Machine Learning, 26, 99–146 (1997)

$\mathcal{C}$lausal $\mathcal{D}$iscovery

LUC DE RAEDT luc.deraedt@cs.kuleuven.ac.be
LUC DEHASPE luc.dehaspe@cs.kuleuven.ac.be
Department of Computer Science, Katholieke Universiteit Leuven,
Celestijnenlaan 200A, B-3001 Heverlee, Belgium

Editor: Stephen Muggleton and David Page

Abstract. The clausal discovery engine $\mathcal{C}$LAUDIEN is presented. $\mathcal{C}$LAUDIEN is an inductive logic programming engine that fits in the descriptive data mining paradigm. $\mathcal{C}$LAUDIEN addresses characteristic induction from interpretations, a task which is related to existing formalisations of induction in logic. In characteristic induction from interpretations, the regularities are represented by clausal theories, and the data using Herbrand interpretations. Because $\mathcal{C}$LAUDIEN uses clausal logic to represent hypotheses, the regularities induced typically involve multiple relations or predicates. $\mathcal{C}$LAUDIEN also employs a novel declarative bias mechanism to define the set of clauses that may appear in a hypothesis.

Keywords: Inductive Logic Programming, Knowledge Discovery in Databases, Data Mining, Learning, Induction, Semantics for Induction, Logic of Induction, Parallel Learning

1. Introduction

Despite the fact that the areas of knowledge discovery in databases (Fayyad *et al.*, 1995) and inductive logic programming (Muggleton & De Raedt, 1994) have both enjoyed a lot of attention recently, the combination of the two areas has seldomly been studied (Džeroski, 1995). Enhancing data mining tools with relational abilities as offered by inductive logic programming is of crucial importance for the practice of knowledge discovery due to the central role of relational databases in database technology (Morik & Brockhausen, 1996). Yet, most data mining techniques focus on learning within a single relation. On the other hand, inductive logic programming has always focused on learning classification rules, i.e. on performing concept-learning from positive and negative examples of a concept. In contrast, descriptive data mining is often aimed at finding interesting regularities in unclassified data.

$\mathcal{C}$LAUDIEN[1] combines data mining principles with inductive logic programming. As such it discovers clausal regularities from unclassified data. To this aim, a novel semantics (or problem-setting) for inductive logic programming has been developed, cf. (De Raedt & Džeroski, 1994), in which examples are represented by Herbrand interpretations and the aim is to discover a logically maximally general hypothesis that has all the examples as models. The novel semantics is called characteristic induction from interpretations. The special case, where the data consists of a single model or interpretation was earlier proposed in a slightly different form by Nicolas Helft (Helft, 1989). The setting is compared and contrasted with other formalisations of inductive logic programming and its various properties are presented. One of the properties of the proposed semantics is monotonicity, meaning that whenever two individual clauses are valid on the data, their conjunction will also be valid on the data. Monotonicity is not satisfied by the usual inductive logic programming semantics. Monotonicity makes it easy to implement a parallel clausal discovery engine. Algorithms that address the proposed problem-setting are presented, shown to be correct and tested on a wide range of applications.

A key ingredient of the clausal discovery engine is the definition of the declarative bias, which determines the type of regularity searched for. Declarative bias is essential in descriptive data mining as such systems have a less operational criterion of success than concept-learning. In concept-learning, one typically searches for any hypothesis consistent with the data whereas data mining is looking for all interesting or valid regularities. The number of regularities satisfying the criterion can be very large as shown also in propositional approaches to data mining. As the search space of clausal logic is larger (and even infinite) than that of propositional logic, bias is of crucial importance in clausal discovery. To declaratively represent the bias of the clausal discovery engine, a new formalism, called $\mathcal{D}$LAB, derived from the work of (Adé *et al.*, 1995, Emde *et al.*, 1983, Kietz & Wrobel, 1992, Bergadano & Gunetti, 1993, Cohen, 1994) is proposed. Moreover, it is shown how the specification of the syntax of the clauses allowed in the hypothesis can be automatically translated in a refinement operator for the considered language. $\mathcal{D}$LAB should also be useful in other inductive logic programming systems.

The practice of the clausal discovery engine is demonstrated using a variety of experiments. The first experiment demonstrates the generality of the clausal discovery engine in a data mining context by showing that the engine is able to emulate many of the descriptive data mining systems specifically designed for particular induction tasks such as finding functional or multi-valued dependencies and association rules. This is achieved by tuning $\mathcal{C}$LAUDIEN's parameters, especially the declarative bias. In a second example, inspired by (Bratko & Grobelnik, 1993), we show how functors are handled to recover loop invariants from program traces. The third experiment, in finite element mesh-design (Dolšak & Muggleton, 1992, Lavrač & Džeroski, 1994), shows that – although $\mathcal{C}$LAUDIEN is not intended to perform classification tasks – it can also be successfully applied in this context. Two further experiments, on mutagenesis (Srinivasan *et al.*, 1995b) and water-quality ((Džeroski *et al.*, 1994)), show $\mathcal{C}$LAUDIEN's performance on particular data mining tasks.

This paper is organised as follows: In Section 2, we review the concepts from (inductive) logic programming used, in Section 3, we introduce the novel semantics for inductive logic programming and contrast it with existing ones, in Section 4, we present a sequential and parallel algorithm for performing clausal discovery, we introduce a novel mechanism to declaratively represent the bias of the discovery engine, and present heuristics and extensions of the proposed algorithm, in Section 5, we show the effectiveness of the engine on a wide range of applications. Finally, in Sections 6 and 7, we conclude and touch upon related work.

2. (Inductive) Logic Programming Concepts

We assume some familiarity with first order logic (see (Bratko, 1986, Lloyd, 1987, Genesereth & Nilsson, 1987, De Raedt, 1996) for an introduction).

A first order alphabet is a set of predicate symbols, constant symbols and functor symbols. A clause is a formula of the form $A_1, ..., A_m \leftarrow B_1, ..., B_n$ where the A_i and B_i are logical atoms. An atom $p(t_1, ..., t_n)$ is a predicate symbol p followed by a bracketed n-tuple of terms t_i. A term t is a variable V or a functor symbol $f(t_1, ..., t_k)$ immediately followed by a bracketed k-tuple of terms t_i. Constants are functor symbols of arity 0. *Functor-free* clauses are clauses that contain only variables as terms.

The above clause can be read as A_1 or ... or A_m if B_1 and ... and B_n. All variables in clauses are universally quantified, although this is not explicitly written. Extending the usual convention for *definite clauses* (where $m = 1$), we call $A_1, ..., A_m$ the *head* of the clause and $B_1, ..., B_n$ the *body* of the clause. A *fact* is a definite clause with an empty body, $(m = 1,\ n = 0)$.

A *Herbrand interpretation* over a first order alphabet is a set of ground atoms constructed with the predicate, constant and functor symbols in the alphabet. Roughly speaking, a Herbrand interpretation represents a kind of possible world by specifying all true facts in the world. All facts not stated are assumed to be false.

A Herbrand interpretation is the equivalent of an example in propositional approaches to inductive learning using e.g. attribute value representations or boolean logic. Suppose we are using an attribute value representation where all attributes can have two values (say true and false). An example would then state for all attributes whether its value is true or false. This corresponds to the Herbrand interpretation consisting of all attributes (i.e. propositions) having the value true in the example. This is also similar to computational learning theory applied to boolean logic, which has used boolean variable assignments (i.e. assignments of 1 or 0 to the variables).

As in concept-learning, a notion of coverage is needed. When a Herbrand interpretation is a model for a theory, we will consider the interpretation ‘covered’ by the theory. Formally, a Herbrand interpretation I is a model for a clause c if and only if for all grounding substitutions θ of c : $body(c)\theta \subset I \rightarrow head(c)\theta \cap I \neq \emptyset$. We also say c is true in I. A Herbrand interpretation I is a model for a clausal theory T if and only if it is a model for all clauses in T. Roughly speaking, the truth of a clause c in an interpretation I can be determined by running the query $? - body(c), not\ head(c)$ on a database containing I using a theorem prover (such as PROLOG). If the query succeeds, the clause is false in I. If it finitely fails, the clause is true.

Inductive logic programming systems typically deal with background knowledge. In our setting, background knowledge (a definite clause theory) will be used to complete an observation (in this case, also a set of definite clauses) into a Herbrand interpretation. The least Herbrand interpretation of a definite clause theory is the set of all ground facts (using the predicates, functors and constants of the definite clause theory) that are logically entailed by the definite clause theory. We will use the notation $M(T)$ to denote the least Herbrand model of a definite clause theory T.

Example 1 *Consider the following definite clause theory:*

$$flies(X) \leftarrow normal(X), bird(X)$$

$$normal(tweety) \leftarrow$$

$$bird(tweety) \leftarrow$$

Then the least Herbrand model of this theory is:

$$\{bird(tweety), normal(tweety), flies(tweety)\}$$

This Herbrand interpretation is a model for the clause:

$$flies(X) \leftarrow bird(X)$$

The following clause is false in the Herbrand interpretation:

$$\leftarrow bird(X), normal(X)$$

We will employ two notions of generality in this paper. A clausal theory T_1 is *logically more general than* a clausal theory T_2 if and only if $T_1 \models T_2$, i.e. if T_1 logically entails T_2. The other notion employed is Plotkin's θ-subsumption (Plotkin, 1970). A clause c_1 *θ-subsumes* clause c_2 if and only if there exists a substitution θ such that $c_1\theta \subseteq c_2$.

3. Logical Frameworks for Induction

At present, there exist several formalisations of induction in clausal logic. Firstly, there the normal inductive logic programming setting (sometimes also called the explanatory setting) introduced by Gordon Plotkin (Plotkin, 1970), which is employed by the large majority of inductive logic programming systems, cf. (Muggleton & De Raedt, 1994), which aims at discriminating positive observations from negative ones, and hence is classification oriented. Secondly, there is Nicolas Helft's non-monotonic setting (Helft, 1989), which aims at characterising one or more observations, and hence is oriented towards descriptive data mining. Thirdly, there is the confirmatory setting by Peter Flach (Flach, 1995). Fourthly, there is Mannila's general framework for data mining (cf. (Mannila, 1995)). Fifth, there is the setting introduced by De Raedt and Džeroski (De Raedt & Džeroski, 1994), which we will employ for clausal discovery, and which we will call characteristic induction from interpretations[2]. In this section, we will introduce this induction setting and discuss its relation to the other ones.

3.1. Characteristic induction from interpretations

Our setting for induction is derived from Nicolas Helft's non-monotonic semantics for induction (Helft, 1989), cf. (De Raedt & Džeroski, 1994). Although it differs from Helft's setting in several respects, it is similar in spirit. The ideas are 1) that all observations are completely specified, and 2) that a hypothesis should reflect what is in the data. The first idea is implemented by representing the observations as Herbrand interpretations, with the consequence that all observations are assumed to be completely specified (as in attribute-value learning). The second idea is enforced by requiring all hypotheses to be true in all of the observations. Since we are only working with one type of observation, we perform *characteristic* induction, a term which is due to (Michalski, 1983).

Ignoring for the moment the use of background knowledge, characteristic induction from interpretations can be defined as follows.

Definition 1 (Characteristic induction from interpetations) *Let O be a set Herbrand Interpretations, $\mathcal{L}$ a set of clauses. $H \subset \mathcal{L}$ is a solution if and only if H is a logically maximally general valid*[3] *hypothesis. A hypothesis H is valid if and only if for all $o_i \in O$, H is true in o_i.*

We will impose syntactic restrictions on the space of hypotheses through the language $\mathcal{L}$, which determines the set of clauses that can be part of a hypothesis. The language $\mathcal{L}$ is an important parameter of the induction task. It can have different properties (e.g. be infinite or finite) depending on the problem.

Language Assumption. The language assumption states that the alphabet of the hypotheses language $\mathcal{L}$ only contains constant, functor or predicate symbols that occur in one of the observations or in the background theory.

Example 2 *Imagine we are observing different gorilla colonies and we observe two different colonies*

$o_1 = \{female(liz), male(richard), gorilla(liz), gorilla(richard)\}$
$o_2 = \{female(ginger), male(fred), gorilla(ginger), gorilla(fred)\}$.

A clause is range-restricted *if all variables in the head of the clause also appear in the body of the clause. If $\mathcal{L}$ is restricted to range-restricted, constant-free clauses a solution is:*

(1) gorilla(X) ← female(X)
(2) gorilla(X) ← male(X)
(3) male(X), female(X) ← gorilla(X)
(4) ← male(X),female(X)

This is a solution because all clauses (1-4) are true in the Herbrand interpretations o_1, o_2. Furthermore, all other valid clauses over the same alphabet are logically entailed by this hypothesis. To see this, observe that as all predicates are unary and there are only three predicates, it suffices to restrict our attention to clauses with at most 3 literals in the head and at most 3 literals in the body as all clauses with more literals are equivalent to one of this form. The result then follows by enumerating the clauses, and removing logically redundant ones.

Background knowledge can easily be incorporated in the above definition. Let B be a background theory in the form of a definite clause theory^4. Let each observation $o_i \in O$ also be a definite clause theory. Then a hypothesis will be valid if and only if for all $o_i \in O$, H is true in $M(B \cup o_i)$. Thus, background knowledge is used to complete the observations into Herbrand interpretations. From now on, for reasons of readability, we will act as if no background knowledge is used. However, all of our definitions and results also hold when background knowledge is used as just indicated.

3.2. Properties of the framework

First, each observation is a Herbrand interpretation. This is only justified when complete knowledge of all (relevant) aspects of the observation is available. As an illustration, suppose we have two birds, the first of which is known to be black, and the second having an unknown colour. Under these circumstances, it is not valid to say that all birds are black (as we do not know whether this statement holds for the second bird). Thus the use of Herbrand interpretations assumes complete knowledge of each observation o_i. If such knowledge is not available one should be cautious with this approach.

Second, we are interested in hypotheses that are valid. Intuitively, validity means that the hypothesis holds on the data, i.e. that the induced hypothesis postulates true regularities present in the observations. This is $-$ as we shall see $-$ a stronger requirement than those employed in the normal inductive logic programming framework. Validity is a monotone property at the level of hypotheses:

Property 1 (Monotonicity) *If H_1 is valid and H_2 is valid with respect to a set of observations O, then $H_1 \cup H_2$ is valid.*

This property means that all well-formed clauses in $\mathcal{L}$ can be considered completely independent of each other. It will turn out to be very important for efficiency reasons as it essentially allows for parallel search (cf. Section 4.3).

Third, the condition of maximal generality (cf. also (De Raedt, 1996) for an alternative explanation). This condition appears in the definition because the most interesting hypotheses are the most informative and hence the most general. Without this condition, the empty hypothesis (which is always valid) would be a trivial solution and this is undesirable.

The casual reader less interested in logical and formal aspects of the framework and relations to other logical frameworks may want to go to section 4.

The question now arises as to the circumstances under which a maximally general valid hypothesis exists. In general, for infinite hypotheses spaces, a maximally general hypothesis will not exist. This is demonstrated in Example 3.

Example 3 *Consider the single observation $\{parent(luc, soetkin) \leftarrow\}$. Then the following clauses are all valid:*

(1) $\leftarrow$ parent(X_1,X_1)
(2) $\leftarrow$ parent(X_1,X_2), parent(X_2,X_1)
(3) $\leftarrow$ parent(X_1,X_2), parent(X_2,X_3), parent(X_3,X_4),parent(X_4,X_1)

...

It is clear that there exists here a strictly ascending chain (according to generality) of clauses which are all valid. If we restrict $\mathcal{L}$ to this set of clauses, the maximally general hypothesis should be an infinite clause.

However, in case a maximally general hypothesis exists, then all such hypotheses are logically equivalent.

