An Extended Transformation Approach to Inductive Logic Programming

NADA LAVRAČ
Jožef Stefan Institute
and
PETER A. FLACH
University of Bristol

Inductive logic programming (ILP) is concerned with learning relational descriptions that typically have the form of logic programs. In a transformation approach, an ILP task is transformed into an equivalent learning task in a different representation formalism. Propositionalization is a particular transformation method, in which the ILP task is compiled to an attribute-value learning task. The main restriction of propositionalization methods such as LINUS is that they are unable to deal with nondeterminate local variables in the body of hypothesis clauses. In this paper we show how this limitation can be overcome, by systematic first-order feature construction using a particular individual-centered feature bias. The approach can be applied in any domain where there is a clear notion of individual. We also show how to improve upon exhaustive first-order feature construction by using a relevancy filter. The proposed approach is illustrated on the "trains" and "mutagenesis" ILP domains.

Categories and Subject Descriptors: I.2.3 [**Artificial Intelligence**]: Deduction and Theorem Proving—*Logic programming*; I.2.4 [**Artificial Intelligence**]: Knowledge Representation Formalisms and Methods—*Predicate logic*; I.2.6 [**Artificial Intelligence**]: Learning—*Concept learning*; *Induction*

General Terms: Algorithms, Experimentation

Additional Key Words and Phrases: Data mining, inductive logic programming, machine learning, relational databases

1. INTRODUCTION

Inductive logic programming (ILP) [Muggleton 1992; Muggleton and De Raedt 1994; Lavrač and Džeroski 1994; Bergadano and Gunetti 1995; Nienhuys-Cheng and de Wolf 1997] is a research area that has its backgrounds in inductive machine learning and computational logic. ILP research aims at a formal framework as well as practical algorithms for inductive learning of relational descriptions that typically have the form of logic programs. From computational

Authors' addresses: N. Lavrač, Jožef Stefan Institute, Jamova 39, 1000 Ljubljana, Slovenia; email: Nada.Lavrac@ijs.si; P.A. Flach, University of Bristol, Woodland Road, Bristol BS8 1UB, United Kingdom; email: Peter.Flach@bristol.ac.uk.

logic, ILP has inherited its sound theoretical basis, and from machine learning, an experimental approach and orientation toward practical applications. ILP research has been strongly influenced also by computational learning theory, and recently by knowledge discovery in databases [Fayyad et al. 1995] which led to the development of new techniques for relational data mining.

In general, an ILP learner is given an initial theory T (background knowledge) and some evidence E (examples), and its aim is to induce a theory H (hypothesis) that together with T *explains* some properties of E. In most cases the hypothesis H has to satisfy certain restrictions, which we shall refer to as the *bias*. Bias includes prior expectations and assumptions, and can therefore be considered as the logically unjustified part of the background knowledge. Bias is needed to reduce the number of candidate hypotheses. It consists of the language bias L, determining the hypothesis space, and the search bias which restricts the search of the space of possible hypotheses.

The background knowledge used to construct hypotheses is a distinctive feature of ILP. It is well known that relevant background knowledge may substantially improve the results of learning in terms of accuracy, efficiency, and the explanatory potential of the induced knowledge. On the other hand, irrelevant background knowledge will have just the opposite effect. Consequently, much of the art of inductive logic programming lies in the appropriate selection and formulation of background knowledge to be used by the selected ILP learner.

This work shows, that by devoting enough effort to the construction of features, to be used as background knowledge in learning, even complex relational learning tasks can be solved by simple propositional rule learning systems. In propositional learning, the idea of augmenting an existing set of attributes with new ones is known under the term constructive induction. A first-order counterpart of constructive induction is predicate invention. This work takes the middle ground: we perform a simple form of predicate invention through first-order feature construction, and use the constructed features for propositional learning. In this way we are able to show that the traditional limitations of transformation-based approaches such as the limited hypothesis language of LINUS (i.e., no local variables in clause bodies) and its successor DINUS (only determinate local variables) [Lavrač and Džeroski 1994] can be alleviated by means of nondeterminate first-order feature construction.

Our approach to first-order feature construction can be applied in the so-called *individual-centered* domains, where there is a clear notion of individual. Such domains include classification problems in molecular biology, for example, where the individuals are molecules. Individual-centered representations have the advantage of a strong language bias, because local variables in the bodies of rules either refer to the individual or to parts of it. However, not all domains are amenable to the approach we present in this paper—in particular, we cannot learn recursive clauses, and we cannot deal with domains where there is not a clear notion of individual (e.g., many program synthesis problems).

This paper first presents the main ILP setting, called the predictive ILP setting, and outlines some transformations of this basic setting (Section 2). Among these, propositionalization approaches to ILP are studied in detail, in particular the LINUS propositionalization approach (Section 3). The main goal of this

work is to show that the traditional limitations of LINUS and DINUS can be alleviated by means of nondeterminate feature construction used to define the LINUS background knowledge on the basis of structural properties (Section 4). We can improve upon exhaustive first-order feature construction by using a filter for eliminating irrelevant features (Section 5). The proposed approach to learning of nondeterminate clauses through first-order feature construction is illustrated by the "trains" and "mutagenesis" ILP domains. Related work is discussed in Section 6.

2. THE FRAMEWORK OF INDUCTIVE LOGIC PROGRAMMING

In its most general form, induction consists of inferring general rules from specific observations (also called examples, data-cases, etc.). This section introduces the basic ILP setting, which deals with inducing first-order classification rules. The definitions given below are fairly standard, except for a distinction between foreground and background predicates that was introduced in [Flach and Kakas 2000] in order to distinguish between abduction and induction. We also discuss some transformations of the basic setting, which are intended to reformulate the problem into an equivalent but easier solvable one.

2.1 The Basic ILP Setting

In learning of first-order classification rules, the induced rules should *entail* the observed examples. Roughly speaking, the examples establish partial extensional specifications of the predicates to be learned or *foreground predicates*, and the goal is to find intensional definitions of those predicates. In the general case, this requires suitably defined auxiliary or *background predicates* (simple recursive predicates such as `member/2` and `append/3` notwithstanding). The induced set of rules or *inductive hypothesis* then provides an intensional connection between the foreground predicates and the background predicates; we will sometimes call such rules *foreground-background* rules. We will also use the terms *facts* to refer to extensional knowledge, and *rules* to refer to intensional knowledge. The terms "knowledge" or "theory" may refer to both facts and rules. Thus, predictive induction infers foreground-background rules from foreground facts and background theory.

Definition 2.1 (*Predictive ILP*). Let P_F and N_F be sets of ground facts over a set of foreground predicates F, called the *positive examples* and the *negative examples*, respectively. Let T_B, the *background theory*, be a set of clauses over a set of background predicates B. Let L be a *language bias* specifying a *hypothesis language* $\mathcal{H}_L$ over $F \cup B$ (i.e., a set of clauses). A predictive ILP task consists in finding a *hypothesis* $H \subseteq \mathcal{H}_L$ such that $\forall p \in P_F : T_B \cup H \models p$ and $\forall n \in N_F : T_B \cup H \not\models n$.

The subscripts F and B are often dropped, if the foreground and background predicates are understood. Also, sometimes positive examples and negative examples are collectively referred to as examples E.

Definition 2.1 is underspecified in a number of ways. First, it does not rule out trivial solutions like $H = P$ unless this is excluded by the language bias

(which is not often the case, since the language bias cannot simply exclude ground facts, because they are required by certain recursive predicate definitions). Furthermore, the definition does not capture the requirement that the inductive hypothesis correctly predicts unseen examples. It should therefore be seen as a general framework, which needs to be further instantiated to capture the kinds of ILP tasks addressed in practice. We proceed by briefly discussing a number of possible variations, indicating which of these we can handle with the approach proposed in this paper.

Clauses in T and H are often restricted to definite clauses with only positive literals in the body. Some ILP algorithms are able to deal with normal clauses which allow negated literals in the body. One can go a step further and allow negation over several related literals in the body (so-called *features*). The transformation-based method we describe in Section 4 can handle each of these cases.

In a typical predictive ILP task, there is a single foreground predicate to be learned, often referred to as the *target predicate*. In contrast, *multiple predicate learning* occurs when $|F| > 1$. Multiple predicate learning is hard if the foreground predicates are mutually dependent, i.e., if one foreground predicate acts as an auxiliary predicate to another foreground predicate, because in that case the auxiliary predicate is incompletely specified. Approaches to dealing with incomplete background theory, such as abductive concept learning [Kakas and Riguzzi 1997], can be helpful here. Alternatively, multiple predicate learning may be more naturally handled by a descriptive ILP approach [Flach and Lachiche 2001], which is not intended at learning of classification rules but at learning of properties or constraints that hold for E given T. In this paper we restrict attention to predictive ILP tasks where there is a single target predicate.

The problems of learning recursive rules, where a foreground predicate is its own auxiliary predicate, are related to the problems of multiple predicate learning. In general, recursive rules are not amenable to transformation into propositional form without combinatorial explosion. We will not deal with learning recursive programs in this paper.

Definition 2.1 only applies to boolean classification problems. The definition could be extended to multiclass problems, by supplying the foreground predicate with an extra argument indicating the class. In such a case, a set of rules has to be learned for each class. It follows that we can also distinguish binary classification problems in which both the positive and negative class have to be learned explicitly (rather than by negation-as-failure, as in the definition). Most learning tasks in this paper are of this binary classification kind.

In *individual-centered* domains there is a notion of individual, e.g., molecules or trains, and learning occurs at the level of individuals only. Often, individuals are represented by a single variable, and the foreground predicates are either unary predicates concerning boolean properties of individuals, or binary predicates assigning an attribute-value or a class-value to each individual. It is however also possible that individuals are represented by tuples of variables (see Section 3.1.2 for an example). Local variables referring to parts of individuals are introduced by so-called structural predicates. Individual-centered

representations allow for a strong language bias for feature construction, and form the basis for the extended propositionalization method presented in Section 4.

Sometimes a predictive ILP task is unsolvable with the given background theory, but solvable if an additional background predicate is introduced. For instance, in Peano arithmetic, multiplication is not finitely axiomatizable unless the definition of addition is available. The process of introducing additional background predicates during learning is called *predicate invention*. Even when learning nonrecursive theories, in which case an equivalent hypothesis without the invented predicates is guaranteed to exist, predicate invention can be useful to guide learning, and to identify additional useful domain knowledge.[1] As mentioned in the introduction, in this paper we systematically construct new features to be used in learning, which can be seen as a simple form of predicate invention.

Finally, sometimes an initial foreground-background hypothesis H_0 may be given to the learner as a starting point for hypothesis construction. Such a situation occurs for example in incremental learning, where examples become available and are processed sequentially. Equivalently, we can perceive this as a situation where the background theory also partially defines the foreground predicate(s). This is usually referred to as *theory revision*. In this paper, we only address nonincremental learning without initial theory.

2.2 Transformations of the Basic Setting

The predictive ILP framework as discussed above has several degrees of freedom: the language biases for background knowledge and hypotheses, whether to use general background rules or only specific background facts, etc. One way of understanding these different possible settings is by studying the conditions under which one can be transformed into the other. In this section we briefly discuss some of these transformations, including an extreme form called *propositionalization*, which compiles a predictive ILP task down to an attribute-value learning task.

2.2.1 *Flattening.* The background theory defines a number of auxiliary predicates for use in the bodies of hypothesis clauses. Many of these will be defined intensionally, but some may be specified extensionally. Typically, background facts occur as artefacts of using a function-free representation. We may either have a function-free hypothesis language, or function-free background theory and examples as well. The transformation to a function-free language is called *flattening* [Rouveirol 1994].

