Križman, V. (1993) The utility of background knowledge in learning medical diagnostic rules, *Applied Artificial Intelligence*, **7**: 273–293.

Li, M. and Vitányi, P. (1991) Learning simple concepts under simple distributions, *SIAM Journal of Computing,* **20**(5): 911–935.

Lloyd, J. (1987) *Foundations of Logic Programming*, 2nd edition (Berlin: Springer).

Michalski, R. (1980) Pattern recognition as rule-guided inductive inference, *IEEE Transactions on Pattern Analysis and Machine Intelligence,* **2**(4): 349–361.

Michalski, R. (1983) A theory and methodology of inductive learning. In R. S. Michalski, J. C. Carbonell and T. Mitchel (eds) *Machine Learning: An Artificial Intelligence Approach*, volume I (Palo Alto, CA: Tioga), 83–134.

Michalski, R., Mozetič, I., Hong, J. and Lavrač, N. (1986) The multi-purpose incremental learning system AQ15 and its testing application on three medical domains. In *Proceedings of Fifth National Conference on Artificial Intelligence* (San Mateo: Morgan Kaufmann), 1041–1045.

Mozetič, I. (1987) Learning of qualitative models. In I. Bratko and N. Lavrač (eds) *Progress in Machine Learning* (Wilmslow: Sigma Press), 201–217.

Muggleton, S. (1987) DUCE: An oracle-based approach to constructive induction. In *Proceedings of Tenth International Joint Conference on Artificial Intelligence* (San Mateo: Morgan Kaufmann), 287–292.

Muggleton, S. (1991) Inductive logic programming, *New Generation Computing,* **8**(4): 295–318.

Muggleton, S. (ed.) (1992) *Inductive Logic Programming* (London: Academic Press).

Muggleton, S. and Buntine, W. (1988) Machine invention of first-order predicates by inverting resolution. In *Proceedings of Fifth International Conference on Machine Learning* (San Mateo: Morgan Kaufmann), 339–352.

Muggleton, S. and Feng, C. (1990) Efficient induction of logic programs. In *Proceedings of First Conference on Algorithmic Learning Theory* (Tokyo: Ohmsha), 368–381.

Nunez, M. (1991) The use of background knowledge in decision tree induction, *Machine Learning,* **6**(3): 231–250.

Plotkin, G. (1969) A note on inductive generalization. In B. Meltzer and D. Michie (eds) *Machine Intelligence 5* (Edinburgh: Edinburgh University Press), 153–163.

Quinlan, J. (1986) Induction of decision trees, *Machine Learning,* **1**(1): 81–106.

Quinlan, J. (1990) Learning logical definitions from relations, *Machine Learning,* **5**(3): 239–266.

Quinlan, J. (1991) Knowledge acquisition from structured data—using determinate literals to assist search, *IEEE Expert,* **6**(6): 32–37.

Russell, S. (1989) *The Use of Knowledge in Analogy and Induction* (London: Pitman).

Sammut, C. and Banerji, R. (1986) Learning concepts by asking questions. In R. Michalski, J. Carbonell and T. Mitchell (eds) *Machine Learning: An Artificial Intelligence Approach*, volume II, (San Mateo, CA: Morgan Kaufmann), 167–191.

Shapiro, E. (1983) *Algorithmic Program Debugging* (Cambridge: MIT Press).

Ullman, J. (1988) *Principles of Database and Knowledge Base Systems*, volume I (Rockville: Computer Science Press).

Utgoff, P. and Mitchell, T. M. (1982) Acquisition of appropriate bias for inductive concept learning. In *Proceedings of National Conference on Artificial Intelligence* (Los Altos: Morgan Kaufmann), 414–417.

Wnek, J., Sarma, J., Wahab, A. A. and Michalski, R. S. (1990) Comparing learning paradigms via diagrammatic visualization: A case study in single concept learning using symbolic, neural net and genetic algorithm methods. In *Proceedings of Fifth International Symposium on Methodologies for Intelligent Systems*, Knoxville.

Wrobel, S. (1988) Automatic representation adjustment in an observational discovery system. In *Proceedings of Third European Working Session on Learning* (London: Pitman), 253–262.