Property 2 *If there exists a solution, then the solution is unique up to logical equivalence.*

Proof: suppose there are two maximally general solutions H_1 and H_2 and $\not\models H_1 \leftrightarrow H_2$. Because of monotonocity $H_1 \cup H_2$ must also be valid, and $H_1 \cup H_2$ is strictly more general than H_1 and than H_2. This contradicts the fact that H_1 and H_2 are maximally general. □

There are two possible ways to avoid the problems with infinite solutions. The first solution is to require that the set of well-formed clauses $\mathcal{L}$ is finite. Although this solution may appear to be undesirable, it is made by the vast majority of current approaches to inductive logic programming. It will be used in the implementation of the clausal discovery engine and enforced using the declarative language bias formalism. The second solution is due to Nicolas Helft (but generalized here) and works only when the Herbrand interpretations are finite.

Definition 2 (Injectivity) *Let c be $p_1, ..., p_m \leftarrow q_1, ..., q_n$ and let vars$(c) = \{X_1, ..., X_k\}$. The clause c is injective with regard to a set of observations O if and only if either, $m > 0$ and there exists an observation $o \in O$, and a substitution θ such that $(q_1 \wedge ... \wedge q_n \wedge X_1 \neq X_2, ..., X_i \neq X_j, ...)\theta$ is true in o augmented with standard inequality, or, $m = 0$ and for all k, clause $\neg q_k \leftarrow q_1, ..., q_{k-1}, q_{k+1}, ..., q_n$ is injective.*

Injectivity Assumption. The injectivity assumption requires that all clauses in a solution be injective.

The problems with Example 3 disappear when the injectivity assumption is made. Indeed, the unique maximally general injective valid clause is clause (2). The intuition here is that one should not employ more variables than needed, and as the maximum chain of constants linked by the parent relation is 2, we should not introduce more variables.

Property 3 *If the Herbrand interpretations $o_i \in O$ are finite and the injectivity assumption holds, then there exists a finite set of clauses that forms a solution.*

Proof: Let n be the maximum number of terms occurring in one of the Herbrand interpretations. By assumption n is finite. Let $X_1, ..., X_n$ be n different variables. As each injective clause can contain at most n different variables, it suffices to consider clauses with as only variables the $X_1, ..., X_n$. Therefore the only literals that need to be considered are those with the predicates and terms in the Herbrand interpretations o_i, and the variables $X_1, ..., X_n$. As there are only a finite number of such literals, the number of clauses containing such literals is also finite. Let H contain all such clauses that are valid. H is finite and an injective solution. ■

The injectivity assumption, however, does not help when the Herbrand universe is infinite: see Example 4.

Example 4 *Let* o *be* $M(\{parent(X, p(X)), human(a)\})$. *Then the problems outlined in Example 3 reappear.*

3.2.1. Additional options

A weaker but also useful condition than injectivity is that of non-triviality.

Non-triviality Assumption. Let c be $p_1, ..., p_m \leftarrow q_1, ..., q_n$. The clause c is non-trivial w.r.t. a set of observations O if and only if either $m > 0$ and there exists an observation $o \in O$ and a substitution θ such that $(q_1 \wedge ... \wedge q_n)\theta$ is true in o, or, $m = 0$ and for all k there exists a substitution θ and an observation o such that $(q_1 \wedge ... \wedge q_{k-1} \wedge q_{k+1} \wedge q_n)\theta$ is true in o.

Non-triviality is used to exclude clauses that trivially hold from the hypotheses. Without non-triviality, one can always postulate implications, provided that the condition part never holds.

Example 5 *Consider as background theory:*

colour(X) $\leftarrow$ *black(X)*
colour(X) $\leftarrow$ *white(X)*

and as observation $\{swan(s), white(s)\}$. *Without requiring non-triviality the clause swan(X)* $\leftarrow$ *black(X) is valid. This is not always desirable.*

An alternative to the non-triviality condition for denials would be to demand maximally general clauses.

Maximally general clauses. Under this assumption, it is required that all clauses c in a solution H, are maximally general and valid. This means that there is no clause c' that θ-subsumes c and is also valid on the observations[5].

The condition of maximally general clauses is however harder to enforce than non-triviality due to the possibility of strictly infinitely ascending chains of clauses under θ-subsumption, which may again lead to a need for adding infinite clauses to the hypotheses.

Another option relates to the issue of redundant hypotheses. Clauses that belong to the background theory may reappear in the induced hypothesis. This is not always desirable. It can be avoided by the non-redundancy assumption.

Non-redundancy Assumption. No clause $c \in H$ is logically entailed by B, i.e. for all c $\in H : B \not\models c$.

A related requirement requires a minimal solution, i.e. a solution in which no clause is logically redundant with respect to the induced hypothesis.

Compactness Assumption. No clause $c \in H$ is logically entailed by $H - \{c\}$, i.e. for all c $\in H : H - \{c\} \not\models c$.

3.3. Relation to other frameworks for induction

3.3.1. Michalski's notions

The problem of characteristic induction from interpretations as formalized here, can be regarded as a logical formalisation of the task addressed by Michalski's INDUCE system (Michalski, 1983). Employing the framework of logic programming has several advantages. First, the definitions employed have a clear and well understood meaning. Second, using (and implementing) background knowledge is very easy (employing e.g. PROLOG).

3.3.2. Helft's and Flach's notions

The key difference with Helft's notion of induction is that Helft assumes a single observation. Working with multiple observations is more natural as many well-known machine learning notions such as for instance incrementality have a clear meaning in our framework. Furthermore, by working with multiple observations, the boolean PAC-learning setting is generalized, cf. also (De Raedt & Džeroski, 1994). Other differences with Helft's framework include the use of *Herbrand* models as well as that we allow for functors.

Flach's adequacy conditions for induction provide a framework for reasoning about the properties and semantics of induction. However, Flach's adequacy conditions allow for many instantiations. Our framework can be considered one such instantiation, which is close to Flach's *confirmatory setting*.

3.3.3. Normal Inductive Logic Programming

Our setting for induction is specifically tailored towards the discovery of regularities that hold in a set of (unclassified) observations or that *characterize* the observations. Within inductive logic programming and other forms of machine learning, people have classically focused on learning rules that *discriminate* positive observations from negative ones. Within normal inductive logic programming this is captured in the following definition, due to (Plotkin, 1970).

Definition 3 (Normal Inductive Logic Programming) *Let P be a set of true observations, N be a set of false observations, B a background theory. $H \subset \mathcal{L}$ is a solution if and* *only if H is complete with regard to the positive observations and consistent with regard to the negative observations. A hypothesis H is complete with regard to P and B if and only if $B \cup H \models P$; H is consistent with regard to N and B if and only if $B \cup H \cup N \not\models \Box$.*

Example 6 *Suppose $P = \{flies(tweety), flies(woody)\}$, $N = \{\neg flies(oliver)\}$, $B = \{bird(tweety), bird(woody), bird(oliver), normal(tweety), normal(woody)\}$. Then a solution would be flies(X) ← bird(X), normal(X).*

The aim of normal inductive logic programming is to induce a hypothesis that logically entails all of the true observations and none of the false observations. An important property is:

Property 4 *If H_1 is consistent and H_2 is consistent with respect to a background theory B and a set of observations O, then $H_1 \cup H_2$ need not be consistent with O.*

This property is the cause of some well-known problems when learning multiple predicates or recursive predicates in the normal inductive logic programming setting, cf. (De Raedt *et al.*, 1993, Bergadano & Gunetti, 1993, Cameron-Jones & Quinlan, 1993). The reason for this is that inconsistencies may arise when H_1 and H_2 can resolve together.

Flach's (Flach, 1992) definition of weak induction (from which his later notion of confirmatory induction is derived) is the special case of normal inductive logic programming where only consistency with the negative examples is required. The reader may notice that also for this setting by Flach, the above property holds.

The differences between our induction setting and normal inductive logic programming are akin to the differences between knowledge discovery (or data mining) and concept-learning. The differences can be explained in terms of the two ideas underlying our induction setting, i.e. learning from interpretations versus learning from implications, characteristic versus discriminant induction.

A first important difference is due to the representation of the examples. In our setting examples are interpretations, in normal inductive logic programming, examples are implications or clauses. Using interpretations to describe observations is the first order equivalent of what is done in attribute value learning. In attribute value learning each example is described by means of a complete vector of attribute value pairs. Completeness in this respect means that a value for each attribute is known. Working with interpretations thus implicitly corresponds to assuming that all aspects of each observation is known: all examples are assumed to be completely described, and all facts not stated in the observation are regarded false. This contrasts with normal inductive logic programming approaches where examples are definite clauses (possibly obtained after applying some form of saturation on a ground fact). Using definite clauses one can model incomplete information and induce hypotheses that realize an inductive leap on the examples. Let us illustrate this point using a variant of Example 6. The example can be straightforwardly transformed in a set of interpretations, one interpretation for each of the birds, i.e. $tweety$, $woody$, and $oliver$. In this case, complete knowledge of the birds is available. Now, both our setting and normal inductive logic programming would consider *flies(X) ← bird(X), normal(X)* as (part of) a solution. However, let us assume that the fact $flies(tweety)$ is unknown. In normal inductive logic programming the previous solution would still hold and the induction procedure would postulate that $flies(tweety)$ holds. Hence, an inductive leap would result. However, when working with interpertations it would no longer hold as there would be a normal bird of which it is not known whether it flies. This clearly shows that learning from interpretations $-$ in contrast to learning from implications $-$ assumes complete information about the examples and does not allow inductive leaps on the observations, i.e. applying the induced hypotheses on the observations will not result in postulating new facts. Learning from interpretations makes inductive leaps of a different kind, in the sense that it postulates that the induced hypotheses will be valid on unseen observations.

This is the theoretical point of view. In practise however, learning from interpretations can still be applied in the presence of a limited form of incompleteness. The trick is to put the predicates that are known to be incomplete in the condition part of the rules. Thus, with $flies(tweety)$ unknown in Example 6, solutions in our setting would include *bird(X)* ← *flies(X)* and *normal(X)* ← *flies(X)*. Notice we have then learned necessary conditions for $flies(X)$ instead of sufficient ones. From a theoretical perspective, one could handle incomplete information when learning from interpretations by using incomplete interpretations, which would list the known true, and the known false facts. A hypothesis H would then be considered valid with an observation o and a background theory B if and only if $B \wedge H \wedge o \not\models \Box$, which again closely corresponds to Flach's notion of weak induction. Some ideas along this line have also been investigated by (Fensel *et al.*, 1995, Wrobel & Džeroski, 1995). From a practical perspective however, complete knowledge is often available (cf. attribute value learning where missing values arise only seldomly, or well-known inductive logic programming problems such as mutagenesis (Srinivasan *et al.*, 1995b)). Furthermore, it is the assumption of complete knowledge that makes the monotonicity property hold, which is crucial for efficiency reasons, cf. Section 4.3 on parallel search.

The second difference can be explained using the notions of characteristic induction versus discriminant induction. In discriminant induction, the aim is to find a hypothesis that discriminates observations belonging to two classes, i.e. the positive observations from the negative ones. In characteristic induction, the aim is to find a most informative hypothesis that explains all of the (unclassified) observations. A most informative hypothesis is one that covers the least number of examples (the most specific one under coverage). When learning form interpretations most informative means logically maximally general. The reason is that the logically more general hypotheses have the least number of models, hence, they cover the least number of observations (in this case a hypothesis covers an example if the example is valid in the hypothesis). In contrast, when learning from implications most informative means logically maximally specific, as these hypotheses cover the least observations (in this case a hypothesis covers an example if the hypothesis logically entails the example).

These two differences motivate the use of the term characteristic induction from interpretations. Furthermore, it would be adequate to name the normal inductive logic programming setting, discriminant induction from implications (or from entailment, cf. (De Raedt, 1996)).

These two aspects of induction allow us also to describe two other problem settings that have been considered. First, there is the normal inductive logic programming where the set of negative examples is empty. This setting can be described as *characteristic* *induction from implications*, it corresponds to learning from positive data only, and has been considered by many researchers. Secondly, there is no reason why one cannot learn clauses that discriminate interpretations in several classes, e.g. interpretations that are a model for a theory true versus interpretations that are not. This alternative setting has been adopted in the ICL system of (De Raedt & Van Laer, 1995). The ICL setting, discriminant induction from interpretations, provides a clue as how problems and solutions along the different dimensions relate to each other. It should be clear that the set of clauses output by characteristic induction (using the positive observations only) is typically a superset of that produced by a discriminant procedure (we are ignoring all non-logical aspects of induction engines, such as heuristics, now). For instance, when working with interpretations characteristic induction will produce a large set of clauses valid on the positive observations, whereas discriminant induction will retain a minimal subset needed for discriminating the negative observations.

3.3.4. Mannila's data mining framework

Heikki Mannila (Mannila, 1995) recently introduced a general definition for data mining. He views data mining as the process of constructing a theory $Th(\mathcal{L}, r, q)$, where $\mathcal{L}$ is a set of sentences to consider, r the data(base), and q the quality criterion. The aim then is to find all sentences ϕ in the language $\mathcal{L}$ that satisfy the quality criterion w.r.t. the data r, i.e.

$$Th(\mathcal{L}, r, q) = \{\phi \in \mathcal{L} \mid q(r, \phi(r))\textit{is true}\}$$

Our formalisation of induction is a special case of Mannila's one, where $\mathcal{L}$ contains the clauses to consider, and the quality criterion q is true whenever the clause ϕ is valid on the data in r. This clearly shows that characteristic induction from interpretations is a real data mining task.

4. A clausal discovery engine

This section provides a detailed description of our clausal discovery engine.

4.1. A Clausal Discovery Algorithm

The key to arrive at a clausal discovery algorithm for characteristic induction from interpretations is the well-known property/definition of logical entailment.

Property 5 (Pruning) *Let G be a logical generalisation of S, i.e. $G \models S$. If an interpretation M is a model for G then M will also be a model of S.*

The contraposition states that if M is not a model for S then M will not be a model for any logical generalisation G of S. This contraposition shows that large parts of the search space can be pruned. Indeed, given an observation o and hypothesis H such that H is false in o, all logical specialisations of H will be false in o and can thus be pruned.

By now, we can apply classical machine learning principles to obtain an algorithm for characteristic induction from interpretations. First, machine learning principles state that induction is a search process through a partially ordered space induced by the generalisation relation, cf. (Mitchell, 1982). Second, machine learning systems typically search the space specific-to-general or general-to-specific. The question then arises as to which of these strategies is the most feasible one. Theoretically, there may however be a problem when searching (logically) specific-to-general as one should then start from the most specific hypothesis which could be an infinite one. Furthermore, it is well-known in machine learning that pruning parts of the search space is more reliable when working general-to-specific. Therefore, we will only consider general-to-specific search. Third, as characteristic induction aims at a logically maximally general hypothesis, it should not use a covering approach but rather an exhaustive search of the relevant parts of the search space.

In order to arrive at a general algorithm in Figure 1, we only need to define the search space and the operator for traversing it. In the remainder of this paper, we will use the notation $\mathcal{L}$ to denote the search space consisting of clauses, and a refinement operator ρ based on θ-subsumption (Plotkin, 1970) to traverse it.

Definition 4 *A refinement operator ρ (with transitive closure ρ^*) for a language $\mathcal{L}$ is a mapping from $\mathcal{L}$ to $2^{\mathcal{L}}$ such that*

1. $\forall c \in \mathcal{L} : \rho(c) \subset \{c' \in \mathcal{L} \mid c'$ *is a proper maximally general specialisation of* c *under* θ*-subsumption*$\}$*, and*
2. ρ *is complete, i.e.* $\rho^*(\Box) = \mathcal{L}$ *where* $\Box$ *is the most general element in* $\mathcal{L}$.