Example 2.2 (*Flattening*). Suppose our background theory contains the usual definition of `append/3`:

```
append([],Ys,Ys).
append([X|Xs],Ys,[X|Zs]):-append(Xs,Ys,Zs).
```

[1]Predicate invention can be seen as an extreme form of multiple predicate learning where some of the foreground predicates have no examples at all.

Given a few foreground facts such as `p([1,2],[2,1])`, `p([2],[2])` and `p([],[])`, the definition of naive reverse may be induced:

```
p([],[]).
p([H|T],L):-p(T,LT),append(LT,[H],L).
```

If a function-free hypothesis language is required, we will need to emulate the list functor as a predicate `cons/3` in our background theory, with the `cons(H,T,[H|T])` definition. The flattened (i.e., function-free) definition of naive reverse then becomes

```
p([],[]).
p(HT,L):-cons(H,T,HT),p(T,LT),cons(H,[],HL),append(LT,HL,L).
```

If we require a function-free language altogether, then we need to introduce *names* for complex terms such as `[1,2]`. The above examples would then also need to be flattened to `p(list12,list21)`, `p(list2,list2)`, and `p(null, null)`, and the `cons/3` predicate would be defined *extensionally* by an enumeration of ground facts such as `cons(1,list2,list12)` and `cons(2,null,list2)`.

The extensional definition of `cons/3` in Example 2.2 constitutes a set of background facts, whereas the definition of `append/3` remains intensional, even when flattened. Where appropriate, we will use the superscript i (e) to refer to intensional (extensional) background knowledge, i.e., $T = T^i \cup T^e$.

2.2.2 *Propositionalization.* Consider a simple learning task in which all predicates in the hypothesis language are unary, there are no function symbols, and hypothesis rules have a single universally quantified variable. In this case we can compile examples and background knowledge into a single extensional table, as follows. The table contains one column for each predicate in the hypothesis language, and one row for each example. Given an example, we set the value in the column for the foreground predicate to `true` if it is a positive example, and to `false` if it is a negative example. We then query the background knowledge to obtain the truth values in the other columns. The propositionalization approach will be further explained in Sections 3 and 4.

This transformation is similar to flattening in that it generates extensional background knowledge—however, it is different in that the new background facts *replace* the original background rules. This is so, because the table contains all the information needed to determine, for any example and any hypothesis rule, whether that rule correctly classifies the example. Moreover, the dataset contained in the table can be handled by any propositional or attribute-value learner, hence the name *propositionalization* for this transformation.[2]

We will have much more to say about propositionalization in the sections to come. For the moment, it suffices to make some general remarks. First of all, the condition that hypothesis rules are built from unary predicates with a single universally quantified variable is sufficient, but not necessary for propositionalization to be applicable. In general, the process is equally applicable if

[2]See Flach [1999] for a discussion why attribute-value representations are usually called "propositional."

the hypothesis language consists of definite rules with all variables occurring in the body also occurring in the head (so-called *constrained* rules), provided these rules are nonrecursive and stratifiable. It is possible to transform an unconstrained rule with existential variable(s) in the body into a constrained rule by introducing additional background predicates "hiding" the existential variables. This is the basis of the propositionalization method presented in Section 4.

Furthermore, the extensional dataset constructed by propositionalization need not be completely boolean. For instance, in a multiclass learning problem the head predicate would actually be a function, and the corresponding column would contain all possible class values. The same happens if some of the background predicates used in the hypothesis language are replaced by functions.

Finally, not all propositionalization approaches actually generate the extensional dataset. For instance, the first-order Bayesian classifier 1BC [Flach and Lachiche 1999] performs propositionalization only conceptually in deciding what first-order features (i.e., conjunctions of atoms) to generate, then evaluates those features directly on the first-order dataset. The main difference between attribute-value learning and ILP is that in attribute-value learning the attributes to be used in hypotheses are exactly those which are used to describe the examples, while the fundamental problem of ILP is to decide which features to use for constructing hypotheses. Propositionalization approaches use all features that can be constructed under a propositionalization transformation.

2.2.3 *Saturation.* For completeness, we note that the above process of "extensionalizing" intensional background knowledge is a transformation in its own right known as *saturation* [Rouveirol 1994] or *bottom-clause construction* [Muggleton 1995], which is also applicable outside the context of propositionalization. It can be understood as the construction of a ground clause $p(x_j) : -B^e(x_j)$, where $p(x_j)$ denotes the example (i.e., the foreground predicate p applied to the individual x_j), and $B^e(x_j)$ is a conjunction of all ground facts regarding x_j that can be derived from the background knowledge (here, it is usually assumed that the background knowledge is partitioned in parts describing individual examples, and a general part). In general, $B^e(x_j)$ will have a different format (and may even be infinite) for each individual x_j; only in propositionalization approaches they all have the same, fixed format. Notice, that if we saturate all examples this way, we obtain a "background-free" statement of the predictive ILP task as induction from ground clauses: find a hypothesis $H \subseteq \mathcal{H}_L$ such that $H \models p(x_j) : -B^e(x_j)$ for all j.

3. THE LINUS TRANSFORMATION METHOD

This section presents a method for effectively using background knowledge in learning both propositional and relational descriptions. The LINUS algorithm is a descendant of the learning algorithm used in QuMAS (Qualitative Model Acquisition System) which was used to learn functions of components of a qualitative model of the heart in the KARDIO expert system for diagnosing cardiac arrhythmias [Bratko et al. 1989]. The method, implemented in the system

LINUS, employs propositional learners in a more expressive logic programming framework.

3.1 Learning with LINUS

LINUS is an ILP learner which induces hypotheses in the form of constrained deductive hierarchical database (DHDB) clauses (a formal definition of DHDB and DDB clauses is introduced in Section 3.2). As input it takes training examples E, given as ground facts, and background knowledge T in the form of (possibly recursive) deductive database (DDB) clauses. The main idea of LINUS is to transform the problem of learning relational DHDB descriptions into an attribute-value learning task [Lavrač et al. 1991]. This is achieved by the so-called *DHDB interface*. The interface transforms the training examples from the DHDB form into the form of attribute-value tuples. This results in an extensional table, as introduced in Section 2.2.2. The most important feature of this interface is that by taking into account the types of the arguments of the target predicate, applications of background predicates and functions are considered as attributes for attribute-value learning.[3] Existing attribute-value learners can then be used to induce if-then rules. Finally, the induced if-then rules are transformed back into the form of DHDB clauses by the DHDB interface.

The LINUS algorithm can be viewed as a toolkit of learning techniques. These include a decision tree learner and two rule learners among which CN2 [Clark and Niblett 1989; Clark and Boswell 1991] has been used in our experiments. Recently, LINUS has been upgraded with an interface to MLC++ [Kohavi et al. 1996].

3.1.1 *The Algorithm.* The LINUS learning algorithm is outlined in Figure 1. Notice that the algorithm is given a type signature , defining the types of arguments of foreground and background predicates. In general, possible types include predefined atomic types such as integers, reals and booleans; user-defined enumerated types; tuple types (Cartesian products); constructor types; and predefined compound types like lists and sets.[4] Originally, LINUS could deal only with a typed Datalog representation, i.e., all types are atomic and either user-defined by enumerating its constants, or numerical. One could argue that this is not a real restriction, since compound types can be flattened as explained in Section 2.2.1. However, this means the type structure cannot be used as a declarative bias to guide learning. We overcome this limitation and define a declarative bias for compound type structures in Section 4.4.

[3]An application of a background predicate and function actually results in a call of a propositional feature (determined by the types of predicate/function arguments) obtained by instantiating the arguments of the background predicate/function by values defined by the training examples. If the call of a predicate succeeds, the call results in value `true`; otherwise the call returns value `false`. A call to a function returns the computed function value. Examples in Sections 3.1.2 and 3.1.3 illustrate the case for background predicates but not for functions.

[4]Not all of these types are available in every language—e.g., Prolog handles tuples and lists by means of constructors (functors), and sets can be seen as higher-order terms [Lloyd 1999].

Input. a set of examples E, a background theory T_B, and a type signature τ;
Output. an induced set of DHDB clauses H.

(1) From E, establish the training set of positive and negative ground facts $P \cup N$ of the target predicate p. (This step is required if only positive examples are given—in which case negative examples are constructed by closed-world assumption or near-miss modification—and can be skipped otherwise.)
(2) Transform the ILP problem to propositional form.
  (a) Determine the possible applications of background predicates B on the arguments of target predicate p, taking into account argument types from τ. Form a list of all literals to be used as features (attributes) for propositional learning.
  (b) Transform facts from the DHDB form into attribute-value tuples. For each example, the truth value of each of the propositional features is determined by calls to the background knowledge T_B.
(3) Induce a concept description by an attribute-value learner.
(4) Transform the induced attribute-value descriptions back into the form of DHDB clauses H.

Fig. 1. The LINUS transformation algorithm.

3.1.2 *A Sample Run of LINUS on a Chess Endgame Problem.* Let us illustrate LINUS on a simple example. Consider the problem of learning illegal positions on a chess board with only two pieces, white king and black king. The target predicate `illegal(WKf,WKr,BKf,BKr)` states that the position in which the white king is at `(WKf,WKr)` and the black king at `(BKf,BKr)` is illegal. Arguments `WKf` and `BKf` are of type `file` (with values `a` to `h`), while `WKr` and `BKr` are of type `rank` (with values `1` to `8`). Background knowledge is represented by two symmetric predicates `adjFile(X,Y)` and `adjRank(X,Y)`, which can be applied on arguments of type `file` and `rank`, respectively, and express that `X` and `Y` are adjacent. The built-in symmetric equality predicate `X=Y`, which works on arguments of the same type, may also be used in the induced clauses.

What follows is the description of the individual steps of the algorithm.

Step 1. First, the sets of positive and negative facts are established. In our domain, one positive example (illegal endgame position, labeled $\oplus$) and two negative examples (legal endgame positions, labeled $\ominus$) are given:

```
% illegal(WKf,WKr,BKf,BKr).
   illegal(a,6,a,7).   ⊕
   illegal(f,5,c,4).   ⊖
   illegal(b,7,b,3).   ⊖
```

Step 2. The given facts are then transformed into attribute-value tuples. To this end, the algorithm first determines the possible applications of the background predicates on the arguments of the target predicate, taking into account argument types. Each such application is considered as an attribute. In our example, the set of attributes determining the form of the tuples is the following:

⟨`WKf=BKf, WKr=BKr, adjFile(WKf,BKf), adjRank(WKr,BKr)`⟩

Table I. Propositional Form of the illegal Position Learning Problem

	Propositional features			
Class	WKf=BKf	WKr=BKr	adjFile(WKf,BKf)	adjRank(WKr,BKr)
true	true	false	false	true
false	false	false	false	true
false	true	false	false	false

Since predicates adjFile and adjRank are symmetric, their other two possible applications adjFile(BKf,WKf) and adjRank(BKr,WKr) are not considered as attributes for learning.[5]

The tuples, i.e., the values of the attributes, are generated by calling the corresponding predicates with argument values from the ground facts of predicate illegal. In this case, the attributes can take values true or false. For the given examples, the following tuples are generated:

```
% ⟨WKf=BKf, WKr=BKr, adjFile(WKf,BKf), adjRank(WKr,BKr)⟩
          ⟨ true, false, false, true ⟩   ⊕
          ⟨ false, false, false, true ⟩   ⊖
          ⟨ true, false, false, false ⟩   ⊖
```

These tuples are generalizations (relative to the given background knowledge) of the individual facts about the target predicate. This step is actually the propositionalization step described in Section 2.2.2; to make the connection with that section, note that the above tuples form a propositional table shown in Table I.