Completeness means that all elements of the language can be generated using ρ. In our framework, optimal refinement operators are the most desirable ones :

Definition 5 *A refinement operator ρ (with transitive closure ρ^*) is optimal if and only if* $\forall c, c_1, c_2 \in \mathcal{L} : c \in \rho^*(c_1)$ *and* $c \in \rho^*(c_2) \rightarrow c_1 \in \rho^*(c_2)$ *or* $c_2 \in \rho^*(c_1)$.

Optimal refinement operators are more efficient than classical refinement operators because they generate each candidate clause exactly once. A known problem with classical refinement operators is that they generate candidate clauses (and their refinements) more than once, making the search intractable. Optimality is thus desirable for efficiency reasons. (van der Laag & Nienhuys-Cheng, 1994) have shown that specific types of operators (such as optimal ones) do not exist for the infinite language of full clausal logic. However, for finite languages (which is the assumption in the implementation), optimal as well as complete operators do exist.

The algorithm in Figure 1 starts with an empty hypothesis H, and a queue Q containing only the most general element in the considered language $\mathcal{L}$. It then applies a search process where each element c is deleted from the queue Q, and tested for validity on the observations O. If the clause is valid, and not to be *pruned1* (see below), it is added to the hypothesis. If c is invalid, its refinements generated and those refinements which are not to be *pruned2* (see below) are added to the queue. When the queue is empty, the algorithm halts and outputs the current hypothesis.

```
function ClausalDiscovery
    inputs : $O$: set of Closed Observations, $\rho$: refinement operator
    outputs : Characteristic Hypothesis

$H$ := $\emptyset$
$Q$ := $\{\square\}$
while $Q \neq \emptyset$ do
    delete $c$ from $Q$
    if c is valid on O
        and not prune1(c)
    then add $c$ to $H$
    else for all $c' \in \rho(c)$ for which not prune2(c’) do
            add $c'$ to $Q$
        endfor
    endif
endwhile
reduce(H)
endfunction
```

Figure 1. A clausal discovery algorithm

The ClausalDiscovery algorithm has a number of parameters, which are printed in *italics*. They can be used to specify the many options of the clausal discovery engine. The *delete* function determines the search-strategy. When delete is first in first out one realizes breadth-first search, when it is last in first out then depth-first, when it is according to some ranking of the clauses, it is best-first. Different heuristics for ranking clauses are discussed in Section 4.6. The function *valid* determines when a clause is accepted as (part of) a solution. When coping with noisy data it is often useful to relax the validity requirements as detailed in Section 4.5. The functions *prune1*, *prune2* and *reduce* are meant to implement the options (including a special type of pruning when the language is fair), cf. Section 4.2. Most important is the language bias and corresponding refinement operator. The declarative language bias mechanism $\mathcal{D}$LAB and the corresponding refinement operators are discussed in Section 4.4. Finally, a parallel version of this algorithm is indicated in Section 4.3 and Appendix A.

4.2. Properties and Extensions

We first prove that the ClausalDiscovery engine is correct, and then discuss three extensions. The first extension allows to deal with infinite models, the second one concerns the options and the third one is an optimisation for *fair* languages.

4.2.1. Property

Ignoring for the moment the functions *prune1*, *prune2*, and *reduce*, which are used to implement the options (cf. below), it is easy to see that:

Property 6 *ClausalDiscovery outputs a maximally general valid hypothesis within $2^{\mathcal{L}}$ if it terminates and ρ is complete with regard to $\mathcal{L}$.*

Proof: If the algorithm would perform an exhaustive search of $\mathcal{L}$ and would add all valid clauses to H, the result trivially holds. Now, a clause c is only pruned when it is θ-subsumed by a valid clause $c' \in H$. Because c' logically entails c, H is as general as $H \wedge c$, implying that c may be pruned without losing information. □

4.2.2. Termination

The algorithm may not always terminate because of two reasons:

- the refinement graph searched may be infinite, which may lead the algorithm to exploring infinite paths through the search-space;
- testing whether a clause is valid on an observation using $Body \wedge \neg Head$ (as outlined above) is only semi-decidable in the general case.

The first problem can be avoided when working with finite Herbrand interpretations and using the injectivity assumption, or when using only finite languages. The second problem only arises when the Herbrand interpretation of an observation is infinite. Two approaches can be taken in this case. First, one can use an h-easy notion of validity (by setting the function *valid* accordingly).

Definition 6 (h-easy validity) *A clause c is h-easy valid on an observation o if and only if an SLDNF-interpreter (with depth-bound h) fails when answering the query ?-body(c), not head(c). on the knowledge base $B \cup o$.*

SLDNF-resolution is the basis of the logic programming language PROLOG, see (Lloyd, 1987) for more details. By employing a depth-bound on the depth of the proof tree, termination is guaranteed. However, soundness is lost in the following sense. If a clause is h-easy valid, it may be invalid in the logical sense. When employing h-easy validity, this may result in finding a logically inconsistent hypothesis $H \models \Box$, so care should be taken with this approach.

Second, one can approximate the infinite models by finite subsets of them, and one can then use a flattening approach (Rouveirol, 1994, De Raedt & Džeroski, 1994) to allow for clauses that have only infinite models. Since this approach is detailed in (De Raedt & Džeroski, 1994), we do not further elaborate on this here.

4.2.3. *Implementing the options*

Prune1 can be used to enforce maximally general clauses by removing all clauses c' that are not maximally general.

Prune2 can be used to enforce injectivity, non-triviality, and non-redundancy by removing all clauses c that are not injective, trivial or redundant.

Reduce can be used to enforce compactness, cf. (De Raedt & Bruynooghe, 1993). This involves the use of a theorem-prover. In the current implementation, SATCHMO by (Manthey & Bry, 1988) is employed.

4.2.4. *Fairness*

An important optimisation is possible in case the language considered is *fair* (cf. (De Raedt & Bruynooghe, 1993)).

Definition 7 *A language $\mathcal{L}$ is fair if and only if $\forall$ clauses A, B, C and $\forall$ substitutions θ, such that $A \in \mathcal{L}, A \vee B \in \mathcal{L}$ and $A\theta \vee B\theta \vee C \in \mathcal{L}$, we also have that $A\theta \vee C \in \mathcal{L}$.*

Let $A = \neg male(X)$, B = $\neg gorilla(X)$, C = $\neg tall(X)$, and θ = $\{\}$. Assume that all conditions are satisfied, i.e. $\neg male(X)$; $\neg male(X) \vee \neg gorilla(X)$; and $\neg male(X) \vee \neg gorilla(X) \vee \neg tall(X) \in \mathcal{L}$. Fairness then requires that $\neg male(X) \vee \neg tall(X) \in \mathcal{L}$.

If the language is fair, one can optimise the search using the following property by safely pruning away certain clauses.

Property 7 (Fairness) *Given a fair language $\mathcal{L}$, a set of observations O, a clause A, a refinement $A \vee B$ of A, and $B \rightarrow A$ is valid in O, ClausalDiscovery may prune2 $A \vee B$ as well as its refinements.*

Proof: We first prove that $\forall B$ and $\forall\theta : A\theta \vee C$ is valid in O if and only if $A\theta \vee B\theta \vee C$ is valid in O (0).

1. because $A\theta \vee C$ θ-subsumes $A\theta \vee B\theta \vee C$, $A\theta \vee C$ logically entails $A\theta \vee B\theta \vee C$. Therefore, if $A\theta \vee C$ is valid, $A\theta \vee B\theta \vee C$ is also valid.

2. Suppose now that $A\theta \vee B\theta \vee C$ is valid and $A\theta \vee C$ is invalid in O. (1)
   Then there is a substitution σ such that $(A\theta \vee C)\sigma$ is ground and false in some observation $o \in O$. Therefore $\neg A\theta\sigma \wedge \neg C\sigma$ is true in o. Hence $\neg A\theta\sigma$ is true in o. (2)
   It was given that $B \rightarrow A$ is true in O, therefore the contraposition $\neg A \rightarrow \neg B$ is also true in o. From this and (2) it follows that $\neg B\theta\sigma$ is true in o.
   Therefore $A\theta \vee B\theta \vee C$ is false in o as there is a substitution σ for which it is false. This contradicts (1) and concludes the proof of (0).

From (0) it follows that $A \vee B$ is valid if and only if A is valid (choose C = $\{\}$ and θ = $\{\}$ in (0)). Now, if A is valid (and part of the hypothesis), $A \vee B$ need not be part of the final hypothesis (because it is logically entailed by A and hence redundant if A is added to the hypothesis). If A is invalid, then $A \vee B$ is invalid (hence $A \vee B$ should not be part of the final hypothesis). This shows that $A \vee B$ need not be part of the final hypothesis.

We still have to show that it is safe to also prune the refinements of $A \vee B$. First note that all refinements of $A \vee B$ (under θ-subsumption) are of the form $A\theta \vee B\theta \vee C$. From (0), it then follows that $A\theta \vee C$ is valid if and only if $A\theta \vee B\theta \vee C$ is valid, hence the two clauses are equivalent w.r.t. validity. Because of fairness, $A\theta \vee C$ will be considered by ClausalDiscovery. Hence, it is safe to prune $A\theta \vee B\theta \vee C$.

■

To illustrate the property, reconsider the example above. Assume now also that $gorilla(X) \rightarrow male(X)$ is valid. The property then states that it is safe to prune $\neg gorilla(X) \vee \neg male(X)$, and its refinements such as $\neg gorilla(X) \vee \neg male(X) \vee \neg tall(X)$ as equivalent clauses (w.r.t. validity) such as $\neg gorilla(X) \vee \neg tall(X)$ will be considered because of validity. More examples of fair and unfair languages are given in Section 4.4 on declarative language bias.

4.3. Parallellism

Due to the monotonicity property of our induction framework, it is relatively easy to parallellize the ClausalDiscovery engine. ClausalDiscovery essentially traverses the space of clauses exhaustively and general-to-specific. This yields a search-tree in which the nodes are clauses, and there is a subtree of a clause for each refinement (under the operator ρ) of the clause. Now, due to monotonicity all subtrees of the search-tree can be processed independently of each other and therefore in parallel. The resulting algorithm is presented in Appendix A.

4.4. Declarative language bias

Even if we choose the search space $\mathcal{L}$ to be finite, it is in most cases impractical to define $\mathcal{L}$ extensionally. We then need a formalism to formulate an intensional syntactic definition of language $\mathcal{L}$.

The problem of making this type of syntactic bias a parameter to the learning or discovering engine has been studied extensively, especially in frameworks that use first-order clausal logic (see (Muggleton & De Raedt, 1994, Adé *et al.*, 1995) for an overview). For $\mathcal{C}$LAUDIEN we developed a new formalism called $\mathcal{D}$LAB (Declarative LAnguage Bias)[6]. $\mathcal{D}$LAB extends the syntactic bias of (Adé *et al.*, 1995) which in turn integrates the schemata of (Emde *et al.*, 1983, Kietz & Wrobel, 1992), and the predicate sets of (Bergadano & Gunetti, 1993, Bergadano, 1993). When compared to Cohen's antecedent description grammars (Cohen, 1994), $\mathcal{D}$LAB is a special case where the definite clause grammar is fixed and hidden. This grammar takes the $\mathcal{D}$LAB formula as its single argument. In that sense $\mathcal{D}$LAB is a higher order formalism based on the lower order antecedent description grammar.

We present an overview of $\mathcal{D}$LAB in two stages. First, we discuss syntax, semantics and a refinement operator for $\mathcal{D}\text{LAB}^{\ominus}$, a subset of $\mathcal{D}$LAB. We then extend $\mathcal{D}\text{LAB}^{\ominus}$ to full $\mathcal{D}$LAB. An earlier version of this section appeared in (Dehaspe & De Raedt, 1996).

4.4.1. $\mathcal{D}\text{LAB}^{\ominus}$

A $\mathcal{D}\text{LAB}^{\ominus}$ grammar is a finite set of templates to which the clauses in search space $\mathcal{L}$ conform. We first give a recursive syntactic definition of the $\mathcal{D}\text{LAB}^{\ominus}$ formalism.

Definition 8 ($\mathcal{D}\text{LAB}^{\ominus}$ syntax)

1. *a $\mathcal{D}\text{LAB}^{\ominus}$ atom is either a logical atom, or of the form $Min \cdot\cdot Max : L$, with Min and Max integers such that $0 \leq Min \leq Max \leq length(L)$, and with L a list of $\mathcal{D}\text{LAB}^{\ominus}$ atoms;*
2. *a $\mathcal{D}\text{LAB}^{\ominus}$ template is of the form $A \leftarrow B$, where A and B are $\mathcal{D}\text{LAB}^{\ominus}$ atoms;*
3. *a $\mathcal{D}\text{LAB}^{\ominus}$ grammar is a set of $\mathcal{D}\text{LAB}^{\ominus}$ templates.*

The following are a few examples of syntactically well-formed $\mathcal{D}\text{LAB}^{\ominus}$ grammars:

- $\{say(Hello) \leftarrow to\_world\}$
- $\{false \leftarrow 0 \cdot\cdot 2 : [male(X), female(X)]\}$
- $\{2 \cdot\cdot 2 : [a(X), b(Y)] \leftarrow 1 \cdot\cdot 2 : [c(X), 0 \cdot\cdot 1 : [d(Y)]],$
  $0 \cdot\cdot 1 : [n, 1 \cdot\cdot 2 : [o, 1 \cdot\cdot 1 : [p, q], r], s] \leftarrow true\}$

The hypothesis space that corresponds to a $\mathcal{D}\text{LAB}^{\ominus}$ grammar is then constructed via the (recursive) selection of all sublists of L with length within range $Min \ldots Max$ from each $\mathcal{D}\text{LAB}^{\ominus}$ atom $Min \cdot\cdot Max : L$. This idea can be elegantly formalised and implemented using the Definite Clause Grammar (DCG) notation, which is an extension of PROLOG (cf. (Clocksin & Mellish, 1981, Sterling & Shapiro, 1986))[7].

Definition 9 ($\mathcal{D}\text{LAB}^{\ominus}$ semantics) *Let $\mathcal{G}$ be a $\mathcal{D}\text{LAB}^{\ominus}$ grammar, then*

$$dlab\_generate(\mathcal{G}) = \{dlab\_dcg(A) \leftarrow dlab\_dcg(B) | (A \leftarrow B) \in \mathcal{G}\}$$

generates all clauses in the corresponding hypothesis space, where $dlab\_dcg(E)$ is a list of logical atoms generated by $dlab\_dcg$:

$$\begin{aligned}
dlab\_dcg(E) &\longrightarrow [E], \{E \neq Min \cdot\cdot Max : L\}. && (1)\\
dlab\_dcg(Min \cdot\cdot Max : []) &\longrightarrow \{Min \leq 0\}, []. && (2)\\
dlab\_dcg(Min \cdot\cdot Max : [\_|L]) &\longrightarrow dlab\_dcg(Min \cdot\cdot Max : L). && (3)\\
dlab\_dcg(Min \cdot\cdot Max : [E|L]) &\longrightarrow \{Max > 0\}, dlab\_dcg(E),\\
&\qquad dlab\_dcg((Min-1) \cdot\cdot (Max-1) : L). && (4)
\end{aligned}$$

From the semantics of a $\mathcal{D}\text{LAB}^{\ominus}$ grammar we derive a formula for calculating the size of its hypothesis space.

Property 8 ($\mathcal{D}\textsc{lab}^{\ominus}$ size) *Let $\mathcal{G} = \{A_1 \leftarrow B_1, \ldots, A_m \leftarrow B_m\}$ be a $\mathcal{D}\textsc{lab}^{\ominus}$ grammar, then the size of the corresponding hypothesis space equals $dlab\_size(G)$, with*

$$
\begin{array}{l}
dlab\_size(\mathcal{G}) = \sum_{i=1}^{m} (ds(A_i) * ds(B_i)) \ \ ; \\
ds(E) = 1, \textit{where } E \textit{ is a logical atom} \ \ ; \\
ds(Min \cdot\cdot Max : [L_1, \ldots, L_n]) = \sum_{k=Min}^{Max} e_k(ds(L_1), \ldots, ds(L_n)) \ \ ; \\
e_0(s_1, \ldots, s_n) = 1 \ \ ; \\
e_n(s_1, \ldots, s_n) = \prod_{i=1}^{n} s_i \ \ ; \\
e_k(s_1, s_2, \ldots, s_n) = e_k(s_2, \ldots, s_n) + s_1 * e_{k-1}(s_2, \ldots, s_n), \textit{with } k < n \ \ .
\end{array}
$$

Proof: The first rule states that the size of the language defined by a $\mathcal{D}\textsc{lab}^{\ominus}$ grammar equals the sum of the sizes of the languages defined by its individual $\mathcal{D}\textsc{lab}^{\ominus}$ templates. The latter size can be found by multiplying the number of headlists and the number of bodylists covered by the head and body $\mathcal{D}\textsc{lab}^{\ominus}$ atoms.

A $\mathcal{D}\textsc{lab}^{\ominus}$ atom which is not of the form $Min \cdot\cdot Max : L$ has a coverage of exactly one, as is expressed in the second rule.

Some more intricate combinatorics underlies the third rule. Basically, we select k objects from $\{L_1, \ldots, L_n\}$, for each k in range $Min \ldots Max$, hence the summation $\sum_{k=Min}^{Max}$. Inside this summation we would have the standard formula $n!/k! * (n-k)!$ if our case had been an instance of the prototypical problem of finding all combinations, without replacement, of k marbles out of an urn with n marbles. This formula does not apply due to the fact that we rather have n urns ($\{L_1, \ldots, L_n\}$) with one or more marbles ($ds(L_i) \geq 1$), and only combinations that use at most one marble from each urn should be counted. Therefore we need $e_k(s_1, \ldots, s_n)$, where e_k is the elementary symmetric function (MacDonald, 1979) of degree k and the s_i are the numbers of marbles in each urn. The first base case of this recursive function accounts for the fact that there is only one way to select 0 objects. In the second base case, where $k = n$, one has to take an object from each urn. As for each urn there are s_i choices, the number of combinations equals the product of all s_i. The final recursive case applies if $k < n$. It is an addition of two terms, one for each possible operation on urn 1 (represented by s_1). Either we skip this urn, and then we still have to select k elements from urns 2 to n. The number of such combinations is given by $e_k(s_2, \ldots, s_n)$. Or else we do take a marble from the first urn. We then have to multiply s_1, the choices for the first urn, with $e_{k-1}(s_2, \ldots, s_n)$, the number of $k-1$ order combinations of elements from urns 2 to n.

■

Given a $\mathcal{D}\textsc{lab}^{\ominus}$ atom $Min \cdot\cdot Max : L$, four choices of values for Min and Max determine the following cases of special interest[8]:

1. **all sublists**: $Min = 0, Max = len$
   e. g. $\mathcal{G}1 = \{h \leftarrow 0 \cdot\cdot len : [a, b, c]\}$

2. **all non-empty sublists**: $Min = 1, Max = 1$
   e. g. $\mathcal{G}2 = \{h \leftarrow 1 \cdot\cdot len : [a, b, c]\}$

3. **exclusive or**: $Min = 1, Max = 1$
   e. g. $\mathcal{G}3 = \{h \leftarrow 1 \cdot\cdot 1 : [a, b, c]\}$

4. **combined occurence**: $Min = Max = len$
   e. g. $\mathcal{G}4 = \{h \leftarrow len \cdot\cdot len : [a, b, c]\}$

These special cases can be nested to construct more complex grammars exemplified below.

$$\begin{array}{l}
\mathcal{G}5 = \{h \leftarrow 1 \cdot\cdot len : [a, 1 \cdot\cdot 1 : [b, c]]\} \\
\mathcal{G}6 = \{h \leftarrow 1 \cdot\cdot len : [a, len \cdot\cdot len : [b, c]]\} \\
\mathcal{G}7 = \{h \leftarrow len \cdot\cdot len : [a, 1 \cdot\cdot 1 : [b, c]]\} \\
\mathcal{G}8 = \{h \leftarrow 0 \cdot\cdot len : [len \cdot\cdot len : [a, 0 \cdot\cdot len : [len \cdot\cdot len : [b, 0 \cdot\cdot len : [c]]]]]\}
\end{array}$$

Table 1 gives the corresponding hypothesis spaces for grammars $\mathcal{G}1 - \mathcal{G}8$. A $\surd$ in the column of grammar $\mathcal{G}i$ marks the clauses of the first column that are in the corresponding hypothesis space.

Except for $\mathcal{G}8$, all grammars in Table 1 define fair languages (see Definition 7). Grammar $\mathcal{G}8$ illustrates how taxonomies can be encoded, such that each atomic formula necessarily co-occurs with *all* its ancestors and never combines with other nodes. In the case of $\mathcal{G}8$, c only co-occurs with its both ancestors a, b. It is the exlusion of the combination of an atomic formula with a strict subset of ancestors (a, c in our example) which causes the definition of fairness to be violated. A more elaborate example is grammar $\mathcal{G}9$, which encodes the taxonomy for suits of playing cards:

$$\begin{array}{l}
\mathcal{G}9 = \{ok(C) \leftarrow \\
\quad len \cdot\cdot len : [card(C), \\
\qquad\qquad 0 \cdot\cdot 1 : [len \cdot\cdot len : [red(C), 0 \cdot\cdot 1 : [hearts(X), diamonds(C)]], \\
\qquad\qquad\qquad len \cdot\cdot len : [black(C), 0 \cdot\cdot 1 : [clubs(X), spades(C)]], \\
\qquad\qquad] \qquad]\}
\end{array}$$

$$dlab\_generate(\mathcal{G}9) = \begin{cases}
[ok(C)] \leftarrow [card(C)] \\
[ok(C)] \leftarrow [card(C), red(C)] \\
[ok(C)] \leftarrow [card(C), red(C), hearts(C)] \\
[ok(C)] \leftarrow [card(C), red(C), diamonds(C)] \\
[ok(C)] \leftarrow [card(C), black(C)] \\
[ok(C)] \leftarrow [card(C), black(C), clubs(C)] \\
[ok(C)] \leftarrow [card(C), black(C), spades(C)]
\end{cases}$$

Table 1. The semantics of some sample $\mathcal{D}$LAB grammars

	$\mathcal{G}1$	$\mathcal{G}2$	$\mathcal{G}3$	$\mathcal{G}4$	$\mathcal{G}5$	$\mathcal{G}6$	$\mathcal{G}7$	$\mathcal{G}8$
$[h] \leftarrow []$	$\surd$							$\surd$
$[h] \leftarrow [a]$	$\surd$	$\surd$	$\surd$		$\surd$	$\surd$		$\surd$
$[h] \leftarrow [b]$	$\surd$	$\surd$	$\surd$		$\surd$			
$[h] \leftarrow [c]$	$\surd$	$\surd$	$\surd$		$\surd$			
$[h] \leftarrow [a, b]$	$\surd$	$\surd$			$\surd$		$\surd$	$\surd$
$[h] \leftarrow [a, c]$	$\surd$	$\surd$			$\surd$		$\surd$	
$[h] \leftarrow [b, c]$	$\surd$	$\surd$				$\surd$		
$[h] \leftarrow [a, b, c]$	$\surd$	$\surd$		$\surd$		$\surd$		$\surd$

In Appendix B, we show how a refinement operator for a $\mathcal{D}$LAB$^\ominus$ language can be obtained from the $\mathcal{D}$LAB$^\ominus$ grammar. Furthermore, Appendix B touches upon some of the key implementation aspects of the $\mathcal{C}$LAUDIEN engine.

4.4.2. $\mathcal{D}$LAB$^\ominus$ Extended: $\mathcal{D}$LAB

In an extended version $\mathcal{D}$LAB mainly two features have been added to improve readability of more complex grammars: second order variables, and sublists on the term level.

Definition 10 ($\mathcal{D}$LAB syntax)

1. *a $\mathcal{D}$LAB term is either*
   - *(A) a variable symbol, or*
   - *(B) of the form $f(t_1, \ldots, t_n)$, where f is a function symbol followed by a bracketed $n - tuple$ $(0 \leq n)$ of $\mathcal{D}$LAB terms t_i, or*
   - *(C) of the form $Min \cdot\cdot Max : L$, where Min and Max are integers with $0 \leq Min \leq Max \leq length(L)$, and with L a list of $\mathcal{D}$LAB terms;*
2. *a $\mathcal{D}$LAB atom is either*
   - *(A) of the form $p(t_1, \ldots, t_n)$, where p is a predicate symbol followed by a bracketed $n - tuple$ $(0 \leq n)$ of $\mathcal{D}$LAB terms t_i, or*
   - *(B) of the form $Min \cdot\cdot Max : L$, where Min and Max are integers with $0 \leq Min \leq Max \leq length(L)$, and with L a list of $\mathcal{D}$LAB atoms;*
3. *a $\mathcal{D}$LAB template is of the form $A \leftarrow B$, where A and B are $\mathcal{D}$LAB atoms;*
4. *a $\mathcal{D}$LAB variable is of the form $dlab\_var(p_0, Min \cdot\cdot Max, [p_1, \ldots, p_n])$, where Min and Max are integers with $0 \leq Min \leq Max \leq n$, and with p_i a predicate symbol or a function symbol*
5. *a $\mathcal{D}$LAB grammar is a couple $(\mathcal{T}, \mathcal{V})$, where $\mathcal{T}$ is a set of $\mathcal{D}$LAB templates, and $\mathcal{V}$ a set of $\mathcal{D}$LAB variables.*

We will now define the conversion of $\mathcal{D}$LAB grammars $(\mathcal{T}, \mathcal{V})$ to the $\mathcal{D}$LAB$^\ominus$ format such that the above definitions of semantics, size, and a refinement operator remain valid for the enriched formalism. First, to remove the second order variables $\mathcal{V}$ we recursively replace all $\mathcal{D}$LAB terms and atoms

$p(t_1, \ldots, t_n)$ in $\mathcal{T}$ such that $dlab\_var(p, Min \cdot\cdot Max, [p_1, \ldots, p_m]) \in \mathcal{V},$ with
$Min \cdot\cdot Max : [p_1(t_1, \ldots, t_n), \ldots, p_m(t_1, \ldots, t_n)]$.

Next we recursively remove sublists on the termlevel by replacing from left to right all $\mathcal{D}$LAB terms

$$p(t_1, \ldots, t_i, Min \cdot\cdot Max : [L_1, \ldots, L_n], t_{i+2}, \ldots, t_m), \text{with}$$
$$Min \cdot\cdot Max : [p(t_1, \ldots, t_i, L_1, t_{i+2}, \ldots, t_m), \ldots, p(t_1, \ldots, t_i, L_n, t_{i+2}, \ldots, t_m)] \enspace .$$

When applied subsequently, these two algorithms transform a $\mathcal{D}$LAB grammar $\mathcal{G} = (\mathcal{T}, \mathcal{V})$ into $(\mathcal{G}', \emptyset)$, where $\mathcal{G}'$ is an equivalent $\mathcal{D}\text{LAB}^{\ominus}$ grammar.

For a demonstration of the power of $\mathcal{D}\text{LAB}^{\ominus}$ and $\mathcal{D}$LAB we refer to the experiments in Section 5.

4.5. Quantifying Validity

There are at least three reasons why the *logical* validity requirement should be quantified and sometimes relaxed. First, when coping with real data, it is an illusion to find rules that are valid on all of the observations. The same situation arises in discriminant induction when trying to discriminate two classes of observations. As very often complete and consistent hypotheses do not exist, discriminant induction allows to relax the completeness and consistency requirements. It is therefore also of practical interest to see how the validity requirement of characteristic induction from interpretations can be relaxed. This corresponds to relaxing the q in Mannila's definition. Secondly, a quantified notion of validity will also be useful to label the induced clauses, and to rank them according to validity. Such a ranking is essential for expert evaluation and post-processing of discovered rules. Thirdly, quantified notions of validity may turn out useful for heuristically searching the space, cf. Section 4.6.

There are two natural ways to quantify validity. For the first one we introduce the concept of non-trivial observations. The set $O' \subset O$ of non-trivial observations contains all observations for which clause c is non-trivial (cf. non-triviality assumption in Section 3.2). We can then relax the condition that clauses in hypotheses are valid on *all* observations, and rather require validity on a certain percentage of all non-trivial observations. This can be realized by setting $GA(c)$ larger than a fixed percentage.

Definition 11 *(Global Accuracy) Let c be a clause, let O' be the non-trivial observations for c, let $pg(c)$ be the number of observations in O' which are a model for c, let $ng(c)$ be the number of observations in O' which are not a model for c. Then $GA(c)$, the global accuracy of the clause c, is $pg(c)/(pg(c) + ng(c))$.*

Global accuracy still requires that the clause is completely true on a number of observations. When the observations are incomplete, even global accuracy will be hard to obtain. Furthermore, there is the special case of the framework, where only a single observation is taken into account. This special case is important in a data mining context, as one often deals with a single interpretation (in which various observations are mixed). Local accuracy, which measures the degree to which a clause is true in an interpretation may offer a solution in this case. Local accuracy employs the notions of positive and negative substitutions.

We first introduce the notions of positive and negative substitutions of a clause.

Definition 12 (Positive and Negative Substitutions) *θ is a positive substitution for a clause $p_1, ..., p_m \leftarrow q_1, ..., q_n$ with $m > 0$, and observations O, if and only if 1) $(p_1, ..., p_m \leftarrow q_1, ..., q_n)\theta$ is ground, 2) there exists an observation $o_i \in O$ such that (a) $(q_1 \wedge ... \wedge q_n)\theta$ is true and ground in o_i, and (b) $(p_1 \vee ... \vee p_m)\theta$ is true in o_i.*
θ is a negative substitution if and only if it satisfies (1) and (2a) and does not satisfy (2b).

This definition should only be applied when the clause is range-restricted. From a practical point of view, there are often problems when merely counting substitutions because there is no direct correspondence guaranteed between what is being counted (substitutions) and the entities the clause deals with (e.g. birds, or meshes, or molecules, ...). Secondly, the above definition will result in problems when applying it to denials (i.e. clauses of the form $\leftarrow q_1, ..., q_n$). Therefore it is often convenient to transform a clause

$$p_1, ..., p_m \leftarrow q_1, ..., q_n$$

where all p_i, q_j are logical atoms, into the following logically equivalent form

$$p_1, ..., p_m, \neg q_{i+1}, ..., \neg q_n \leftarrow q_1, ..., q_i$$

before constructing positive and negative substitutions. The positive and negative substitutions of the two clauses will not necessarily be the same. However, by appropriately choosing the literals $q_1, ..., q_i$ it is possible that meaningful entities are counted. In the $\mathcal{C}$LAUDIEN implementation, the user is offered the possibility of specifying which literals to consider in the body of the clause and which ones in the head, when considering positive and negative substitutions.

By now we can define local accuracy.

Definition 13 *(Local Accuracy) Let c be a clause, let O be the observations considered, let $pl(c)$ be the number of positive substitutions for c, let $nl(c)$ be the number of negative substitutions for c. Then $LA(c)$, the local accuracy of the clause c, is $pl(c)/(nl(c)+pl(c))$.*

Again, validity can be relaxed by setting $LA(c)$ larger than a fixed percentage.

In data mining, one often labels the induced rules with information indicating accuracy of the rule and in how many cases it applies, i.e. the coverage. The above notions of accuracy are useful as an accuracy label of clauses. The following notions of global and local coverage will be used as coverage labels of clauses.