Step 3. Next, an attribute-value rule learner is used to induce a set of if-then rules from the above tuples:

```
Class = true if (WKf=BKf) = true ∧ adjRank(WKr,BKr) = true
```

Step 4. Finally, the induced if-then rules for Class = true are transformed back into DHDB clauses. In our example, we get the following clause:

```
illegal(WKf,WKr,BKf,BKr):-WKf=BKf, adjRank(WKr,BKr).
```

In summary, the learning problem is transformed from a relational to an attribute-value form and solved by an attribute-value learner. The induced hypothesis is then transformed back into the relational form.

3.1.3 *A Sample Run of LINUS on a Family Relations Problem.* The algorithm is illustrated also on a simple ILP problem of learning family relationships. The task is to define the target predicate daughter(X,Y), which states that person X is a daughter of person Y, in terms of the background predicates female, parent, and equality. All the variables are of type person,

[5]It should be noted that if the user so chooses the arguments of the target predicate—WKf, WKr, BKf and BKr—may be included in the selected set of attributes for learning (see Section 3.2.2, item 1 of Definition 3.4, allowing for a binding of a variable to a value to appear in clause body, e.g., WKf=a), so that the form of the tuples would be as follows: ⟨WKf, WKr, BKf, BKr, WKf=BKf, WKr=BKr, adjFile(WKf,BKf), adjRank(WKr,BKr)⟩.

Table II. A Simple Family Relationships Problem

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

Table III. Propositional Form of the daughter Relationship Problem

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

which is defined as person = {ann, eve, pat, sue, tom}. There are two positive and two negative examples of the target predicate. The training examples and the background predicates (excluding the built-in predicate equality) are given in Table II.

Step 1. Positive and negative examples are explicitly given, as shown in Table II.

Step 2. 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 predicate 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 III, where f stands for female, m for male, and p for parent. In this table, *variables* stand for the arguments of the target predicate, and *propositional features* denote the newly constructed attributes of the propositional learning task. When learning function-free DHDB clauses, only the new attributes are considered for learning. If we remove the function-free restriction, the arguments of the target predicate (named *variables* in Table III) are used as attributes in the propositional task as well (see Section 3.2.2 for a more detailed explanation).

Step 3. In the third step, an attribute-value learning program induces the following if-then rule from the tuples in Table III:

Class = true **if** female(X) = true ∧ parent(Y,X) = true

Step 4. In the last step, the induced if-then rules are transformed back into DHDB clauses. In our example, we get the following clause:

```
daughter(X,Y):-female(X),parent(Y,X).
```

Note that the same result can be obtained on a similar learning problem, where the target predicate daughter(X,Y) is to be defined in terms of the

Table IV. Intensional Background Knowledge for Learning the daughter Relationship

```
mother(eve,sue).   parent(X,Y):-     female(ann).
mother(ann,tom).      mother(X,Y).   female(sue).
father(pat,ann).   parent(X,Y):-     female(eve).
father(tom,sue).      father(X,Y).
```

background predicates female, male, and parent, given the background knowledge from Table IV. It illustrates that in this approach intensional background knowledge in the form of nonground clauses can be used in addition to ground facts.

3.1.4 *Discussion.* The chess endgame domain in Section 3.1.2 is a simplified example of an individual-centered domain: the individuals are endgame positions (i.e., 4-tuples of white king file and rank and black king file and rank), and the concept to be learned is a property of such positions. In the family domain (Section 3.1.3), however, there is no clear choice of individual. On the one hand, we could choose persons as individuals, but then we could only learn properties of persons, and not the daughter predicate as in the example. Taking pairs of individuals would solve this problem, but would lead to awkward definitions, as we would need background predicates to refer to the first or second person in a pair, or to swap persons in a pair as required by the daughter predicate:

```
daughter_pair(P):-first(P,X),female(X),swap(P,P1),parent_pair(P1).
```

The third alternative, taking whole family trees as individuals, suffers from the same problem as the first, namely that daughter is not a property of the individual.

More generally, one can draw a distinction between *concept learning*, on the one hand, and *program synthesis* on the other. Concept learning, and classification problems in general, is inherently individual-centered, as belonging to the concept or to a certain class is a property of the individual. Program synthesis, on the other hand, typically involves some calculation to determine the value of one argument of the target predicate, given the others. It can be trivially transformed into a boolean classification problem, by ignoring the modes and viewing it as a predicate rather than a function, but this ignores crucial information and makes the task even harder. The extended transformation method we present in Section 4 is especially suited for concept learning and classification tasks, but not for program synthesis tasks.

3.2 The LINUS Language Bias

As already mentioned in Section 3.1, LINUS induces hypotheses in the form of constrained deductive hierarchical database (DHDB) clauses. As input it takes training examples E, given as ground facts, and background knowledge T in the form of (possibly recursive) deductive database (DDB) clauses. Variables are typed.

Table V. Relating Database and Logic Programming Terminology

DDB terminology	**LP terminology**
relation name p	*predicate symbol p*
attribute of relation p	*argument of predicate p*
tuple $< a_1, \ldots, a_n >$	*ground fact* $p(a_1, \ldots, a_n)$
relation p – a set of tuples	*definition of predicate p – a set of ground facts*

This section first introduces the definitions of DHDB and DDB clauses in the database terminology and in the logic programming terminology, and continues by describing the actual LINUS language bias.

3.2.1 *The Terminology.* Deductive databases extend relational databases by allowing 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. Table V relates the basic deductive database [Ullman 1988] and logic programming [Lloyd 1987] terms.

Notice in Table V that a definition of a predicate is introduced as a set of ground facts. In general, of course, a *predicate definition* is a set of program clauses with the same predicate symbol (and arity) in their heads. A *program clause* is a clause of the form

$$L :- L_1, \ldots, L_m$$

where L is an atom, and each of $L_1, \ldots, L_m$ is a literal of the form L or `not` L, where L is an atom. A *normal program* is a set of program clauses.

The following definitions are adapted from [Lloyd 1987] and [Ullman 1988].

Definition 3.1 (*Deductive Database*). A *Datalog clause* is a program clause with no function symbols of nonzero arity (i.e., only variables and constants can be used as predicate arguments). A *constrained clause* is a program clause in which all variables in the body also appear in the head. A *database clause* is a typed program clause of the form

$$L :- L_1, \ldots, L_m$$

where L is an atom and $L_1, \ldots, L_m$ are literals. A *deductive database* (DDB) is a set of database clauses.

Database clauses use variables and function symbols in predicate arguments. As recursive predicate definitions are allowed, the language is substantially more expressive than the language of relational databases. If we restrict database clauses to be nonrecursive, we obtain the formalism of deductive hierarchical databases.

Definition 3.2 (*Deductive Hierarchical Database*). A *level mapping* of a database is a mapping from its set of predicates to the nonnegative integers; the value of a predicate under this mapping is called its *level*. A *deductive hierarchical database* (DHDB) is a deductive database which has a level mapping

such that for every database clause the level of its body predicates is less than the level of its head predicate.

While the expressive power of DHDB is the same as that of a relational database, DHDB allows intensional relations which can be much more compact than a relational database representation.

3.2.2 *The LINUS Language Bias of Constrained Clauses*

Definition 3.3 (*LINUS Background Knowledge*). There are two kinds of background predicates defined in the background knowledge T_B.

(1) *Utility functions* f_j are predicates with input/output mode declarations. When applied to ground input arguments from the training examples, utility functions compute the unique ground values of their output arguments.
(2) *Utility predicates* q_i have only *input* arguments and can be regarded as boolean utility functions having values `true` or `false` only.

Definition 3.4 (*LINUS Literals*). 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., $f_j(X, Y, Z), Z = a$, for X, Y being input, and Z being an output argument.

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 negated. 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.

Notice that the output argument of a utility function can only be used to form equalities and inequalities with constants. This restriction is alleviated in the DINUS algorithm (see Section 3.3).

Definition 3.5 (*LINUS Hypothesis Language*). The LINUS hypothesis language 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.

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 learner will use conditions of the form $A =$ `true` or $A =$ `false`

in the induced rules, where A is a feature (an attribute) (cf. examples in Sections 3.1.2 and 3.1.3). These are transcribed to literals A and `not` A, respectively, in the DHDB clauses. For case (1) an attribute-value learner will formulate 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 $f_j(X, Y, Z)$ has to be added to the DHDB clause so that the value of Z can be computed from the arguments of the target predicate.

3.3 DINUS: Extending LINUS to Learn Determinate Clauses

As explained in the previous section, LINUS cannot induce clauses with variables that occur in the body but not in the head. In this section we briefly outline a method that upgrades LINUS to learn a restricted class of clauses with determinate local variables (variables that occur only in the body, and that have only one possible binding given the bindings of the other variables).

We use the following definition of determinacy, adapted from [Džeroski et al. 1992].

Definition 3.6 (*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, for the given $E \cup T_B$, exactly one binding given the bindings of its variables that appear in preceding literals.

Determinate literals are a natural extension of the utility functions used in LINUS, in the sense that their output arguments are used as input arguments to arbitrary literals.

Example 3.7 (*Determinate Grandmother*). In the `grandmother(X,Y)` definition below, a new variable `Z` has been introduced in both clauses by a determinate literal (the value of variable `Z` is uniquely defined by the value of variable `Y`, since child `Y` has only one father and only one mother `Z`).

```
grandmother(X,Y):-father(Z,Y),mother(X,Z).
grandmother(X,Y):-mother(Z,Y),mother(X,Z).
```

The logically equivalent definition below of the predicate `grandmother` is in general nondeterminate.

Example 3.8 (*Nondeterminate Grandmother*). Exchanging the literals in the clauses for grandmother in Example 3.7 does not change their logical meaning.

```
grandmother(X,Y):-mother(X,Z),father(Z,Y).
grandmother(X,Y):-mother(X,Z),mother(Z,Y).
```

However, the resulting clauses may be nondeterminate since, for the given dataset, the value of variable `Z` is not necessarily uniquely defined by the value of variable `X`, since mother `X` can have more than one child `Z`.

While LINUS allows only global variables from the head of a clause to appear in the literals in clause body (constrained clauses), using determinacy allows

for a restricted form of local variables to be introduced in the body of an induced clause. This approach to learning of determinate clauses is implemented in the system DINUS [Džeroski et al. 1992; Lavrač and Džeroski 1992; 1994].

Compared to the LINUS algorithm outlined in Figure 1, the DINUS learning algorithm consists of the same main steps. Steps 2 and 4, outlined below, are more elaborate in DINUS.

—In step 2, the ILP problem is transformed to propositional form as follows.
  —The algorithm first constructs, from background predicates B, a list of determinate literals that introduce new local variables of depth at most i. These literals are not used for learning as they do not discriminate between positive and negative examples (due to determinacy, the new variables have a unique binding for each example, and the literals are evaluated `true` for all examples, positive and negative).
  —Next, it constructs, from background predicates B, a list of literals using as arguments only variables from the head literal and new determinate variables. These literals form the features (attributes) to be used for learning in step 3. For these features, for each example, their truth value is determined by calls to background knowledge T_B.
—In step 4, the induced propositional description is transformed back to a set of determinate DHDB clauses, by adding the necessary determinate literals which introduced the new variables.