Definition 14 *(Global Coverage) Let O' be the non-trivial observations for c, let $pg(c)$ and $ng(c)$ be computed w.r.t. the observations O'. Then the global coverage of a clause $GC(c) = pg(c) + ng(c)$.*

The reason for restricting the attention to those observations for which the clause is non-trivial is that otherwise all clauses will have a global coverage equal to the number of observations. When applying global coverage to valid denials, the coverage will be 0, by Definitions 3.2.1 and 14 of non-triviality and global coverage. Therefore, in that case one should first apply the clause transformation introduced above.

Definition 15 *(Local Coverage) The local coverage of a clause* $LC(c) = pl(c) + nl(c)$.

The notions accuracy and coverage are related to the confidence and support thresholds used in the literature on discovery of association rules in large databases(Agrawal *et al.*, 1993).

4.6. Heuristics

Discriminant approaches employ various types of heuristics to guide the search towards those clauses that best discriminate the positive from the negative examples, or to prune clauses from the search space. Various heuristics have been proposed, e.g. information content (Quinlan, 1990), minimal length description (Srinivasan *et al.*, 1992), accuracy estimates (Lavrač & Džeroski, 1994), etc.

Our induction framework can easily adapt these heuristics using the measures of validity defined in the previous subsection. More specifically, whereas discriminant induction heuristics are based on the proportions of positive and negative examples, clausal discovery can use the notions of positive and negative substitutions pl and nl, or alternatively, the number of positive and negative observations pg and ng. Given a clause c, a set of observations O, and a background theory, one can now basically employ all favourite heuristics. One only has to substitute our numbers in the well-known formulae. This procedure works for evaluating clauses as well as for evaluating refinement steps. An example of a the first type of heuristic is accuracy, and of the second type of heuristic, entropy as applied in FOIL (Quinlan, 1990). Many other heuristics are known in the literature, for an overview see (Lavrač & Džeroski, 1994) and (Klösgen, 1996).

As clausal discovery aims at a maximally general hypothesis, and the number of clauses in such a maximally general hypothesis may be very large, characteristic induction procedures should try to discover as many interesting clauses as possible using a limited amount of resources. Indeed, as resources are always limited (one cannot search forever), clausal discovery heuristics should employ heuristics of the first type, focusing on the most interesting clauses first. Using heuristics and limited resources (whether time or space), certain unpromising parts of the search space may not be considered. This leads to the view that characteristic induction procedures should be *any time* algorithms, i.e. algorithms that are able to find approximate solutions in any time, and improve upon those (by discovering more clauses) when more resources are available.

In the experiments with the $\mathcal{C}$LAUDIEN system we will mainly employ the following heuristic (based on the minimal description length principle): $p/(l + n)$ where p accounts for the positive substitutions or interpretations, n for the negative ones, and l is the clause length, computed as the number of literals in the clause tested. The heuristic is then combined with the local or global measures provided earlier. It is merely used to order the clauses on the queue, implementing an any time algorithm. Though the heuristic works fine in practice, it is unclear whether it is the most adequate one. Other well-known heuristics from the data mining paradigm could also be employed (cf. (Klösgen, 1996)).

5. Applications of Clausal Discovery

The distinction between characteristic and discriminant induction discussed in Section 3 cascades to the level of the presentation of experimental results. For discriminating learners there is a standard two-phased assessment method in which classification rules learnt in a training stage are tested on (unseen) data. The quality of the system is typically associated with the percentage of successful class predictions. The domain of clausal discovery (as well as data mining in general) lacks such a clear cut evaluation criterion. The main goal is to discover *interesting* properties, but *interestingness* is in general hard to quantify, subjective and dated. Even worse, contrary to classification accuracy, which is based on elementary statistics, it can only be judged upon by an expert in the application domain.

An alternative evaluation criterion for discovery systems is then based on the iterative nature of the knowledge discovery process. Feedback from the domain expert will often trigger new, slightly altered experiments. Discovery systems that are highly tunable and versatile are better prepared to take this kind of feedback into account, and thus are *more likely* to produce interesting output in the end. Our aim in this section is then to give a flavour of the tunability and versatility of $\mathcal{C}$LAUDIEN. We will demonstrate how $\mathcal{C}$LAUDIEN can solve different discovery tasks, and how the system can be tuned to discover different types of rules in the same dataset. All tests were done on a SPARCserver1000.

5.1. Clausal discovery for data mining

One of the popular subjects in the field of knowledge discovery in databases is to induce large sets of rules of a particular type or syntax, cf. Mannila's definition of data mining in Section 3.2.3. The types of rules considered include: functional and multivalued dependencies (see e.g. (Flach, 1993, Savnik & Flach, 1993, Kantola *et al.*, 1992)), determinations (see e.g. (Schlimmer, 1991, Shen, 1992)), association rules (cf. (Agrawal *et al.*, 1993)), and strong rules (cf. (Piatetsky-Shapiro, 1991)). Various special purpose algorithms have been developed to handle the different types of rules. However, it turns out that because of the expressiveness of first order logic and the $\mathcal{D}$LAB formalism of $\mathcal{C}$LAUDIEN, many of the tasks performed by these special purpose algorithms can be reformulated in terms of the $\mathcal{C}$LAUDIEN framework. As a consequence, the task performed by these algorithms is a special case of that performed by $\mathcal{C}$LAUDIEN.

Let us first provide evidence for this claim, and then discuss its implications and restrictions.

We start by showing how $\mathcal{C}$LAUDIEN can induce functional and multi-valued dependencies on an example that is due to Flach (Flach, 1993). We ran $\mathcal{C}$LAUDIEN on the following data from Flach (the term *train*$(From, Hour, Min, To)$ denotes that there is a train from $From$ to To at time $Hour, Min$):

train(utrecht,8,8,den-bosch)
train(maastricht,8,10,weert)
train(utrecht,9,8,den-bosch)
train(maastricht,9,10,weert)
train(utrecht,8,13,eindhoven-bkln)
train(utrecht,8,43,eindhoven-bkln)
train(utrecht,9,13,eindhoven-bkln)
train(utrecht,9,43,eindhoven-bkln)
train(utrecht,8,31,utrecht)
train(tilburg,8,10,tilburg)
train(utrecht,8,25,den-bosch)
train(tilburg,9,10,tilburg)
train(utrecht,9,25,den-bosch)
train(tilburg,8,17,eindhoven-bkln)
train(tilburg,8,47,eindhoven-bkln)
train(tilburg,9,17,eindhoven-bkln)
train(tilburg,9,47,eindhoven-bkln)

using $\mathcal{D}$LAB grammar $(train\_temps, \emptyset)$:

```
train_temps = {1-1 : [From1 = From2, Hour1 = Hour2, Min1 = Min2, To1 = To2]
                 <--
               len-len : [train(From1,Hour1,Min1,To1,Plat1),
                          train(From2,Hour2,Min2,To2,Plat2),
                          0-len:[From1 = From2, Hour1 = Hour2,
                                 Min1 = Min2, To1 = To2]
                         ]
              }
```

$\mathcal{C}$LAUDIEN found (as Flach's INDEX) the following two dependencies:

```
From1 = From2 <-- train(From1,Hour1,Min1,To1),train(From2,Hour2,Min2,To2),
                  To1=To2,Min1=Min2
From1 = From2 <-- train(From1,Hour1,Min1,To1),train(From2,Hour2,Min2,To2),
                  From1=From2,Min1=Min2
```

It is straightforward to write $\mathcal{D}$LAB statements that would find only determinations of the form $P(X,Y) \leftarrow Q(X,Z), R(Z,Y)$ (as (Shen, 1992)), determinations as (Schlimmer, 1991) and multivalued dependencies as in (Flach, 1993).

Very popular in the data mining literature are association rules. Association rules are defined over a single relation composed of a set of attributes R over the binary domain $\{0,1\}$. An association rule is then of the form $X \Rightarrow Y$ where $X \subset R$ and $Y \subset (R - X)$. Typically, one is interested in all association rules c for which $LA(c) > \sigma$ and $LC(c) > \gamma$, for a certain threshold. Using local validity and the following type of $\mathcal{D}$LAB declaration, $\mathcal{C}$LAUDIEN would also solve the problem of finding association rules. The $\mathcal{D}$LAB declaration $(assoc\_temps, assoc\_vars)$ assumes that the relation under consideration is r with arity n, '=' denotes unification, and further that each attribute can have only two values: 0 and 1. The statement can be trivially generalized when an attribute can have more or other values.

```
assoc_temps = {
  {(X1, ..., Xn) = (Y1, ... ,Yn)
   <--
   len-len:
     [r(X1, ... , Xn),
      1-1:[len-len:[Y1 = bit,
                    0-len:[1-1:[X2,Y2],1-1:[X3,Y3],...,1-1:[Xn,Yn]] = bit
                   ]
           len-len:[Y2 = bit,
                    0-len:[1-1:[X1,Y1],1-1:[X3,Y3],...,1-1:[Xn,Yn]] = bit
                   ]
           ...
           len-len:[Yn = bit,
                    0-len:[1-1:[X1,Y1],1-1:[X2,Y3],...,1-1:[Xn-1,Yn-1]] = bit
                   ]
     ]
  }

assoc_vars = {dlab_variable(bit, 1-1, [0,1]}
```

The $\mathcal{D}$LAB statement will allow at most one literal per attribute in the body of the clause. If the literal is of the form X=value, then it occurs in the X part of the association rule $X \Rightarrow Y$, otherwise in the Y part. A clause generated by this $\mathcal{D}$LAB grammar could be e.g. $(X1, X2, X3, X4) = (Y1, Y2, Y3, Y4) \leftarrow r(X1, X2, X3, X4), X1 = 0,$ $Y2 = 1, Y4 = 0$ denoting the association rule $X1 = 0 \Rightarrow Y2 = 1 \wedge Y4 = 0$.

Strong rules (Piatetsky-Shapiro, 1991) can be defined in a similar way. Facilities offered by $\mathcal{C}$LAUDIEN to prune potentially large sets of association rules include:

- increase the $LA(c)$ threshold
- increase the $LC(c)$ threshold
- make the $\mathcal{D}$LAB template more specific

These examples clearly illustrate that $\mathcal{C}$LAUDIEN can perform many of the tasks addressed in the data mining literature. We therefore believe that $\mathcal{C}$LAUDIEN should be considered as a general purpose data mining environment and framework, which can be used for reasoning about and experimenting with various data mining problems. Of course, data mining research has always aimed at coping with large data sets in an efficient way, leading to very fast algorithms. As there is a general trade-off between generality of systems and their efficiency, $\mathcal{C}$LAUDIEN cannot be expected to solve the above data mining problems as efficient as the best data mining algorithms. Nevertheless, we believe (and the other experiments in this section confirm our belief) that $\mathcal{C}$LAUDIEN is reasonably efficient and can cope with reasonably large data sets. Furthermore, though data mining has focused on handling large data sets, inductive logic programming has focused on searching large hypotheses spaces.

5.2. Recovering program loop invariants

A standard method for the design and development of program loops is based on the list of relations between variable values which remain invariant during the repetition. Such a list of invariant relations fully captures the behaviour of loops and as such provides a key to their understanding and to proving their correctness. We here demonstrate how $\mathcal{C}$LAUDIEN can recover this type of specifications from program traces (see also (Bratko & Grobelnik, 1993)).

```
**function** Product
    **inputs :** $x, y$: positive integers,
    **outputs :** $z$: the product of $x$ and $y$

$z$ := 0 ; $u$ := $x$ ; $v$ := $y$ ;
**while** † $(u \neq 0)$ **do**
    **if** $odd(u)$ **then** $z$ := $z + v$;
    $u$ := $u$ **div** 2;
    $v$ := $2 * v$
**endwhile**
**return** $z$
**endfunction**
```

Figure 2. An algorithm for calculating the product of two positive integers

To generate data for this experiment we ran the algorithm in Figure 2 121 times, with inputs x, y varying between 0 and 10. During each run we recorded at each iteration the values of z, u, v at position † preceding the test $(u \neq 0)$ of the loop. We thus produced 121 observations with a single fact $input(x(X), y(Y))$ and a varying number of facts $trace(z(Z), u(U), v(V))$. A sample of these observations is given in Table 2.

Table 2. Sample observations in the invariant relations application

observation 1	observation 2	observation 3
$input(x(0), y(0))$ $trace(z(0), u(0), v(0))$	$input(x(7), y(6))$ $trace(z(0), u(7), v(6))$ $trace(z(6), u(3), v(12))$ $trace(z(18), u(1), v(24))$ $trace(z(42), u(0), v(48))$	$input(x(9), y(10))$ $trace(z(0), u(9), v(10))$ $trace(z(10), u(4), v(20))$ $trace(z(10), u(2), v(40))$ $trace(z(10), u(1), v(80))$ $trace(z(90), u(0), v(160))$

With the $\mathcal{D}$LAB grammar (ir_temps, ir_vars) shown in Figure 3, $\mathcal{C}$LAUDIEN discovered the following two invariant relations:

```
U >= 0 <-- input(x(X),y(Y)), trace(z(Z),u(U),v(V))
Term = XY <-- input(x(X),y(Y)), trace(z(Z),u(U),v(V)),
                XY is X * Y, _Term is Z + U * V
```

```
ir_temps = {0-1:[compare(U, 0), _Term =  XY]
            <--
            len-len:[input(x(X), y(Y)),
                     trace(z(Z), u(U), v(V)),
                     0-len:[XY is X * Y,
                             1-1:[Term is Z + U,
                                  Term is Z + V,
                                  Term is Z + U + V,
                                  Term is Z * U + V,
                                  Term is Z + U * V,
                                  Term is Z * V + U,
                                  Term is Z * U * V
                     ]      ]    ]
           }

ir_vars = {dlab_variable(compare, 1-1, [<, >, =,  =<, >=]}
```

Figure 3. A $\mathcal{D}$LAB grammar for the invariant relations application

which is equivalent to $(z + u * v = x * y) \wedge (u \geq 0)$. Notice that if this relation is indeed invariant at position †, then whenever the loop terminates on $u = 0$, the intended final relation $z = x * y$ holds.

This application demonstrates that $\mathcal{C}$LAUDIEN is able to handle structured terms (e.g. $Z + U * V$). Though, in this experiment built-in predicates were employed, similar results would have been obtained using the pure PROLOG notation for natural numbers, i.e. using 0 and the successor functor.

5.3. Finite element mesh-design

One standard benchmark for inductive logic programming systems operating under the discriminant setting, is that of learning finite element mesh-design (see e.g. (Dolšak & Muggleton, 1992, Lavrač & Džeroski, 1994)). Here we will address the same learning task. However, whereas the other approaches require positive as well as negative examples, $\mathcal{C}$LAUDIEN needs only the positive. Secondly, the other approaches employ Michalski's covering algorithm, where the aim is to find hypotheses that cover each positive example once. $\mathcal{C}$LAUDIEN follows an alternative approach, as it merely looks for valid rules. There is therefore no guarantee that hypotheses found by $\mathcal{C}$LAUDIEN will cover all positives and also a hypothesis may cover a positive example several times. We believe – and our experiments in mesh-design show – that when the data are sparse, the $\mathcal{C}$LAUDIEN approach may be preferrable.

The original mesh-application contains data about 5 different structures (a-e), with the number of edges per structure varying between 28 and 96. There are 278 positive examples (and 2840 negative ones) and the original background theory contains 1872 facts. The original background theory was made determinate (because the GOLEM system of (Muggleton & Feng, 1990) cannot work with indeterminate clauses). As $\mathcal{C}$LAUDIEN does not suffer from this restriction, we could compact the database to 639 (equivalent) facts. An example of a positive example is $mesh(b11, 6)$ meaning that edge 11 of structure b should be divided in 6 subedges. Background knowledge contains information about edge types, boundary conditions, loading, and the geometry of the structure. Some of the facts are shown below:

Edge types: $long(b19)$, $short(b10)$, $notimportant(b2)$, $shortforhole(b28)$, $halfcircuit(b3)$, $halfcircuithole(b1)$
Boundary conditions: $fixed(b1)$, $twosidefixed(b6)$
Loading: $notloaded(b1)$, $contloaded(b22)$
Geometry: $neighbour(b1, b2)$, $opposite(b1, b3)$, $same(b1, b3)$

We ran $\mathcal{C}$LAUDIEN on this data-set using a slightly different but equivalent representation for examples, using the leave-one-out strategy. All data were put into one observation. Counts of local accuracy $LA(c)$ and local coverage $LC(c)$ were done w.r.t. to the literal $mesh(E, R)$. Further settings include:

search strategy: best first
heuristic: $p/(l + n)$
$LA(c)$ threshold: 0.9
$LC(c)$ threshold: 2
$\mathcal{D}$LAB grammar: see Figure 4

The $\mathcal{D}$LAB grammar in Figure 4 defines a language of about $4.9 * 10^7$ rules. The antecedents of these rules specify at least the type, boundary conditions, loading or resolution of the edges that occur in the rule. Moreover, if two edges occur, the antecedent specifies their topology. The power of the $\mathcal{D}$LAB formalism is thus used to prevent the generation of a large class of uninteresting rules.

On average $\mathcal{C}$LAUDIEN halted after 7972 cpu seconds, visited 48534 nodes, which corresponds to about 0.01%, of the total search space, and discovered 495 valid rules. The high number of solutions can be explained by the low $LC(c)$ threshold.

In accordance to the any time character of $\mathcal{C}$LAUDIEN, the discovered rules were tested against the structure left out at regular cpu time intervals. In cases where more than one rule applied, the earliest found rule with the highest heuristic value was preferred. In Figure 5 the percentage of correct predictions is plotted against cpu time elapsed. Notice the quality of the theory improves more or less logarithmically. Figure 5 also shows results for GOLEM and FOIL as they are reported in (Lavrač & Džeroski, 1994).

We believe the results of these tests are very encouraging because the rules learned by $\mathcal{C}$LAUDIEN have by far the best classification accuracy and also because the cpu-requirements of $\mathcal{C}$LAUDIEN are of the same order as those by the other systems. The high classification accuracy can be explained by the sparseness of the data and the non-covering approach. FOIL and GOLEM are implemented in C, and $\mathcal{C}$LAUDIEN in PROLOG.

The experiment clearly shows that an any time algorithm (implemented in PROLOG) is not necessarily slower than a covering approach. (Part of) a possible explanation for this may be that CLAUDIEN is the only system that does not need to employ the (large number) of negative examples.

5.4. Mutagenesis

To illustrate the scientific discovery potential of CLAUDIEN we selected a problem from the field of organic chemistry which was recently brought to the attention of the inductive logic programming community by the Oxford University Computing Laboratory, in collaboration with the London Biomolecular Modelling Laboratory (Srinivasan *et al.*, 1995b). An observation here corresponds to a nitroaromatic compound with an associated mutagenicity value. There are 188 observations, 125 of which are labelled "active", meaning they have high mutagenicity. The observations further list information on atom and bond structures, a measure of hydrophobicity ($logp$), the energy of the compound's lowest unoccupied molecular orbital ($lumo$), and generic structural characteristics. For more details we refer to (Srinivasan *et al.*, 1995b).

So far experiments have focused on finding theories that discriminate between active and inactive compounds. For instance, with PROGOL (Muggleton, 1995) a predictive

```
mesh_temps =
   {R = resolution
    <--
    len-len:[ mesh(E,R),
              1-len: [type(E),boundary(E),loading(E)],
              0-len: [len-len: [geometry(E,E2),
                                1-len: [mesh(E2,resolution),
                                        type(E2),boundary(E2),loading(E2)
            ]        ]         ]       ]
   }
mesh_vars =
   {dlab_variable(resolution,1-1,[1,2,3,4,5,6,7,8,9,10,11,12,17]),
    dlab_variable(type,1-1,[long,usual,short,circuit,half_circuit,
                            quarter_circuit,short_for_hole,long_for_hole,
                            circuit_hole,half_circuit_hole,notimportant]),
    dlab_variable(boundary,1-1,[free,one_side_fixed,two_side_fixed,
                                fixed]),
    dlab_variable(loading,1-1,[noload,one_side_loaded,two_side_loaded,
                               cont_loaded]),
    dlab_variable(geometry,1-1,[neighbour,opp,eq])}
```

Figure 4. A DLAB grammar for the mesh application

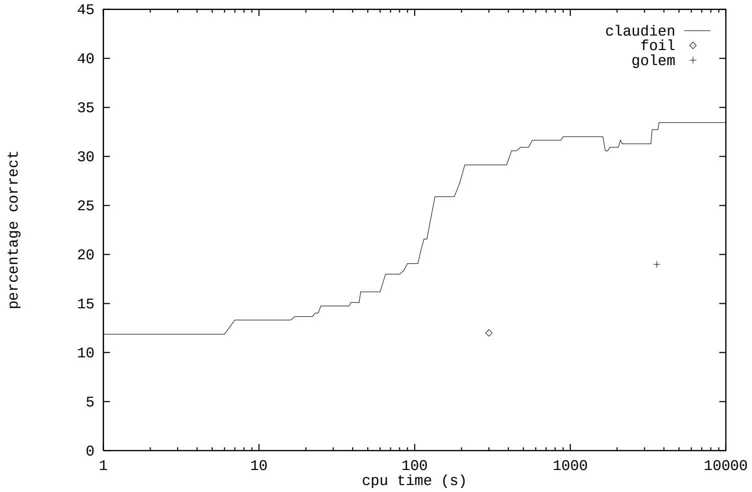

Figure 5. Comparing $\mathcal{C}$LAUDIEN to FOIL and GOLEM.

accuracy of 0.88 was obtained from a 10-fold cross-validation (Srinivasan *et al.*, 1995a). Despite the classification oriented approach of PROGOL, the most interesting outcome of the experiments of the Oxford - London team is *not* a classification criterion, but rather a new structural alert for mutagenic compounds. The new structural alert encodes one of the rules found by PROGOL. However, as PROGOL aims at classification, it is interested in as short a hypothesis as possible, implying that it aims at a minimal number of rules. Indeed, according to Michalski's covering approach, if a positive example is covered once by a rule in the hypothesis, it is no longer considered. Because of this, greedy classification algorithms may miss alternative explanations of the same data. $\mathcal{C}$LAUDIEN performing essentially an informed exhaustive search, will not miss such alternative explanations.

To test this hypothesis, we ran $\mathcal{C}$LAUDIEN on the mutagenisis problem with the aim of finding as much regularities of high accuracy and coverage as possible. The full $\mathcal{D}$LAB grammar for this task can be found in Appendix C. We here mention only a special feature $\#$ borrowed from PROGOL to generate thresholds for the values $logp$, $lumo$, and atomic charge. Clauses output by $\mathcal{D}$LAB contain bodyliterals such as $geteq(logp, LP, \#(T))$, where, before validity of the clause is calculated, $\#(T)$ is replaced by a constant such that the clause is non-trivially valid in at least one observation.

A sample of the results is shown below and was obtained in several runs of $\mathcal{C}$LAUDIEN, with a best-first search, with heuristic $p/(l+n)$, sometimes with slight variants of the $\mathcal{D}$LAB grammar, sometimes with alternative thresholds for $GA(c)$ and $GC(c)$. We ran first ran $\mathcal{C}$LAUDIEN with settings $GA(c) > 0.9$ and $GC(c) > 80$. In 90 cpu seconds, 35 rules were discovered, all variants of the following two:

```
active <--  lumo(Lumo) , lteq(lumo,Lumo,-1.62)
```

```
(accuracy: 0.9, coverage: 90)

active <--  not methyl(SP) , logp(LP) , gteq(logp,LP,3)
(accuracy: 0.9, coverage: 103)
```

We then lowered the $GC(c)$ threshold to 70. In two short subsequent runs, first with tests on thresholds for $logp$, $lumo$, and atomic charge disallowed, then with the structural characteristic $methyl$ removed from the language, two alternative explanations were discovered:

```
active <--  not methyl(SP) , atom(A1,Elem1,Type1,Charge1) , Type1 = 27,
            atom(A2,Elem2,Type2,Charge2), bond(A1,A2,7)
(accuracy: 0.91, coverage: 76)

active <-- benzene(SP),atom(A1,Elem1,Type1,Charge1),Type1 = 27,
           lteq(charge,Charge1,0.006)
(accuracy: 0.93, coverage: 70)
```

The underlying idea here is that the insights of one run, can be used in the next run. E.g. if the *not methyl* condition was allowed, nearly all rules discovered contained that condition. By excluding this condition, alternative explanations were found. Thus, the expert can and should guide the discovery process.

5.5. River water quality

The next application is taken from the domain of environmental monitoring (Džeroski *et al.*, 1994) (see also (Džeroski, 1995)). The goal here is to capture the expertise of an expert river ecologist who classified 292 field samples of benthic communities from British Midland Rivers. Each sample is described by means of the abundances (recorded on a scale of 0 to 6) of eighty different microinvertebrate families. The expert classified the samples into five classes.

In a first experiment we limited ourselves to discovering characteristics of poorest quality water. A simplified version of the $\mathcal{D}$LAB grammar used is shown in Figure 6.

The size of the actual language used was of order 10^{96}. The accuracy threshold for $GA(c)$ was set to 1, but we used an extra feature of $\mathcal{C}$LAUDIEN to list (but not prune) all rules with accuracy above a lower accuracy level set to 0.3. With 20% of the samples belonging to water quality class 0, the idea here was to delineate subgroups of water samples with a percentage of class 0 above average. Other relevant settings were:

search strategy: best first
heuristic: $p/(l + n)$
$GC(c)$ threshold: 10

We ran $\mathcal{C}$LAUDIEN for about 1500 cpu seconds. In this period 2752 rules were discovered. After post-processing, we derived chains of the following type, where the addition of extra conditions on each new line leads to an increase of $GA(c)$ and a decrease of $GC(c)$.

```
eco_temps = {class(0)
             <--
             0-len:[len-len:[ancylidae(A1),
                    0-1:[compare(abundance,A1)]],
                    len-len:[asellidae(A2),
                    0-1:[compare(abundance,A2)]],
                    ...
                    len-len:[veliidae(A80),
                    0-1:[compare(abundance,A80)]]
                   ]
            }

eco_vars = {dlab_variable(compare, 1-1, [=,<,>],
            dlab_variable(abundance, 1-1, [0,1,2,3,4,5,6]}
```

Figure 6. A $\mathcal{D}$LAB grammar for the river water quality application

		$GA(c)$	$GC(c)$
class(0) if	true,	0.20	292
	heptageniidae(D32),	0.69	75
	hydropsychidae(D37),	0.73	49
	oligochaeta(D54),	0.74	46
	perlodidae(D57),	0.89	35
	rhyacophilidae(D69),	0.93	29
	tipulidae(D76),	0.96	26
	D76 = 2	1	17

This setting where low accuracy rules are shown but not pruned, seems particularly interesting in cases where no rules with both high accuracy and high coverage are to be expected, for instance when sufficient conditions have to be discovered for the occurrence of rare "faults" in processes, machines, or human beings.

For a second experiment with the river quality data, we turned the lower accuracy facility off, set $GA(c)$ to 0.95, and modified the language such that rules could cover more than one class:

```
eco_temps = {class(1-2:[0,1,2,3,4])
            <--
            ....}
```

In a search space, now of order 10^{97}, $\mathcal{C}$LAUDIEN discovered 49 rules in 24 hours of cpu time. For instance,

```
class(2) <-- asellidae(A2), chironomidae(A11), gammaridae(A26),
            A26 = 2, lymnaeidae(A46)
(accuracy: 0.96, coverage: 28)
```

```
class(2), class(3) <-- asellidae(A2), glossiphoniidae(A28), physidae(A59)
(accuracy: 0.95, coverage: 22)
```

Ten of these rules have the disjunction $class(2), class(3)$ in the head, the others only $class(2)$. After we eliminated the abundance level tests, and lowered the $GA(c)$ threshold to 0.9, $\mathcal{C}$LAUDIEN discovered the following two rules with class disjunction within 20 cpu seconds:

```
class(2), class(3) <-- physidae(A59), tubificidae(A77)
(accuracy: 0.9, coverage: 40)

class(2), class(3) <-- asellidae(D2), physidae(D59)
(accuracy: 0.92, coverage: 39)
```

Finally, we removed $class(2)$ from the language, and raised the $GC(c)$ threshold to 30. In this modified setting, $\mathcal{C}$LAUDIEN discovered 65 rules within 14 hours of cpu time, three of which are shown below:

```
class(0), class(1) <-- perlodidae(D57)
(accuracy: 1, coverage: 57)

class(0), class(1) <-- elminthidae(D21) , tubificidae(D77)
(accuracy(0.9), coverage: 80)

class(0), class(1) <-- heptageniidae(D32)
(accuracy: 1, coverage: 75)
```

In a similar experiment reported in (Džeroski *et al.*, 1994) class disjunction turned out to be the main reason why domain experts judged $\mathcal{C}$LAUDIEN rules to be the most intuitive and promising, as compared to rules discovered by an extended version of the propositional learner CN2 (Clark & Niblett, 1989, Džeroski *et al.*, 1993) and GOLEM. This experiment illustrates $\mathcal{C}$LAUDIEN can also be applied when class boundaries are vague or based on a discretisation of a continuous space. If permitted by the $\mathcal{D}$LAB bias, $\mathcal{C}$LAUDIEN will attempt to disjunctively combine classes to construct valid rules. An analysis of the discovered hypothesis might then inspire the expert to introduce new (super)classes for frequent class combinations.

5.6. ***Parallel*** $\mathcal{C}$LAUDIEN

In the final experiment, our aim was to measure and compare the speed at which sequential and parallel $\mathcal{C}$LAUDIEN traverse the same hypothesis space. We tuned the mesh and ecology experiments such that in an exhaustive run $\mathcal{C}$LAUDIEN visited about 120000 nodes. We then ran $\mathcal{C}$LAUDIEN using a depth-first search strategy with 1, 2, 4, 8, and 16 processes. With each tested clause, and again with each solution found, we recorded the consumed cpu time in seconds[9].

The results of running $\mathcal{C}$LAUDIEN with 1, 2, 4, 8, and 16 processes are reported in Figure 7. In the charts on top, the values on the y-axis are the number of explored nodes. If n is the degree of concurrency, and $explored(p, t)$ the number of nodes explored by process p after p has consumed t cpu seconds, then $y = f(t) = \sum_{p=1}^{n} explored(p, t)$. The clauses that were found to be valid are marked with a diamond. A separate chart with the number of solutions is presented in the lower half of Figure 7.