A more detailed description of the algorithm and examples of its performance can be found in Lavrač and Džeroski [1992; 1994].

4. EXTENDING LINUS TO LEARN NONDETERMINATE CLAUSES

In this section we describe how the DINUS restriction of determinate literals can be overcome, by employing a so-called individual-centered representation. We start by discussing feature construction as an important, possible separate step, in rule construction. We continue by describing a benchmark ILP problem that is essentially nondeterminate, and show that by employing the proposed feature construction approach the LINUS hypothesis language can be extended to learning of nondeterminate DHDB clauses.

4.1 Motivation for Upgrading LINUS

The original LINUS algorithm, described in Section 3, cannot induce clauses with variables that occur in the body but not in the head. Its extension DINUS, described in Section 3.3, enables learning of a restricted class of DHDB clauses with determinate local variables (these have only one possible binding given the bindings of the other variables). The determinacy restriction is due to the nature of the algorithm: to be in line with the "nature" of attribute-value learning, it is natural that one training example is represented by one tuple (one feature vector). If DINUS were to allow for nondeterminacy, one training example would have got expanded to a number of tuples (their number being the product of numbers of possible bindings of new variables). Such an approach would enable handling multiple-instance problems [Dietterich et al.

1997], which would need to be accompanied by a modification of the covers relation for hypotheses (testing whether an example is satisfied by the hypothesis). Such an approach, as successfully proposed by Zucker and Ganascia [1996] (see Section 6 on related work), deviates from the basic "nature" of attribute-value learning. In our work we try to keep the "nature" of attribute-value learning by a transformation approach that results in keeping all the information inherent to one training example in one tuple, and using an arbitrary attribute-value learner to perform induction. Thus Section 4 shows how to extend LINUS to learning of nondeterminate DHDB clauses, without violating this principle of "natural" transformation to an attribute-value learning problem.

Why is it interesting to extend the LINUS propositionalization approach as opposed to using inductive logic programming systems that can do learning without transforming a learning problem to an attribute-value form? Some of the pros and cons are outlined below. The most obvious obvious pros are the much wider understanding and acceptance of attribute-value learning, and the variety of available algorithms, including public domain software such as WEKA [Witten and Frank 2000]. Whereas the main limitation is the inability of the transformation approaches to learn recursive hypotheses. There is also a tradeoff between constructing features in advance, as done in transformation approaches, and the advantageous "while learning" feature construction as done by ILP systems. Despite this advantageous feature of ILP, there are at least two disadvantages. The first concerns the use of negation. As shown in the results of Experiments 4.13 and 4.12 in Section 4.5, rules can be constructed using the negation in clause body, constructed features, and/or individual features. In ILP this is hard to be achieved. The second point relates to building of libraries of constructed features for similar types of tasks, whose utility is shown by Srinivasan and King's [1996] approach to predicate invention achieved by using a variety of predictive learning techniques to learn background knowledge predicate definitions (see Section 6 for more details).

A deeper understanding of the utility of transformation approaches to relational learning as compared to using ILP systems can be found in [De Raedt 1998] which provides an in-depth discussion of the relation between attribute-value learning and ILP.

4.2 Motivation for Feature Construction

Consider the task of rule construction. We assume that rules are implications consisting of a body (antecedent) and a head (consequent). A typical approach to rule construction is to select a head corresponding to a class, and then constructing an appropriate body. Not all rule learning algorithms follow this approach: for instance, CN2 [Clark and Niblett 1989] maintains a beam of best bodies, evaluated w.r.t. information gain (i.e., with maximal improvement in the distribution of classes among the covered examples), and then assigning a rule head (class assignment) maximizing classification accuracy. As another example, association rule learning algorithms, e.g., APRIORI [Agrawal et al. 1996], first search for the most promising bodies (frequent itemsets), and then construct association rules from two frequent itemsets such that one

includes the other. In these two cases, body construction is the main step of rule construction.

Usually, a rule body is a conjunction of (possibly negated) literals, where a literal can be an attribute-value pair or a Prolog literal. Typically, such conjunctions are constructed literal by literal. For instance, FOIL [Quinlan 1990] selects the literal with maximal information gain. It is well-known that this leads to problems with literals introducing new local variables, since these literals do not improve the class distribution unless they are combined with another literal which consumes the variable. There exist various attempts to solve this problem, e.g., look-ahead search which searches for more than one literal at a time. However, this approaches the problem in an ad-hoc fashion. The main issue is that in ILP it is unnatural to separate literals which share variables other than the one(s) occurring in the head of the rule; they should be added as one chunk. Such chunks of related literals are what we call *first-order features*.

Defining a rule body as consisting of features, where a feature is a conjunction of literals, also has the advantage of added expressiveness if we allow features to be negated. This would have the effect of allowing disjunctions of negated literals in the body. It also illustrates that body bias and feature bias are really two different things: e.g., we can allow negation of features but not of literals within features, or vice versa. The added expressiveness can be considerable. For instance, suppose that we define a body as a conjunction of possibly negated features, and a feature as a conjunction of possibly negated literals. The effect would be that any boolean combination of literals would be expressible in this combined body-feature bias.

How can features be constructed? A natural choice—to which we will restrict ourselves in this paper—is to define a feature as a conjunction of (possibly negated) literals. Features describe subsets of the training set that are for some reason unusual or interesting. For instance, the class distribution among the instances described by the feature may be different from the class distribution over the complete training set in a statistically significant way. Alternatively, the feature may simply be shared by a sufficiently large fraction of the training set. In the first case, the feature is said to describe an *interesting subgroup* of the data, and several propositional and first-order systems exist that can discover such subgroups (e.g., Explora [Klösgen 1996], MIDOS [Wrobel 1997], and Tertius [Flach and Lachiche 2001]). In the second case, the feature is said to describe a *frequent itemset*, and again several algorithms and systems exist to discover frequent itemsets (e.g., APRIORI [Agrawal et al. 1996] and WARMR [Dehaspe and Toivonen 1999]). Notice that the systems just mentioned are all discovery systems, which perform descriptive rather than predictive induction. Indeed, feature construction is a discovery task rather than a classification task.

In this section we introduce a simple learning problem that will be used to illustrate different notions introduced in this paper. The learning task is to discover low size-complexity Prolog programs for classifying trains as Eastbound or Westbound [Michie et al. 1994]. The problem is illustrated in Figure 2. We illustrate two possible representations of this learning problem in Prolog.

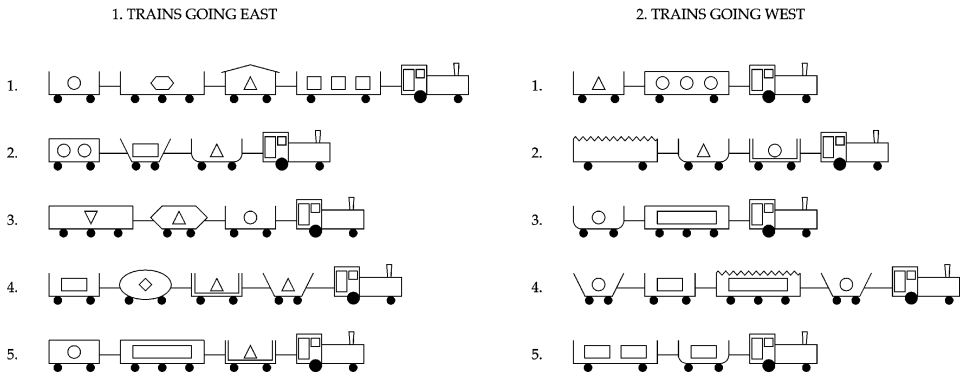

Fig. 2. The 10-train East-West challenge.

Example 4.1 (*East-West Challenge in Prolog, Nonflattened*). Using a term-based (i.e., nonflattened) representation, the first train in Figure 2 can be represented as follows:

```
eastbound([c(rect,short,single,no,  2,l(circ,1)),
           c(rect,long, single,no,  3,l(hexa,1)),
           c(rect,short,single,peak,2,l(tria,1)),
           c(rect,long, single,no,  2,l(rect,3))], true).
```

Here, a train is represented as a list of cars; a car is represented as a six-tuple indicating its shape, length, whether it has a double wall or not, its roof shape, number of wheels, and its load; finally, a load is represented by a pair indicating the shape of the load and the number of objects. A possible inductive hypothesis, stating that a train is eastbound if it has a short open car, is as follows:

```
eastbound(T,true):-member(T,C),arg(2,C,short),arg(4,C,no).
```

Example 4.2 (*East-West Challenge in Prolog, Flattened*). A flattened representation of the same data, using function-free ground facts, is as follows:

```
eastbound(t1,true).

hasCar(t1,c11).                 hasCar(t1,c12).
cshape(c11,rect).               cshape(c12,rect).
clength(c11,short).             clength(c12,long).
cwall(c11,single).              cwall(c12,single).
croof(c11,no).                  croof(c12,no).
cwheels(c11,2).                 cwheels(c12,3).
hasLoad(c11,l11).               hasLoad(c12,l12).
lshape(l11,circ).               lshape(l12,hexa).
lnumber(l11,1).                 lnumber(l12,1).

hasCar(t1,c13).                 hasCar(t1,c14).
cshape(c13,rect).               cshape(c14,rect).
clength(c13,short).             clength(c14,long).
```

```
cwall(c13,single).          cwall(c14,single).
croof(c13,peak).            croof(c14,no).
cwheels(c13,2).             cwheels(c14,2).
hasLoad(c13,l13).           hasLoad(c14,l14).
lshape(l13,tria).           lshape(l14,rect).
lnumber(l13,1).             lnumber(l14,3).
```

Using this representation, the above hypothesis would be written as

```
eastbound(T,true):-hasCar(T,C),clength(C,short),croof(C,no).
```

Strictly speaking, the two representations in Examples 4.1 and 4.2 are not equivalent, since the order of the cars in the list in the first representation is disregarded in the second. This could be fixed by using the predicates `hasFirstCar(T,C)` and `nextCar(C1,C2)` instead of `hasCar(T,C)`. For the moment, however, we stick to the above flattened representation; essentially, this means that we interpret a train as a *set* of cars, rather than a list. Under this assumption, and assuming that each car and each load is uniquely named, the two representations are equivalent. As a consequence, the two hypothesis representations are isomorphic: `hasCar(T,C)` corresponds to `member(T,C)`; `clength(C,short)` corresponds to `arg(2,C,short)`; and `croof(C,no)` corresponds to `arg(4,C,no)`. This isomorphism between flattened and nonflattened hypothesis languages is a feature of what we call *individual-centered* representations [Flach 1999].

4.3 Identification of First-Order Features

In attribute-value learning, features only add expressiveness if the body bias allows negation of features. However, in first-order learning features occupy a central position. The distinguishing characteristic of first-order learning is the use of variables that are shared between some but not all literals. If there are no such variables, i.e., all variables occur in all literals, then the only function of variables is to distinguish literals occurring in rules from literals occurring in examples. If this distinction does not need to be represented syntactically but is maintained by the learning algorithm (the Single Representation Trick), the variables can be dispensed with altogether, thus reducing the rule to a propositional one.

Definition 4.3 (*Global and Local Variables*). The variables occurring in the head of the rule are called *global variables*. Variables occurring only in the body are called *local variables*. A rule in which there are no local variables is called *constrained*; a constrained rule in which every global variable occurs once in every literal is called *semipropositional*.

Global variables are assumed to be universally quantified, and the scope of the universal quantifier is the rule. Local variables, called *existential variables* in logic programming terminology, are existentially quantified, and the scope of the existential quantifier is the rule body.

The key idea of first-order features is to restrict all interaction between local variables to literals occurring in the same feature. This is not really a

restriction, as in some cases the whole body constitutes a single feature. However, often the body of a rule can be partitioned into separate parts which only share global variables.

Example 4.4 (*Single vs. Multiple Features*). Consider the following Prolog rule, stating that a train is eastbound if it contains a short car and a closed car:

```
eastbound(T,true):-hasCar(T,C1),clength(C1,short),
                   hasCar(T,C2),not croof(C2,no).
```

The body of this clause consists of two features: `hasCar(T,C1),clength(C1, short)` or "has a short car" and `hasCar(T,C2),not croof(C2,no)` or "has a closed car." In contrast, the following rule

```
eastbound(T,true):-hasCar(T,C),clength(C,short),not croof(C,no).
```

contains a single feature expressing the property "has a short closed car."

It is easy to recognize the first-order features in any given rule, by focusing on the use of local variables.

Definition 4.5 (*First-Order Features*). For any two body literals L_1 and L_2 of a given rule, L_1 and L_2 belong to the same equivalence class, $L_1 \sim_{lv} L_2$ iff they share a local variable. Clearly, $\sim_{lv}$ is reflexive and symmetric, and hence its transitive closure $=_{lv}$ is an equivalence relation inducing a partition on the body. The conjunction of literals in an equivalence class is called a *first-order feature*.[6]

Example 4.6 (*First-Order Features*). Consider the following clause:

```
eastbound(T,true):-hasCar(T,C),hasLoad(C,L),lshape(L,tria).
```

In this clause the entire rule body is a single first-order feature: $\exists$ `C,L: hasCar(T,C)` $\wedge$ `hasLoad(C,L)` $\wedge$ `lshape(L,tria)`. This feature is true of any train which has a car which has a triangular load.

We note the following simple result, which will be used later.

PROPOSITION 4.7 (PROPOSITIONALIZING RULES). *Let R be a Prolog rule, and let R' be constructed as follows. Replace each feature F in the body of R by a literal L consisting of a new predicate with R's global variable(s) as argument(s), and add a rule $L: -F$. R' together with the newly constructed rules is equivalent to R, in the sense that they have the same success set.*

By construction, R' is a semipropositional rule.

Example 4.8 (*Propositionalizing Rules*). Consider again the following Prolog rule R:

```
eastbound(T,true):-hasCar(T,C1),clength(C1,short),
                   hasCar(T,C2),not croof(C2,no).
```

[6]The concept of first-order feature as defined here has been developed in collaboration with Nicolas Lachiche.

By introducing the following background rules

```
hasShortCar(T):-hasCar(T,C),clength(C,short).
hasClosedCar(T):-hasCar(T,C),not croof(C,no).
```

we can reexpress R as the following semipropositional rule R':

```
eastbound(T,true):-hasShortCar(T),hasClosedCar(T).
```

Notice that R' contains only global variables, and therefore is essentially a propositional rule. R's first-order features have been confined to the background theory. Thus, provided we have a way to construct the necessary first-order features, *body construction in ILP is essentially propositional*.

By recognizing the above property, an ILP problem can be transformed into a propositional learning problem, provided that appropriate first-order features can be constructed.

4.4 A Declarative Bias for First-Order Feature Construction

Definition 4.5 suggests how to recognize features in a given clause, but it does not impose any restrictions on possible features and hence cannot be used as a declarative feature bias. In this section we define such a feature bias, following the term-based individual-centered representations introduced by Flach et al. [1998] and further developed by Flach and Lachiche [1999]. Such representations collect all information about one individual in a single term, e.g., a list of 6-tuples as in the train representation of Example 4.1. Rules are formed by stating conditions on the whole term, e.g., `length(T,4)`, or by referring to one or more subterms and stating conditions on those subterms. Predicates which refer to subterms are called *structural predicates*: they come with the type of the term, e.g., list membership comes with lists, projections (n different ones) come with n-tuples, etc. Notice that projections are determinate, while list membership is not. In fact, the only place where nondeterminacy can occur in individual-centered representations is in structural predicates.

Individual-centered representations can also occur in flattened form. In this case each of the individuals and most of its parts are named by constants, as in Example 4.2. It is still helpful to think of the flattened representation to be obtained from the term-based representation. Thus, `hasCar` corresponds to list/set membership and is nondeterminate (i.e., one-to-many), while `hasLoad` corresponds to projection onto the sixth component of a tuple and thus is determinate (i.e., one-to-one). Keeping this correspondence in mind, the definitions below for the nonflattened case can be easily translated to the flattened case.

We assume a given type structure defining the type of the individual. In what follows, "atomic type" refers to a type with atomic values (booleans, integers, characters, etc.); "compound type" refers to a type whose values have structure (lists, records, strings, etc.); and "component type" refers to one of the types making up a compound type (e.g., record is a component type of list of records).

Definition 4.9 (*Structural Predicates*). Let be a given type signature, defining a single top-level type in terms of component types. A *structural predicate* is a binary predicate associated with a compound type in representing

the mapping between that type and one of its component types. A *functional* structural predicate, or *structural function*, maps to a unique subterm, while a *nondeterminate* structural predicate is nonfunctional.

In general, we have a structural predicate or function associated with each compound type in . In addition, we have *utility predicates* as in LINUS (called *properties* in [Flach and Lachiche 1999]) associated with each atomic component type, and possibly also with compound component types and with the top-level type (e.g., the class predicate). Utility predicates differ from structural predicates in that they do not introduce new variables.

Example 4.10 (*Structural and Utility Predicates*). For the East-West challenge we use the following type signature. `train` is declared as the top-level set type representing an individual. The structural predicate `hasCar` nondeterministically selects a car from a train. `car` is defined as a 6-tuple. The first 5 components of each 6-tuple are atomic values, while the last component is a 2-tuple representing the load, selected by the structural function `hasLoad`. In addition, the type signature defines the following utility predicates (properties): `eastbound` is a property of the top-level type `train`; the following utility predicates are used on the component type `car`: `cshape`, `clength`, `cwall`, `croof`, and `cwheels`; and `lshape` and `lnumber` are properties of the (second level) component type `load`.

To summarize, structural predicates refer to parts of individuals (these are binary predicates representing a link between a compound type and one of its components; they are used to introduce new local variables into rules), whereas utility predicates present properties of individuals or their parts, represented by variables introduced so far (they do not introduce new variables). The language bias expressed by mode declarations used in other ILP learners such as Progol [Muggleton 1995] or WARMR [Dehaspe and Toivonen 1999] partly achieves the same goal by indicating which of the predicate arguments are input (denoting a variable already occurring in the hypothesis currently being constructed) and which are output arguments, possibly introducing a new local variable. However, mode declarations constitute a body bias rather than a feature bias.

Declarations of types, structural predicates, and utility predicates define the feature bias. The actual first-order feature construction will be restricted by parameters that define the maximum number of literals constituting a feature, maximal number of variables, and the number of occurrences of individual predicates.

Definition 4.11 (*First-Order Feature Construction*). A *first-order feature* of an individual is constructed as a conjunction of structural predicates and utility predicates which is well-typed according to . Furthermore

(1) there is exactly one variable with type , which is free (i.e., not quantified) and which will play the role of the global variable in rules;
(2) each structural predicate introduces a new existentially quantified local variable, and uses either the global variable or one of the local variables introduced by other structural predicates;

(3) utility predicates do not introduce new variables (this typically means that one of their arguments is required to be instantiated);

(4) all variables are used either by a structural predicate or a utility predicate.

The following first-order feature could be constructed in the above feature bias, allowing for 4 literals and 3 variables:

```
hasCar(T,C),hasLoad(C,L),lshape(L,tria)
```

4.5 Using First-Order Features in LINUS

In the previous sections we have argued that feature construction is a crucial notion in inductive rule learning. We have given precise definitions of features in first-order languages such as Prolog. First-order features bound the scope of local variables, and hence constructing bodies from features is essentially a propositional process that can be solved by a propositional rule learner such as CN2. In this section we show the usefulness of this approach by solving two nondeterminate ILP tasks with the transformation-based rule learner LINUS [Lavrač and Džeroski 1994].

We provide LINUS with features defining background predicates, as suggested by Proposition 4.7. For instance, in the trains example we add clauses of the following form to the background knowledge:

```
train42(T):-hasCar(T,C),hasLoad(C,L),lshape(L,tria).
```

LINUS would then use the literal `train42(T)` in its hypotheses. Such literals represent propositional properties of the individual.

In the two experiments reported on in this section we simply provide LINUS with all features that can be generated within a given feature bias (recall that such a feature bias includes bounds on the number of literals and first-order features).

Experiment 4.12 (*LINUS Applied to the East-West Challenge*). We ran LINUS on the 10 trains in Figure 2, using a nondeterminate background theory consisting of all 190 first-order features with up to two utility predicates and up to two local variables. Using CN2, the following rules were found (to improve readability, we have expanded the features used in the rules):

```
eastbound(T,true):-
  hasCar(T,C1),hasLoad(C1,L1),lshape(L1,tria),lnumber(L1,1),
  not (hasCar(T,C2),clength(C2,long),croof(C2,jagged)),
  not (hasCar(T,C3),hasLoad(C3,L3),clength(C3,long),lshape(L3,circ)).
eastbound(T,false):-
  not (hasCar(T,C1),cshape(C1,ellipse)),
  not (hasCar(T,C2),clength(C2,short),croof(C2,flat)),
  not (hasCar(T,C3),croof(C3,peak),cwheels(C3,2)).
```

These rules were found allowing negation in the body, but not within features. If negation is also allowed within the feature bias, the following simple rules are induced:

```
eastbound(T,true):-
  hasCar(T,C),clength(C,short),not croof(C,no).
eastbound(T,false):-
  not (hasCar(T,C),clength(C,short),not croof(C,no)).
```

That is, a train is eastbound if and only if it has a short closed car.

The mutagenesis learning task [Muggleton et al. 1998] concerns predicting which molecular compounds cause DNA mutations. The mutagenesis dataset consists of 230 classified molecules; 188 of these have been found to be amenable to regression modeling, and the remaining 42, to which we restrict attention here, as "regression-unfriendly." The dataset furthermore includes two handcrafted indicator attributes I_1 and I_a to introduce some degree of structural detail into the regression equation; following some experiments in [Muggleton et al. 1998] we did not include these indicators.

Experiment 4.13 (*LINUS Applied to Mutagenesis*). We ran LINUS on the 42 regression-unfriendly molecules, using a nondeterminate background theory consisting of all 57 first-order features with one utility literal concerning atoms (i.e., discarding bond information). Using CN2, the following rules were found:

```
mutag(M,false):-not (has_atom(M,A),atom_type(A,21)),
                 logP(M,L),between(1.99,L,5.64).
mutag(M,false):-not (has_atom(M,A),atom_type(A,195)),
                 lumo(M,Lu),between(-1.74,Lu,-0.83),
                 logP(M,L),L>1.81.
mutag(M,false):-lumo(M,Lu),Lu>-0.77.

mutag(M,true):-has_atom(M,A),atom_type(A,21),
               lumo(M,Lu),Lu<-1.21.
mutag(M,true):-logP(M,L),between(5.64,L,6.36).
mutag(M,true):-lumo(M,Lu),Lu>-0.95,
               logP(M,L),L<2.21.
```

Three out of six clauses contain first-order features. Notice how two of these concern the same first-order feature "having an atom of type 21"—incidentally, such an atom also features in the (single) rule found by Progol on the same dataset. Running CN2 with only the lumo and logP attributes produced eight rules; thus, this experiment suggests that first-order features can enhance the understandability of learned rules. Furthermore, we also achieved higher predictive accuracy: 83% with first-order features (as opposed to 76% using only lumo and logP). This accuracy is the same as achieved by Progol, having access to bond information and further structural background knowledge [Muggleton et al. 1998].