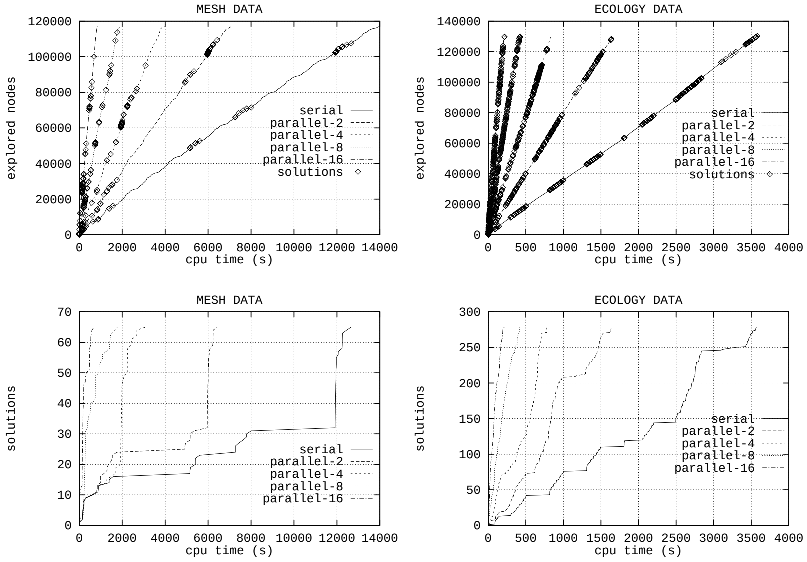

Figure 7. Results of the experiment with parallel $\mathcal{C}$LAUDIEN

The results shown in Figure 7 indicate that for up to 16 processes, the speedup is approximately proportional to the number of processes executing the task: the consumed cpu time is roughly halved each time the number of processes is doubled.

An important question related to the results of our experiments with parallel $\mathcal{C}$LAUDIEN is how long we can go on adding new processes to reduce the consumed cpu time. Apart from obvious hardware restrictions[10], there are mainly two software related limitations we should take into account when trying to solve this question.

The first, application-dependent, upper boundary on the degree of concurrency stems from the fact that a (near) linear speedup can only be obtained if all processes are more or less constantly working on a subtask, i.e. if most of the time there are enough sublanguages L_i available. The maximal number of candidate sublanguages available at a given time equals the total size of all local queues QC (see Figure A.1) and is related to the application-specific average branching factor. It is for instance easy to see that in the extreme case where the branching factor equals 1, concurrency will produce no speedup at all.

Secondly, interprocess communication requires a certain amount of computational overhead. If this overhead increases with the degree of concurrency, as it does with our *naive* implementation of parallel $\mathcal{C}$LAUDIEN, there will be a point where adding more processes is useless, or even counter-productive in terms of consumed cpu time.

6. Related Work

The clausal discovery engine presented here is related to data mining research, semantics for induction and inductive logic programming.

First, the techniques presented fit in an attempt to upgrade the data mining paradigm to considering multiple relations (cf. (Džeroski, 1995)). Evidence for this claim was provided by showing how the semantics for characterizing induction from interpretations fits in Mannila's general framework for data mining as well as by showing that $\mathcal{C}$LAUDIEN can emulate many of the existing data mining systems. The emulations also demonstrate the generality of a first order clausal discovery engine as compared to propositional ones. As we discussed, the price to pay for generality and for expressive power, is a potential loss in efficiency on specific tasks. However, $\mathcal{C}$LAUDIEN was shown not only to be able to search complex and vast hypotheses spaces, but also to handle reasonably large data sets. Furthermore, the task addressed by $\mathcal{C}$LAUDIEN is PAC-learnable (cf. (De Raedt & Džeroski, 1994)), and the implemented engine is much more efficient than the naive algorithm used to prove the PAC-learning results. Thus $\mathcal{C}$LAUDIEN should not be considered inefficient.

Secondly, the presented work also contributes to the semantics for induction. More specifically, it adopts the frameworks by (De Raedt & Džeroski, 1994) and (Helft, 1989). It generalizes the work of Helft by the use of multiple observations (and models) as well as the use of Herbrand interpretations. Furthermore, it discusses many variants, options and extensions of the pure logical view of Helft and De Raedt and Džeroski.

Thirdly, clausal discovery is also a contribution to the field of inductive logic programming, in that it shows how a slightly different formalisation of induction within logic programming results in new possibilities and challenges for inductive logic programming. One important contribution in this respect is the extension from definite clause logic to full clausal logic made possible by the novel semantics.

7. Conclusions

We have presented a clausal discovery engine based on a novel semantics for induction for use in a data mining setting. Theoretical properties of the engine as well as experiments with the engine were presented. A key ingredient of the engine was a declarative language bias formalism, with a corresponding refinement operator.

The clausal discovery engine and theory can be extended in various directions. First, it would be interesting to see how it can handle incompletely specified observations (using partial models). Secondly, how it can perform discriminating induction. A step in this direction was already taken by (De Raedt & Van Laer, 1995). Thirdly, it would be interesting to see how the engine can be coupled to a relational database system and evaluate its performance on huge data bases. Finally, we wonder whether the clausal logic representation can be extended towards full first order logic.

We hope that the presented framework will provide a sound basis for combining data mining principles with inductive logic programming.

Acknowledgments

We would like to thank Sašo Džeroski and Maurice Bruynooghe for their involvement in the research that finally lead to this paper. Further discussions with Nada Lavrač, Stephen Muggleton and Peter Flach proved to be very fruitful. Bojan Dolsak, Sašo Džeroski and Ashwin Srinivasan generously provided the mesh, ecology and mutagenisis data used in the experiments. Patrick Weemeeuw and Bart Demoen provided advice on the parallel implementation of $\mathcal{C}$LAUDIEN. We also thank Wim Van Laer for his significant contribution to the implementation of $\mathcal{C}$LAUDIEN, and for his comments on this paper. Finally, Hendrik Blockeel as well as a (large) number of master's students experimented with earlier versions of the $\mathcal{C}$LAUDIEN implementation.

Luc De Raedt is supported by the Belgian National Fund for Scientific Research and by the ESPRIT projects no. 6020 and 20237 on Inductive Logic Programming and Inductive Logic Programming II.

Appendix A

A parallel implementation

ParallelClausalDiscovery (see Algorithm A.1) is the main function of the parallel version of the algorithm. The input parameter n determines the degree of parallellism, i.e. the

```
function ParaClausalDiscovery
    inputs : $O$: set of Closed observations, $B$: background theory,
        $\rho$: refinement operator, $n$ : number of processors
    outputs : Characterizing Hypothesis

$Q(1)$ := $\{\square\}$
for all $i \in 2 \ldots n$ do $Q(i) := \emptyset$
$H_2$ := fork(ParaCD(2))
. . .
$H_n$ := fork(ParaCD(n))
$H_1$ := ParaCD(1)
$H := \cup H_i$
reduce(H)
return $H$
endfunction

function ParaCD
    inputs : $p$: name of processor,
    outputs : Partial Confirmatory Hypothesis

$H_p$ := $\emptyset$
while not ($\forall i \in 1 \ldots n : Q(i) = \emptyset$) do
    while not ($\forall i \in 1 \ldots n : Q(i) = \emptyset$) and ($Q(p) = \emptyset$) do skip
    $\boxed{Queue \text{ := Q(p)}}$
    while $Queue \neq \emptyset$ do
        for all $i \in 1 \ldots n$ do $\boxed{\textbf{if } \text{Q(i) = } \emptyset \textbf{ then } move \text{ part of Queue to Q(i)}}$
        $delete$ $c$ from $Queue$
        if $c$ is $valid$ on $O$ and not prune1(c)
        then add $c$ to $H_p$
        else for all $c' \in \rho(c)$ for which not prune2(c') do  add $c'$ to $Queue$
        endif
    endwhile
    Q(p) := $\emptyset$
endwhile
return $H_p$
endfunction
```

Figure A.1. A parallel clausal discovery algorithm

maximal number of processes that will be executing concurrently. Processes exchange information through the use of the shared variable Q[11]. For each of the n processes, this variable contains a queue equivalent to queue Q in ClausalDiscovery. Initially, all queues in Q except the one of the first process are set to empty. The queue of the first process is initialized to the top node of the hypothesis space, i.e. $\Box$. The UNIX[12] inspired $fork$ instruction creates a new (*child*) process that will execute the call given as the single argument of $fork$ concurrently with the calling (*parent*) process. ParallelClausalDiscovery calls ParaCD, n times. The $fork$ instruction causes $n - 1$ of these calls to be executed concurrently with the parent process in $n-1$ newly created processes. All results are stored in $H_1 \ldots H_n$ and combined to H, which is ultimately returned as the solution.

The single input parameter p of ParaCD ranges between 1 and n, and identifies the present process. Global variable $Q(p)$ contains a queue of clauses that represents the root of the subtree to be explored by p. The outmost loop terminates the moment this queue is empty for all processes. At that moment the local solution H_p is returned and ParaCD stops. There are two more nested loops. The first one terminates either if the same condition of the outer loop is fulfilled or if the current process has received a new subtree. The body of this loop is empty but for the do-nothing-instruction $skip$. After termination of this first inner loop, $Queue$ gets the value of $Q(p)$. The second inner loop is a near copy of ClausalDiscovery. The only difference is that at the beginning of each step Q is searched for empty queues. If such an empty queue is found on position i in Q, process p cedes part of its subtree to process i by moving part of $Queue$ to $Q(i)$. Which part of $Queue$ is moved will depend on the search strategy chosen by the user (cf. parameter $delete$ in Figure 1). An important general restriction is that the *move* instruction should not be allowed to empty $Queue$, as this might result in a loop where the same subtask is passed round forever. From the moment $Queue$ contains no further candidates for refinement, $Q(p)$ is set to empty in order to inform the other processes that process p is ready to receive a new subtask, i.e. a new subtree.

In case common variables such as Q are used for interprocess communication the synchronisation problem of mutual exclusion occurs. *Mutual exclusion* is concerned with ensuring that a sequence of statements, called a *critical section*, is treated as an indivisible operation that can not be executed by more than one process at the same time. In ParaCD the boxes mark two critical sections. They should prevent that two processes are simultaneously writing to $Q(i)$ or that the incomplete $Q(p)$ is copied to $Queue$ while it is being written by some other process.

It is easy to see that ParallelClausalDiscovery has the same behaviour as ClausalDiscovery.

Appendix B

A $\mathcal{D}$LAB$^{\ominus}$ refinement operator

A refinement operator ρ (cf. Definition 4) for $\mathcal{D}$LAB$^{\ominus}$ is based on the observation that clauses c in $dlab\_generate(DGRAM)$ are defined by a sequence of sublist selections from $\mathcal{D}$LAB$^{\ominus}$ atoms occurring in $DGRAM$. If we enlarge one of these sublists then the clause $c' \supseteq c$ defined by the new sequence is a specialisation of c under θ-subsumption. If we somehow enlarge one sublist in a minimal way, then c' will be a refinement, i.e. a maximally general specialisation of c[13]. To implement this idea we adapt the definite clause grammar $dlab\_dcg$ in Definition 9 in three steps.

First, in order to formalize the above notion of a sequence of sublist selections, we add to $dlab\_dcg$ an extra argument we will refer to as the $\mathcal{D}\mathrm{LAB}^{\ominus}$ path. The $\mathcal{D}\mathrm{LAB}^{\ominus}$ path is meant to keep track of applications of Rules (3) and (4) in $dlab\_dcg$. The application of these rules determines whether the first $\mathcal{D}\mathrm{LAB}^{\ominus}$ atom in list L of $Min \cdot\cdot Max : L$ is either skipped (Rule (3)) or included in the sublist (Rule (4)).

Definition 16 ($\mathcal{D}\mathrm{LAB}^{\ominus}$ path) *Let $DATOM$ be a $\mathcal{D}\mathrm{LAB}^{\ominus}$ atom, and C a list of literals generated by $dlab\_dcg(DATOM)$. $DPATH$ is a $\mathcal{D}\mathrm{LAB}^{\ominus}$ path of C with regard to $DATOM$ if and only if*

- *$DATOM \neq Min \cdot\cdot Max : L$ and $DPATH = DATOM$ or*
- *$DATOM = Min \cdot\cdot Max : [L_1, \ldots, L_n]$ and $DPATH = [P_1, \ldots, P_n]$, with, for each $P_i \in DPATH$,*
  - *$P_i = *$ and L_i is excluded during generation of C (application of Rule (3)/(B.3)), or*
  - *P_i is the $\mathcal{D}\mathrm{LAB}^{\ominus}$ path of C with regard to $\mathcal{D}\mathrm{LAB}^{\ominus}$ atom L_i and L_i is included during generation of C (application of Rule (4)/(B.4))*

For instance,

$DATOM = 0 \cdot\cdot 2 : [gorilla(X), 1 \cdot\cdot 1 : [female(X), male(X)]]$	
$C = dlab\_dcg(DATOM)$	$\mathcal{D}\mathrm{LAB}^{\ominus}$ path of C with regard to $DATOM$
$[]$	$[*, *]$
$[male(X)]$	$[*, [*, male(X)]]$
$[female(X)]$	$[*, [female(X), *]]$
$[gorilla(X)]$	$[gorilla(X), *]$
$[gorilla(X), male(X)]$	$[gorilla(X), [*, male(X)]]$
$[gorilla(X), female(X)]$	$[gorilla(X), [female(X), *]]$

The following is an adaptation of $dlab\_dcg$, with the $\mathcal{D}\mathrm{LAB}^{\ominus}$ path in the second argument position.

$$\begin{aligned}
dlab2(A, A) &\longrightarrow [A], \{A \neq Min \cdot\cdot Max : L\}. && \text{(B.1)}\\
dlab2(Min \cdot\cdot Max : [], []) &\longrightarrow \{Min \leq 0\}, []. && \text{(B.2)}\\
dlab2(Min \cdot\cdot Max : [\_|L], [*|Y]) &\longrightarrow dlab2(Min \cdot\cdot Max : L, Y). && \text{(B.3)}\\
dlab2(Min \cdot\cdot Max : [A|L], [X|Y]) &\longrightarrow \{Max > 0\}, dlab2(A, X), &&\\
&\qquad dlab2((Min - 1) \cdot\cdot (Max - 1) : L, Y). && \text{(B.4)}
\end{aligned}$$

In a second step, we can use the $\mathcal{D}\mathrm{LAB}^{\ominus}$ path DP of a list of literals C to generate superlists of C. Every $*$ in DP marks an occasion for extending C. In terms of Definition 16: we have to locate a $P_i = *$ in DP indicating the corresponding $\mathcal{D}\mathrm{LAB}^{\ominus}$ atom L_i is excluded during generation of C , and then include L_i during generation of superlists C' of C. Definite clause grammar $dlabs$ does that, and moreover returns the $\mathcal{D}\mathrm{LAB}^{\ominus}$ path DP' of C' in the third argument position.

$$\begin{aligned}
dlabs(\_\cdot\cdot Max : [], [], []) &\longrightarrow []. && \text{(B.5)}\\
dlabs(\_\cdot\cdot Max : [A|L], [*|Y], [X|Z]) &\longrightarrow \{Max > 0\}, dlab2(A, X),\\
& dlabs(\_\cdot\cdot(Max - 1) : L, Y, Z). && \text{(B.6)}\\
dlabs(\_\cdot\cdot Max : [\_|L], [*|Y], [*|Z]) &\longrightarrow dlabs(\_\cdot\cdot Max : L, Y, Z). && \text{(B.7)}\\
dlabs(\_\cdot\cdot Max : [A|L], [P|Y], [Q|Z]) &\longrightarrow \{P \neq *, Max > 0\}, dlabs(A, P, Q),\\
& dlabs(\_\cdot\cdot(Max - 1) : L, Y, Z). && \text{(B.8)}\\
dlabs(\_\cdot\cdot Max : [A|L], [X|Y], [X|Z]) &\longrightarrow \{X \neq *, Max > 0\}, dlab2(A, X),\\
& dlabs(\_\cdot\cdot(Max - 1) : L, Y, Z). && \text{(B.9)}
\end{aligned}$$

Notice how in Rule (B.6) of $dlabs$ the previously excluded A (cf. the $*$ in Arg2) is now included with the call of $dlab2(A, X)$. For instance,