5. IRRELEVANT LITERAL ELIMINATION

In the two experiments reported on in the previous section we provided LINUS with all features that can be generated within a given feature bias.

Alternatively, we can use a descriptive learner such as Tertius [Flach and Lachiche 2001] to generate only features that correlate sufficiently with the class attribute, or we can apply a relevancy filter [Lavrač et al. 1998] to eliminate irrelevant features from the set of exhaustively generated features, as shown in this section.

Some features defined by the hypothesis language bias may be irrelevant for the given learning task. This section shows that irrelevant features can be detected and eliminated in preprocessing. Besides reducing the hypothesis space and facilitating the search for the solution, the elimination of irrelevant features may contribute to a better understanding of the problem domain. For example, this may be important in data analysis where irrelevant features may indicate that some measurements are not needed.

The problem of relevance was addressed in early research on inductive concept learning [Michalski 1983]. Basically one can say that all learners are concerned with the selection of "good" features to be used to construct the hypothesis. This problem has attracted much attention in the context of feature selection in attribute-value learning [Caruana and Freitag 1994; John et al. 1994; Skalak 1994].

We give definitions of irrelevant features and outline an algorithm that enables their elimination (for a detailed study of relevance see Lavrač et al. [1999]).

5.1 The p/n Pairs of Examples and Relevance of Features

Consider a two-class learning problem where the training set E consists of positive and negative examples of a concept ($E = P \cup N$), and where examples $e \in E$ are tuples of truth-values of features. The set of all features in the given language bias is denoted by L.

Assume that the training set E is represented as a table where rows correspond to training examples and where columns correspond to (positive and negated) features L. An element in the table has the value `true` when the example satisfies the condition (feature) in the column of the table; otherwise its value is `false`. The table is divided in two parts, P and N, where P are the positive examples, and N are the negative examples. Let $P \cup N$ denote the truth-value table E.

Let us introduce the following definitions and notation:

Definition 5.1 (*p/n Pairs of Examples*). Let $E = P \cup N$, where P are positive and N are negative examples. A p/n pair is a pair of training examples where $p \in P$ and $n \in N$.

Definition 5.2 (*Coverage of p/n Pairs*). Let L denote a set of features. A feature $l \in L$ covers a p_i/n_j pair if the feature has value `true` for p_i and value `false` for n_j.

Notice that in the standard machine learning terminology we may reformulate the definition of coverage of p/n pairs as follows: feature l covers a p/n pair if l covers (has value `true` for) the positive example p and does not cover (has value `false` for) the negative example n.

The notion of p/n pairs can be used to prove important properties of features for building complete and consistent concept descriptions. Assuming that L is rich enough to allow for a complete and consistent hypothesis H to be induced from the set of training examples E, the following result can be proved [Lavrač et al. 1999].

PROPOSITION 5.3. *Let $L' \subseteq L$. A complete and consistent hypothesis H can be found using only features from the set L' if and only if for each possible p/n pair from the training set E there exists at least one feature $l \in L'$ that covers the p/n pair.*

This proposition points out that when deciding about the relevance of features it will be significant to detect which p/n pairs are covered by the feature. Second, it suggests to directly detect useless features as those that do not cover any p/n pair. In addition, an important property of pairs of features can now be defined: the property of the so-called coverage of features.

Definition 5.4 (*Coverage of Features*). Let $l \in L$. Let $E(l)$ denote the set of all p/n pairs covered by feature l. Feature l covers feature l' if $E(l') \subseteq E(l)$.

Definition 5.5 (*Irrelevance of Features*). Feature $l' \in L$ is irrelevant if there exists another feature $l \in L$ such that l covers l' ($E(l') \subseteq E(l)$).

5.2 The Relevancy Filter

It can be shown that if a feature $l' \in L$ is irrelevant then for every complete and consistent hypothesis $H = H(E, L)$ (built from example set E and feature set L) whose description includes feature l', there exists a complete and consistent hypothesis $H' = H(E, L')$, built from the feature set $L' = L \backslash \{l'\}$ that excludes l' [Lavrač et al. 1999]. This theorem is the basis of an irrelevant feature elimination algorithm, outlined below.

The relevancy filter, called REDUCE, was initially developed and implemented in the ILLM (Inductive Learning by Logic Minimization) algorithm for inductive concept learning [Gamberger 1995]. The REDUCE algorithm [Lavrač et al. 1998] first eliminates all the useless features l_i (features that are `false` for all $p \in P$ and features that are `true` for all $n \in N$) and continues by the elimination of other irrelevant features in turn. The algorithm for filtering of irrelevant features is given in Figure 3.

Note that usually the term feature is used to denote a positive literal (or a conjunction of positive literals; let us, for the simplicity of the arguments below, assume that a feature is a single positive literal). In the hypothesis language, the existence of one feature implies the existence of two complementary literals: a positive and a negated literal. Since each feature implies the existence of two literals, the necessary and sufficient condition for a feature to be eliminated as irrelevant is that both of its literals are irrelevant.

This observation directly implies the procedure taken in our experiments. First we convert the starting feature vector to the corresponding literal vector which has twice as many elements. After that, we eliminate the irrelevant literals and, in the third step, we construct the reduced set of features which

Input. tables of positive examples P and negative examples N, and a set of features L;
Output. reduced tables of positive examples P' and negative examples N', and a reduced set of features L'.

$P' \leftarrow P$, $N' \leftarrow N$, $L' \leftarrow L$
for $\forall\, l_i \in L'$, $i \in [1, |L|]$ **do**
  if l_i has value `false` for all rows of P' **then**
    eliminate l_i from L', and
    eliminate column l_i from P' and N' tables
  if l_i has value `true` for all rows of N' **then**
    eliminate l_i from L', and
    eliminate column l_i from P' and N' tables
  if l_i is covered by any $l_j \in L'$ **then**
    eliminate l_i from L', and
    eliminate column l_i from P' and N' tables
endfor

Fig. 3. An algorithm for irrelevant feature elimination.

includes all the features which have at least one of their literals in the reduced literal vector.

It must be noted that direct detection of irrelevant features (without conversion to and from the literal form) is not possible except in the trivial case where two (or more) features have identical values for all training examples. Only in this case a feature f exists whose literals f and $\neg f$ cover both literals g and $\neg g$ of some other feature. In a general case if a literal of feature f covers some literal of feature g then the other literal of feature g is not covered by the other literal of feature f. But it can happen that this other literal of feature g is covered by a literal of some other feature h. This means that although there is no such feature f that covers both literals of feature g, feature g can still turn out to be irrelevant.

5.3 Results of Experiments

The objective of the experiments was to show the utility of the feature elimination algorithm REDUCE in the context of feature construction. The idea of the proposed approach is to allow the generation of a large number of different features, which will surely include all significant ones, and then, prior to the use of an inductive learner, eliminate all irrelevant features in order to keep the computation as effective as possible. Experiments were performed for two different feature biases.

Experiment 5.6 (*The Trains Example with 190 Features*). In this experiment, 190 features were generated with up to two local variables and up to two utility predicates. In order to apply the REDUCE algorithm we first converted the starting feature vector of 190 features to the corresponding feature vector which has twice as many elements, containing 190 initial features (positive features) as well as their negated counterparts (190 negated features). After that, we applied the relevancy filter and, in the third phase, constructed

the reduced set of features which includes all the features which have at least one of their positive or negated feature in the reduced feature set.

Irrelevant feature elimination resulted in eliminating 174 features as irrelevant, keeping the following 16 features in the relevant feature set:

```
hasCar(T,C),cshape(C,u_shaped)
hasCar(T,C),clength(C,long)
hasCar(T,C),croof(C,jagged)
hasCar(T,C),croof(C,peak)
hasCar(T,C),cwheels(C,3)
hasCar(T,C),hasLoad(C,L),lshape(L,rect)
hasCar(T,C),hasLoad(C,L),lshape(L,tria)
hasCar(T,C),clength(C,long),cwheels(C,2)
hasCar(T,C),clength(C,short),croof(C,flat)
hasCar(T,C),hasLoad(C,L),cshape(C,rect),lshape(L,tria)
hasCar(T,C),hasLoad(C,L),clength(C,long),lshape(L,circ)
hasCar(T,C),hasLoad(C,L),clength(C,short),lshape(L,circ)
hasCar(T,C),hasLoad(C,L),cwall(C,double),lshape(L,tria)
hasCar(T,C),hasLoad(C,L),cwall(C,single),lshape(L,circ)
hasCar(T,C),hasLoad(C,L),croof(C,no),lshape(L,rect)
hasCar(T,C),hasLoad(C,L),croof(C,no),lshape(L,circ)
```

By using only these 16 features for learning, CN2 induced the following rules:

```
eastbound(T,false):-
  not (hasCar(T,C1),croof(C1,peak)),
  not (hasCar(T,C2),clength(C2,short),croof(C2,flat)),
  not (hasCar(T,C3),hasLoad(C3,L3),cwall(C3,double),lshape(L3,tria)).
eastbound(T,true):-
  not (hasCar(T,C4),croof(C4,jagged)),
  hasCar(T,C5),hasLoad(C5,L5),lshape(L5,tria),
  not (hasCar(T,C6),hasLoad(C6,L6),clength(C6,long),lshape(L6,circ)).
```

Experiment 5.7 (*The Trains Example with 564 Features*). This experiment was repeated on an extended set of 564 features, containing up to two local variables and up to three utility predicates. The same methodology of feature elimination was used, and the resulting relevant feature set again had only 16 relevant features:

```
hasCar(T,C),clength(C,long)
hasCar(T,C),croof(C,jagged)
hasCar(T,C),croof(C,peak)
hasCar(T,C),hasLoad(C,L),lshape(L,rect)
hasCar(T,C),hasLoad(C,L),lshape(L,tria)
hasCar(T,C),clength(C,long),cwheels(C,2)
hasCar(T,C),clength(C,short),croof(C,flat)
hasCar(T,C),hasLoad(C,L),cshape(C,rect),lshape(L,tria)
hasCar(T,C),hasLoad(C,L),clength(C,long),lshape(L,circ)
hasCar(T,C),hasLoad(C,L),cwall(C,double),lshape(L,tria)
```

```
hasCar(T,C),hasLoad(C,L),croof(C,no),lshape(L,rect)
hasCar(T,C),hasLoad(C,L),croof(C,no),lshape(L,circ)
hasCar(T,C),cshape(C,rect),clength(C,short),cwall(C,single)
hasCar(T,C),hasLoad(C,L),cshape(C,rect),clength(C,short),
   lshape(L,circ)
hasCar(T,C),hasLoad(C,L),cshape(C,rect),cwall(C,single),
   lshape(L,circ)
hasCar(T,C),hasLoad(C,L),clength(C,short),cwall(C,single),
   lshape(L,circ)
```

The last four features are outside the feature bias used in the previous experiment, as they contain three utility predicates.

Using these features, the following rules were induced:

```
eastbound(T,false):-
  hasCar(T,C1),clength(C1,long),
  not (hasCar(T,C2),croof(C2,peak)),
  not (hasCar(T,C3),clength(C3,short),croof(C3,flat)).
eastbound(T,true):-
  not (hasCar(T,C4),croof(C4,jagged)),
  hasCar(T,C5),hasLoad(C5,L5),lshape(L5,tria),
  not (hasCar(T,C6),hasLoad(C6,L6),clength(C6,long),lshape(L6,circ)).
```

Only the rule for negative examples got changed. Notice that none of the features with three utility predicates appears in these rules (i.e., they could have been learned in the previous experiment as well).

6. RELATED WORK

In propositional learning, the idea of augmenting an existing set of attributes with new ones is known under the term *constructive induction*. The problem of feature construction has been studied extensively (see, for instance, Pagallo and Haussler [1990], Wnek and Michalski [1991], Oliveira and Sangiovanni-Vincentelli [1992], and Koller and Sahami [1996]). A first-order counterpart of constructive induction is *predicate invention* (see Stahl [1996] for an overview of predicate invention in ILP).

The work presented in this paper is in the mid-way: we perform a simple form of predicate invention through first-order feature construction, and use the constructed features for propositional learning.

6.1 Closely Related Approaches

Closely related approaches to the propositionalization of relational learning problems include Turney's RL-ICET algorithm [Turney 1996], Kramer et al.'s [1998] stochastic predicate invention approach, and Srinivasan and King's [1996] approach to predicate invention achieved by using a variety of predictive learning techniques to learn background knowledge predicate definitions. These approaches are described in some more detail below.

In the East-West challenge problem, Turney's RL-ICET algorithm achieved one of the best results [Turney 1996]. Its success was due to exhaustive feature

construction, introducing new propositional features for all combinations of up to three Prolog literals. To minimize size complexity, costs of features were computed and taken into account by a cost-sensitive decision tree induction algorithm ICET.

In previous work [Lavrač et al. 1998] we reported on the adaptation of Turney's approach to LINUS, resulting in 1066-long binary feature vectors used for learning by a propositional learning algorithm. Using the relevancy filter presented in Section 5 resulted in significant feature reduction to only 86 relevant features, which enabled more effective learning of low-cost hypotheses.

The approach used by Kramer et al. [1998] can be viewed as an extension of Turney's approach. It is much more sophisticated due to the following main reasons:

—predicate invention (instead of feature construction);
—stochastic search of best features of variable length (instead of an exhaustive propositional feature construction for all combinations of up to three literals);
—a fitness function optimizing description length (similar to Turney's costs of features) as well as other criteria (generality/specificity and variability) used to guide the search of best features (fitness used in feature construction as opposed to using all features by a cost-sensitive learning algorithm).

Like Kramer, Srinivasan and King [1996] perform propositionalization through predicate invention, which is, however, performed through hypothesis generation by using a variety of techniques, including

—statistical techniques (e.g., linear regression);
—propositional learning (e.g., rule and decision tree learning);
—inductive logic programming (e.g., learning hypotheses by P-Progol).

This approach was initially investigated in early experiments by GOLEM [Muggleton and Feng 1990] in the protein secondary structure prediction problem [Muggleton et al. 1992], where the same algorithm GOLEM was used on the same dataset in several runs to generate additional predicates to be considered in learning. This approach was followed by Mizoguchi et al. [1996], and finally developed into a sophisticated data mining methodology by Srinivasan and King. The recently developed "repeat learning framework" [Khan et al. 1998] further elaborates this approach.

6.2 Other Related Approaches

The line of research related to LINUS is reported by Zucker and Ganascia [1996; 1998] Fensel et al. [1995], Geibel and Wysotzki [1996], Sebag and Rouveirol [1997], Cohen [1996], and others.

Zucker and Ganascia [1996; 1998] proposed to decompose structured examples into several learning examples, which are descriptions of parts of what they call the "natural example." The transformation results in a table with multiple rows corresponding to a single example. Note that a problem which is reformulated in this way is not equivalent to the original problem. Although it is not clear whether this approach could successfully be applied to arbitrary

graph structures, for example, their system REMO solves the problem of learning structurally indeterminate clauses, much in line with the extended LINUS propositionalization approach presented in this paper. Their definition of structural clauses whose bodies contain exclusively structural literals, and their algorithm for learning structurally indeterminate clauses can be seen as one of the predecessors of the extended LINUS approach. Recent work by Chevaleyre and Zucker [2000] further elaborates on the issue of transformed representations for multiple-instance data.

Fensel et al. [1995] achieve the transformation from the first-order representation to the propositional representation by ground substitutions which transform clauses to ground clauses. As a consequence they introduce a new definition of positive and negative examples: instead of ground facts they regard ground substitutions as examples. For every possible ground substitution (depending on the number of variables and the alphabet), there is one example in the transformed problem representation. Each background predicate, together with the variable it uses, defines a binary attribute. Like in the previous approach, an example is not described by a single row in the resulting table, but by several rows.

Geibel and Wysotzki [1996] propose a method for feature construction in a graph-based representation. The features are obtained through fixed-length paths in the neighborhood of a node in the graph. This approach distinguishes between "context-dependent node attributes of depth n" and "context-dependent edge attributes of depth n." A context-dependent node attribute $A^n(G)[i, i]$ is defined as follows: For each node i in each example graph G, we define the context of i (of depth n) as all length-n paths from node i and to node i itself. Each such context is used to define a feature. The feature value for an example graph is the number of occurrences of the corresponding context in it. Analogously, a context-dependent edge attribute $A^n(G)[i, j]$ is defined in terms of all length-n paths from node i to node j in a graph G.

Sebag and Rouveirol [1997] developed STILL, an algorithm that performs "stochastic matching" in the test whether a hypothesis covers an example. It operates in the so-called Disjunctive Version Space framework, where for each positive example E, one is interested in the space of all hypotheses covering E and excluding all negative examples F_i. In order to transfer the idea of disjunctive version spaces to first-order logic, STILL reformulates first-order examples in a propositional form. Like in the work by Fensel et al., substitutions are handled as attribute-value examples. In contrast to LINUS and REMO, this reformulation is bottom-up rather than top-down. It is not performed for the complete dataset, but only for one so-called seed example E and for the counterexamples F_i. The reformulation is one-to-one (one example, one row) for the seed example, and one-to-many (one example, several rows) for the counterexamples. Since it would be intractable to use all possible substitutions for the counterexamples, STILL stochastically samples a subset of these. STILL uses a representation, which effectively yields black-box classifiers instead of intelligible features.

Cohen [1996] introduced the notion of "set-valued features," which can be used to transform certain types of background knowledge. A value of a

set-valued feature is allowed to be a set of strings. This type of feature can easily be incorporated in existing propositional learning algorithms. Some first-order learning problems (e.g., text categorization) can be propositionalized in this way.

7. CONCLUSIONS AND FURTHER WORK

This work points out that there is a tradeoff between how much effort a learner puts in the following steps of the hypothesis generation process: rule construction, body construction, and feature construction. We have shown, that by devoting enough effort to feature construction, even complex relational learning tasks can be solved by simple propositional rule learning systems. We perform a simple form of predicate invention through first-order feature construction, and use the constructed features for propositional learning. In this way we have been able to show that the traditional limitations of transformation-based approaches such as LINUS (i.e., no local variables in clause bodies) and its successor DINUS (only determinate local variables) can be alleviated by means of nondeterminate first-order feature construction.

While incorporating feature construction extends the capabilities of LINUS and DINUS, it also introduces restrictions in that it is only applicable to individual-centered domains, where there is a clear notion of individual. The advantage of such domains is that they enable the use of a strong feature bias, without which exhaustive feature construction would be unfeasible. Typical instances of learning tasks we can handle are concept learning and classification tasks; typical instances we cannot handle are program synthesis tasks. We do not see this as a strong limitation, since concept learning and program synthesis are two very different tasks, which are probably not solvable with one and the same learning method.

While in this work feature construction was exhaustive within a given feature bias, in future work we intend to study two nonexhaustive approaches. The first is to use a filter for eliminating irrelevant features, as done in the work by Lavrač et al. [1998] and demonstrated above with some experiments. One may further reduce the set of relevant features by selecting a quasi-minimal set of features needed for hypothesis construction, as proposed by Lavrač et al. [1995]. As an alternative approach, we plan to use a descriptive learner such as Tertius [Flach and Lachiche 2001] for constructing features that correlate significantly with the class attribute.

Postscript

We are pleased to be able to dedicate this paper to Bob Kowalski on the occasion of his 60th birthday. His work on the use of logic and logic programming in artificial intelligence has always been a source of inspiration. While the term "inductive generalization" does appear on the final pages of *Logic for problem solving*, induction remains one of the very few reasoning forms Bob has not tried his hand at—yet. We hope our paper will inspire him to consider taking up this challenge as well.

ACKNOWLEDGMENTS

We thank the three anonymous reviewers for many helpful suggestions. We are grateful to Nicolas Lachiche for joint work on feature construction in 1BC, to Dragan Gamberger for joint work on irrelevant literal elimination, and to Sašo Džeroski and Marko Grobelnik for the collaboration in the development of LINUS. The work reported in this paper was partially supported by the British Royal Society, the British Council, the Slovenian Ministry of Science and Technology, and the Esprit Framework V project Data Mining and Decision Support for Business Competitiveness: A European Virtual Enterprise (IST-1999-11495).

REFERENCES

AGRAWAL, R., MANNILA, H., SRIKANT, R., TOIVONEN, H., AND VERKAMO, A. 1996. Fast discovery of association rules. In *Advances in Knowledge Discovery and Data Mining*, U. Fayyad, G. Piatetski-Shapiro, P. Smyth, and R. Uthurusamy, Eds. AAAI Press, 307–328.

BERGADANO, F. AND GUNETTI, D. 1995. *Inductive Logic Programming: From Machine Learning to Software Engineering*. The MIT Press.

BRATKO, I., MOZETIČ, I., AND LAVRAČ, N. 1989. *KARDIO: A Study in Deep and Qualitative Knowledge for Expert Systems*. MIT Press, Cambridge, MA.

CARUANA, R. AND FREITAG, D. 1994. Greedy attribute selection. In *Proceedings of the 11th International Conference on Machine Learning*. Morgan Kaufmann, 28–36.

CHEVALEYRE, Y. AND ZUCKER, J. 2000. Noise-tolerant rule induction from multi-instance data. In *Proceedings of the ICML-2000 workshop on Attribute-Value and Relational Learning: Crossing the Boundaries*, L. De Raedt and S. Kramer, Eds.

CLARK, P. AND BOSWELL, R. 1991. Rule induction with CN2: Some recent improvements. In *Proc. Fifth European Working Session on Learning*. Springer, Berlin, 151–163.

CLARK, P. AND NIBLETT, T. 1989. The CN2 induction algorithm. *Machine Learning 3*, 4, 261–283.

COHEN, W. 1996. Learning trees and rules with set-valued features. In *Proceedings of the 14th National Conference on Artificial Intelligence*. AAAI Press, 709–716.

DE RAEDT, L. 1998. Attribute-value learning versus inductive logic programming: The missing links. In *Proceedings of the 8th International Conference on Inductive Logic Programming*, D. Page, Ed. Lecture Notes in Artificial Intelligence, vol. 1446. Springer-Verlag, 1–8.

DEHASPE, L. AND TOIVONEN, H. 1999. Discovery of frequent datalog patterns. *Data Mining and Knowledge Discovery 3*, 1, 7–36.

DIETTERICH, T., LATHROP, R., AND LOZANO-PEREZ, T. 1997. Solving the multiple-instance problem with axis-parallel rectangles. *Artificial Intelligence 89*, 31–71.

DŽEROSKI, S., MUGGLETON, S., AND RUSSELL, S. 1992. PAC-learnability of determinate logic programs. In *Proceedings of the 5th ACM Workshop on Computational Learning Theory*. ACM Press, 128–135.

FAYYAD, U., PIATETSKY-SHAPIRO, G., SMYTH, P., AND R. UTHURUSAMY, E. 1995. *Advances in Knowledge Discovery and Data Mining*. The MIT Press.

FENSEL, D., ZICKWOLFF, M., AND WIESE, M. 1995. Are substitutions the better examples? Learning complete sets of clauses with Frog. In *Proceedings of the 5th International Workshop on Inductive Logic Programming*, L. De Raedt, Ed. Department of Computer Science, Katholieke Universiteit Leuven, 453–474.

FLACH, P. A., GIRAUD-CARRIER, C., AND LLOYD, J. 1998. Strongly typed inductive concept learning. In *Proceedings of the 8th International Conference on Inductive Logic Programming*, D. Page, Ed. Lecture Notes in Artificial Intelligence, vol. 1446. Springer-Verlag, 185–194.

FLACH, P. A. AND LACHICHE, N. 1999. 1BC: A first-order Bayesian classifier. In *Proceedings of the 9th International Workshop on Inductive Logic Programming*, S. Džeroski and P. A. Flach, Eds. Lecture Notes in Artificial Intelligence, vol. 1634. Springer-Verlag, 92–103.

FLACH, P. A. 1999. Knowledge representation for inductive learning. In *Symbolic and Quantitative Approaches to Reasoning and Uncertainty (ECSQARU'99)*, A. Hunter and S. Parsons, Eds. Lecture Notes in Artificial Intelligence, vol. 1638. Springer-Verlag, 160–167.

FLACH, P. A. AND KAKAS, A. C. 2000. Abductive and inductive reasoning: background and issues. In *Abductive and inductive reasoning: essays on their relation and integration*, P. A. Flach and A. C. Kakas, Eds. Kluwer Academic Publishers.

FLACH, P. A. AND LACHICHE, N. 2001. Confirmation-guided discovery of first-order rules with Tertius. *Machine Learning 42*, 1/2, 61–95.

GAMBERGER, D. 1995. A minimization approach to propositional inductive learning. In *Proceedings of the 8th European Conference on Machine Learning*. Springer-Verlag, 151–160.

GEIBEL, P. AND WYSOTZKI, F. 1996. Relational learning with decision trees. In *Proceedings of the 12th European Conference on Artificial Intelligence*. 428–432.

JOHN, G., KOHAVI, R., AND PFLEGER, K. 1994. Irrelevant features and the subset selection problem. In *Proceedings of the 11th International Conference on Machine Learning*. Morgan Kaufmann, 190–198.

KAKAS, A. AND RIGUZZI, F. 1997. Learning with abduction. In *Proceedings of the 7th International Workshop on Inductive Logic Programming*, S. Džeroski and N. Lavrač, Eds. Lecture Notes in Artificial Intelligence, vol. 1297. Springer-Verlag, 181–188.

KHAN, K., MUGGLETON, S., AND PARSON, R. 1998. Repeat learning using predicate invention. In *Proceedings of the 8th International Conference on Inductive Logic Programming*, D. Page, Ed. Lecture Notes in Artificial Intelligence, vol. 1446. Springer-Verlag, 165–174.

KLÖSGEN, W. 1996. Explora: A multipattern and multistrategy discovery assistant. In *Advances in Knowledge Discovery and Data Mining*, U. Fayyad, G. Piatetsky-Shapiro, P. Smyth, and R. Uthurusamy, Eds. AAAI Press, 249–271.

KOHAVI, R., SOMMERFIELD, D., AND DOUGHERTY, J. 1996. Data mining using MLC++: A machine learning library in C++. In *Tools with Artificial Intelligence*. IEEE Computer Society Press. `http://www.sgi.com/Technology/mlc`.

KOLLER, D. AND SAHAMI, M. 1996. Toward optimal feature selection. In *Proceedings of the 13th International Conference on Machine Learning*. 284–292.

KRAMER, S., PFAHRINGER, B., AND HELMA, C. 1998. Stochastic propositionalization of nondeterminate background knowledge. In *Proceedings of the 8th International Conference on Inductive Logic Programming*, D. Page, Ed. Lecture Notes in Artificial Intelligence, vol. 1446. Springer-Verlag, 80–94.

LAVRAČ, N. AND DŽEROSKI, S. 1992. Background knowledge and declarative bias in inductive concept learning. In *Proceedings 3rd International Workshop on Analogical and Inductive Inference*, K. Jantke, Ed. Springer-Verlag, 51–71. (Invited paper).

LAVRAČ, N. AND DŽEROSKI, S. 1994. *Inductive Logic Programming: Techniques and Applications*. Ellis Horwood.

LAVRAČ, N., DŽEROSKI, S., AND GROBELNIK, M. 1991. Learning nonrecursive definitions of relations with LINUS. In *Proceedings of the 5th European Working Session on Learning*, Y. Kodratoff, Ed. Lecture Notes in Artificial Intelligence, vol. 482. Springer-Verlag, 265–281.

LAVRAČ, N., GAMBERGER, D., AND DŽEROSKI, S. 1995. An approach to dimensionality reduction in learning from deductive databases. In *Proceedings of the 5th International Workshop on Inductive Logic Programming*, L. De Raedt, Ed. Department of Computer Science, Katholieke Universiteit Leuven, 337–354.

LAVRAČ, N., GAMBERGER, D., AND JOVANOSKI, V. 1999. A study of relevance for learning in deductive databases. *Journal of Logic Programming 40*, 2/3 (August/September), 215–249.

LAVRAČ, N., GAMBERGER, D., AND TURNEY, P. 1998. A relevancy filter for constructive induction. *IEEE Intelligent Systems 13*, 2 (March–April), 50–56.

LLOYD, J. 1987. *Foundations of Logic Programming*, 2nd ed. Springer, Berlin.

LLOYD, J. 1999. Programming in an integrated functional and logic language. *Journal of Functional and Logic Programming 1999*, 3 (March).

MICHALSKI, R. 1983. A theory and methodology of inductive learning. In *Machine Learning: An Artificial Intelligence Approach*, R. Michalski, J. Carbonell, and T. Mitchell, Eds. Vol. I. Tioga, Palo Alto, CA, 83–134.

MICHIE, D., MUGGLETON, S., PAGE, D., AND SRINIVASAN, A. 1994. To the international computing community: A new East-West challenge. Tech. rep., Oxford University Computing laboratory, Oxford, UK.

MIZOGUCHI, F., OHWADA, H., DAIDOJI, M., AND SHIRATO, S. 1996. Learning rules that classify ocular fundus images for glaucoma diagnosis. In *Proceedings of the 6th International Workshop on Inductive Logic Programming*, S. Muggleton, Ed. Lecture Notes in Artificial Intelligence, vol. 1314. Springer-Verlag, 146–162.

MUGGLETON, S., Ed. 1992. *Inductive Logic Programming*. Academic Press.

MUGGLETON, S. 1995. Inverse entailment and Progol. *New Generation Computing, Special issue on Inductive Logic Programming 13*, 3–4, 245–286.

MUGGLETON, S. AND DE RAEDT, L. 1994. Inductive logic programming: Theory and methods. *Journal of Logic Programming 19/20*, 629–679.

MUGGLETON, S. AND FENG, C. 1990. Efficient induction of logic programs. In *Proceedings of the 1st Conference on Algorithmic Learning Theory*. Ohmsha, Tokyo, Japan, 368–381.

MUGGLETON, S., KING, R., AND STERNBERG, M. 1992. Protein secondary structure prediction using logic. In *Proceedings of the 2nd International Workshop on Inductive Logic Programming*, S. Muggleton, Ed. Report ICOT TM-1182. 228–259.

MUGGLETON, S., SRINIVASAN, A., KING, R., AND STERNBERG, M. 1998. Biochemical knowledge discovery using Inductive Logic Programming. In *Proceedings of the first Conference on Discovery Science*, H. Motoda, Ed. Springer-Verlag, Berlin.

NIENHUYS-CHENG, S.-H. AND DE WOLF, R. 1997. *Foundations of Inductive Logic Programming*. Lecture Notes in Artificial Intelligence, vol. 1228. Springer-Verlag.

OLIVEIRA, A. AND SANGIOVANNI-VINCENTELLI, A. 1992. Constructive induction using a non-greedy strategy for feature selection. In *Proceedings of the 9th International Workshop on Machine Learning*. 354–360.

PAGALLO, G. AND HAUSSLER, D. 1990. Boolean feature discovery in empirical learning. *Machine Learning 5*, 71–99.

QUINLAN, J. 1990. Learning logical definitions from relations. *Machine Learning 5*, 239–266.

ROUVEIROL, C. 1994. Flattening and saturation: Two representation changes for generalization. *Machine Learning 14*, 2, 219–232.

SEBAG, M. AND ROUVEIROL, C. 1997. Tractable induction and classification in first-order logic via stochastic matching. In *Proceedings of the 15th International Joint Conference on Artificial Intelligence*. Morgan Kaufmann, 888–893.

SKALAK, D. 1994. Prototype and feature selection by sampling and random mutation hill climbing algorithms. In *Proceedings of the 11th International Conference on Machine Learning*. Morgan Kaufmann, 293–301.

SRINIVASAN, A. AND KING, R. 1996. Feature construction with inductive logic programming: A study of quantitative predictions of biological activity aided by structural attributes. In *Proceedings of the 6th International Workshop on Inductive Logic Programming*, S. Muggleton, Ed. Lecture Notes in Artificial Intelligence, vol. 1314. Springer-Verlag, 89–104.

STAHL, I. 1996. Predicate invention in inductive logic programming. In *Advances in Inductive Logic Programming*, L. De Raedt, Ed. IOS Press, 34–47.

TURNEY, P. 1996. Low size-complexity inductive logic programming: The East-West challenge considered as a problem in cost-sensitive classification. In *Advances in Inductive Logic Programming*, L. De Raedt, Ed. IOS Press, 308–321.

ULLMAN, J. 1988. *Principles of Database and Knowledge Base Systems*. Vol. I. Computer Science Press, Rockville, MA.

WITTEN, I. AND FRANK, E. 2000. *Data Mining: Practical Machine Learning Tools and Techniques with Java Implementations*. Morgan Kaufmann.

WNEK, J. AND MICHALSKI, R. 1991. Hypothesis-driven constructive induction in AQ17: A method and experiments. In *Proceedings of IJCAI-91 Workshop on Evaluating and Changing Representations in Machine Learning*. Sydney, 13–22.

WROBEL, S. 1997. An algorithm for multi-relational discovery of subgroups. In *Proceedings of the 1st European Symposium on Principles of Data Mining and Knowledge Discovery*, J. Komorowski and J. Zytkow, Eds. Springer-Verlag.

ZUCKER, J. AND GANASCIA, J. 1998. Learning structurally indeterminate clauses. In *Proceedings of the 8th International Conference on Inductive Logic Programming*, D. Page, Ed. Lecture Notes in Artificial Intelligence, vol. 1446. Springer-Verlag, 235–244.

ZUCKER, J.-D. AND GANASCIA, J.-G. 1996. Representation changes for efficient learning in structural domains. In *Proceedings of the 13th International Conference on Machine Learning*, L. Saitta, Ed. Morgan Kaufmann, 543–551.

Received March 2000; revised February 2001; accepted February 2001