<table>
<tr><td colspan="2">$DATOM = 0\cdot\cdot 3 : [gorilla(X), female(X), male(X)]$
$C = [female(X)]$
$DP = [*, female(X), *]$</td></tr>
<tr><td>$C' = dlabs(DATOM, DP, DP')$</td><td>$DP'$</td></tr>
<tr><td>$[gorilla(X), female(X), male(X)]$
$[gorilla(X), female(X)]$
$[female(X), male(X)]$
$[female(X)]$</td><td>$[gorilla(X), female(X), male(X)]$
$[gorilla(X), female(X), *]$
$[*, female(X), male(X)]$
$[*, female(X), *]$</td></tr>
</table>

The rules in $dlabs$ can be used to find all specialisations c' of c. As we want our refinement operator to generate only maximally general specialisations of c, a final adaptation of $dlabs$ is required such that it will generate only smallest superlists of C. Roughly stated, exactly one $*$ in the $\mathcal{D}\mathrm{LAB}^{\ominus}$ path DP of a list of literals C should be expanded, and then only in a minimal way. The first requirement, again in terms of Definition 16, says that we should locate exactly one $P_i = *$ in DP , and then include L_i during generation of superlists of C. The second requirement says that the inclusion of L_i should be minimal in the sense that the corresponding $\mathcal{D}\mathrm{LAB}^{\ominus}$ path P_i' should contain the maximally allowed number of $*$'s. For this we need a modified version of $dlab2$, that, given a $\mathcal{D}\mathrm{LAB}^{\ominus}$ atom $Min \cdot\cdot Max : L$, will only generate sublists of length Min. The first requirement is realized in $dlabr$ by eliminating some recursive calls, the second by initialisation of the newly included $\mathcal{D}\mathrm{LAB}^{\ominus}$ atom A with $dlabi$ instead of $dlab2$.

$$\begin{aligned}
dlabr(Min \cdot\cdot Max : [A|L], [*|Y], [X|Y]) &\longrightarrow \{not(dlab\_optimal, member(E, Y), E \neq *)\},\\
& \{Max > 0\}, dlabi(A, X),\\
& dlab2((Min - 1)\cdot\cdot(Max - 1) : L, Y). && \text{(B.10)}\\
dlabr(Min \cdot\cdot Max : [\_|L], [*|Y], [*|Z]) &\longrightarrow dlabr(Min \cdot\cdot Max : L, Y, Z). && \text{(B.11)}\\
dlabr(Min \cdot\cdot Max : [A|L], [X|Z], [Y|Z]) &\longrightarrow \{X \neq *, Max > 0\}, dlabr(A, X, Y),\\
& dlab2((Min - 1)\cdot\cdot(Max - 1) : L, Z). && \text{(B.12)}\\
dlabr(Min \cdot\cdot Max : [A|L], [X|Y], [X|Z]) &\longrightarrow \{X \neq *, Max > 0\}, dlab2(A, X),\\
& dlabr((Min - 1)\cdot\cdot(Max - 1) : L, Y, Z). && \text{(B.13)}
\end{aligned}$$

$$\begin{aligned} dlabi(A, A) &\longrightarrow [A], \{not(A = Min \cdot\cdot Max : L)\}. && \text{(B.14)}\\ dlabi(0 \cdot\cdot \_ : [], []) &\longrightarrow []. && \text{(B.15)}\\ dlabi(Min \cdot\cdot \_ : [A|L], [X|Y]) &\longrightarrow dlabi(A, X), &&\\ & \ dlabi((Min - 1) \cdot\cdot \_ : L, Y). && \text{(B.16)}\\ dlabi(Min \cdot\cdot \_ : [\_|L], [*|Y]) &\longrightarrow dlabi(Min \cdot\cdot \_ : L, Y). && \text{(B.17)} \end{aligned}$$

Notice that Rule B.10 of $dlabr$ contains an extra initial condition:

$$not(dlab\_optimal, member(E, Y), E \neq *)$$

A call to $dlab\_optimal$ should succeed, if we want the refinement operator to be optimal (cf. Definition 5), and fail otherwise.

The extra condition ensures that when working in optimal mode, the refinement operator will never expand $*$'s to the left of already expanded $*$'s. For instance,

$DATOM = 0 \cdot\cdot 3 : [gorilla(X), female(X), male(X)]$ $C = [female(X)]$ $DP = [*, female(X), *]$		
$dlab\_optimal$	$C' = dlabr(DATOM, DP, DP')$	DP'
false	$[gorilla(X), female(X)]$ $[female(X), male(X)]$	$[gorilla(X), female(X), *]$ $[*, female(X), male(X)]$
true	$[female(X), male(X)]$	$[*, female(X), male(X)]$

To further enforce optimality we have to make sure refinement of the head of a clause blocks all future refinements of the body, or vice-versa[14].

We can now formulate the definition of a $\mathcal{D}$LAB$^{\ominus}$ refinement operator based on the twelve definite clause grammar rules of $dlabr$, $dlabi$, and $dlab2$.

Definition 17 (dlab_refine(DINFO,c)) *Given*

- *$\mathcal{D}$LAB$^{\ominus}$ template $HA \leftarrow BA$,*
- *clause $c = H \leftarrow B$, with $c \in dlab\_generate(\{HA \leftarrow BA\})$*
- *HP a $\mathcal{D}$LAB$^{\ominus}$ path of H with regard to HA,*
- *BP a $\mathcal{D}$LAB$^{\ominus}$ path of B with regard to BA,*
- *$DINFO = (HA, HP, BA, BP)$,*

If $dlab\_optimal = false$

$$dlab\_refine(DINFO, c) = dlab\_refh(DINFO, c) \cup dlab\_refb(DINFO, c)$$

If $dlab\_optimal = true$

$$dlab\_refine(DINFO, c) = dlab\_refh((HA, HP, [], []), c) \cup dlab\_refb(DINFO, c)$$

$$\begin{array}{l} dlab\_refh((HA, HP, BA, BP), H \leftarrow B) = \\ \qquad \{((HA, HP', BA, BP), H' \leftarrow B) | H' = dlabr(HA, HP, HP')\} \end{array}$$

$$\begin{array}{l} dlab\_refb((HA, HP, BA, BP), H \leftarrow B) = \\ \qquad \{((HA, HP, BA, BP'), H \leftarrow B') | B' = dlabr(BA, BP, BP')\} \end{array}$$

An initialisation function that returns the most general clauses in $\mathcal{L}$ completes the $\mathcal{D}$LAB$^{\ominus}$ refinement operator:

Definition 18 (dlab_initialize(DGRAM)) *Let DGRAM be a $\mathcal{D}$LAB$^{\ominus}$ grammar, then the following function returns the top nodes in the refinement lattice:*

$$\begin{array}{rl} dlab\_initialize(DGRAM) = & \{dlab\_refh(dlab\_refb(DINFO, \Box))| \\ & (HA \leftarrow BA) \in DGRAM, \\ & DINFO = (0 \cdot\cdot 1 : [HA], [*], 0 \cdot\cdot 1 : [BA], [*])\} \end{array}$$

We are now ready to instantiate the refinement operator in the ClausalDiscovery algorithm (see Figure 1) to $\mathcal{D}$LAB$^{\ominus}$, with $dlab\_optimal = true$. The basic idea is to store elements of type $(DINFO, c)$ in queue Q. As in practise queue Q often grows to a size above 10^5, the explicit storage of nodes $(DINFO, c)$ might quickly exhaust memory resources. The $\mathcal{D}$LAB$^{\ominus}$ formalism however allows for a straightforward optimisation, where only the $\mathcal{D}$LAB$^{\ominus}$ paths are stored in Q together with a pointer to the $\mathcal{D}$LAB$^{\ominus}$ template. Corresponding clauses can then be recovered using $dlab2$[15]. We then use $dlab\_initialize(DGRAM)$ to initialize Q to the most general element(s) in $\mathcal{L}$, and $dlab\_refine(DINFO, c)$ to calculate refinements of the elements we retrieve from Q.

Appendix C

A $\mathcal{D}$LAB grammar for the mutagenesis application

```
muta_temps =
 {active
  <--
  0-len:
   [toggle(structural_property(SP)),
    len-len:
     [atom(A1, Elem1, Type1, Charge1),
      0-len:[toggle(Elem1=element),
             toggle(Type1=atomtype),
             occurs_in(A1, SP)
            ],
      0-len:[len-len:[atom(A2, Elem2, Type2, Charge2),
                      0-len:[toggle(Elem2=element),
                             toggle(Type2=atomtype),
                             occurs_in(A2, SP),
                             bond(A1, A2, 1-1:[_,1,2,3,4,5,7]),
                             len-len:[atom(A3, Elem3, Type3, Charge3),
                                      0-len:[toggle(Elem3=element),
                                             toggle(Type3=atomtype),
                                             occurs_in(A3, SP),
                                             bond(A1, A3, 1-1:[_,1,2,3,4,5,7]),
                                             bond(A2, A3, 1-1:[_,1,2,3,4,5,7])
     ]      ]        ]      ]        ]      ],

    1-1:[eqtest(charge,1-1:[Charge1, Charge2, Charge3], #(T)),
         len-len:[lumo(Lumo),eqtest(lumo,Lumo, #(T))],
         len-len:[logp(LP),eqtest(logp,LP,#(T))]
  ]     ]

 }

muta_vars =
   {dlab_variable(eqtest,1 - 1,[lteq,gteq]),
    dlab_variable(element,1 - 1,[h,c,n,o,br,cl,f,i,s]),
    dlab_variable(atomtype,1 - 1,[1,3,8,10,14,16,19,21,22,25,26,27,28,29,31,32,34,
                                  35,36,38,40,41,42,45,49,50,51,52,72,92,93,94,
                                  95,194,195,230,232]),
    dlab_variable(structural_property,1 - 1,[nitro,carbon_6_ring,benzene,ring_size_6,
                                             ring_size_5,phenanthrene,anthracene,ball3,
                                             hetero_aromatic_5_ring,hetero_aromatic_6_ring,
                                             carbon_5_aromatic_ring,methyl]),
    dlab_variable(toggle,1 - 1,[call,not])
   }
```

Notes

1. Details on how to obtain CLAUDIEN can be found on the World-Wide-Web at URL:

   $$http : //www.cs.kuleuven.ac.be/\tilde{\ }ml/CWIS/claudien - E.shtml$$

   or by FTP access to:

   $$ftp : //ftp.cs.kuleuven.ac.be/pub/logic - prgm/ilp/claudien/claudien3.0/$$

2. There is some historical confusion in terminology here. Helft (Helft, 1989) introduced the term non-monotonic induction, Flach first distinguished weak induction from strong or normal induction (Flach, 1992), but now uses confirmatory and explanatory induction (Flach, 1994, Flach, 1995). Finally, though the setting by (De Raedt & Džeroski, 1994) is a generalisation of Helft's setting, they also used the term non-monotonic. The recent paper by (De Raedt, 1996) attempts to clarify this situation.
3. Notice that 'valid' does not mean 'tautology' here !
4. It is also possible to use non-definite clause theories. However, then the minimal Herbrand model of the theory may not be unique. Helft (Helft, 1989) shows how to deal with this situation.
5. One might as well use implication as a notion of generality, though this would be computationally harder.
6. DLAB is available as a PROLOG library at URL

   $$http : //www.cs.kuleuven.ac.be/\tilde{\ }ml/CWIS/dlab - E.shtml$$

   or by FTP access to:

   $$ftp : //ftp.cs.kuleuven.ac.be/pub/logic - prgm/ilp/dlab$$

7. To simplify our definition of a generation function we here introduce (and will continue to use) a special list notation in which the head and the body of clauses are written as lists: $[A_1, \ldots, A_m] \leftarrow [B_1, \ldots, B_n]$.
8. As a minor extension we will also allow $\mathcal{D}$LAB$^{\ominus}$ atoms of the type $Min \cdot\cdot len : L$ or $len \cdot\cdot len : L$, where len is a constant symbol that abbreviates $length(L)$.
9. As cpu time was measured, we could test parallel $\mathcal{C}$LAUDIEN with degrees above 4 on a machine with only 4 processors. It should be kept in mind however that the speedups here reported will only correspond to real time speedups if a separate processor is dedicated to all concurrent processes.
10. Remember that we assume every process can execute on a separate processor. If not enough processors are available, they have to be switched between processes. By ever increasing the number of processes scheduled for a single processor we will finally overload the operating system.
11. More sophisticated systems for interprocess communication exist, but for reasons of simplicity we will continue to use the most general and basic constructs throughout.
12. UNIX^{TM} Trademark of Bell Laboratories
13. Depending on the $\mathcal{D}$LAB$^{\ominus}$ grammar, this refinement (under θ-subsumption) can be proper or not.
14. In fact, both measures merely prevent the same couple of $\mathcal{D}$LAB$^{\ominus}$ paths (one for the head, one for the body) from being generated more that once. In case the list of body- or headliterals of a single clause corresponds to $n > 1$ $\mathcal{D}$LAB$^{\ominus}$ paths, e.g. $[male(X)]$ given $\mathcal{D}$LAB$^{\ominus}$ atom $1 \cdot\cdot 1 : [male(X), male(X), male(X)]$ $(n = 3)$, $\mathcal{D}$LAB$^{\ominus}$ is likely to generate this clause n times. Part of the responsibility for optimality is thus left to the $\mathcal{D}$LAB$^{\ominus}$ user.
15. In a more sophisticated version of $\mathcal{D}$LAB$^{\ominus}$ the $\mathcal{D}$LAB$^{\ominus}$ paths are flat lists of symbols $0, 1, *$, such that groups of 4 elements in the path can be further compressed to one 81-ary digit.

References

Adé, H., De Raedt, L. and Bruynooghe, M. 1995. Declarative Bias for Specific-to-General ILP Systems. *Machine Learning*, 20(1/2):119 – 154.

Agrawal, R., Imielinski, T. and Swami, A. 1993. Mining association rules between sets of items in large databases. In *Proceedings of the 1993 International Conference on Management of Data (SIGMOD 93)*, pages 207–216.

Bergadano, F. & Gunetti, D. 1993. An interactive system to learn functional logic programs. In *Proceedings of the 13th International Joint Conference on Artificial Intelligence*, pages 1044–1049. Morgan Kaufmann.

Bergadano, F. 1993. Towards an inductive logic programming language. Technical Report ESPRIT project no. 6020 ILP Deliverable TO1, Computer Science Department, University of Torino.

Bratko, I. & Grobelnik, M. 1993. Inductive learning applied to program construction and verification. In *Proceedings of the 3rd International Workshop on Inductive Logic Programming*, pages 279–292.

Bratko, I. 1986. *Prolog Programming for Artificial Intelligence*. Addison-Wesley.

Cameron-Jones, R.M. & Quinlan, J.R. 1993. Avoiding pitfalls when learning recursive theories. In *Proceedings of the 13th International Joint Conference on Artificial Intelligence*, pages 1050–1055. Morgan Kaufmann.

Clark, P. & Niblett, T. 1989. The CN2 algorithm. *Machine Learning*, 3(4):261–284.

Clocksin, W.F. & Mellish, C.S. 1981. *Programming in Prolog*. Springer-Verlag, Berlin.

Cohen, W.W. 1994. Grammatically biased learning: learning logic programs using an explicit antecedent description language. *Artificial Intelligence*, 68:303–366.

De Raedt, L. & Bruynooghe, M. 1993. A theory of clausal discovery. In *Proceedings of the 13th International Joint Conference on Artificial Intelligence*, pages 1058–1063. Morgan Kaufmann.

De Raedt, L. & Džeroski, S. 1994. First order jk-clausal theories are PAC-learnable. *Artificial Intelligence*, 70:375–392.

De Raedt, L. & Van Laer, W. 1995. Inductive constraint logic. In *Proceedings of the 5th Workshop on Algorithmic Learning Theory*, Volume 997 of Lecture Notes in Artificial Intelligence. Springer-Verlag.

De Raedt, L., Lavrač, N. & Džeroski, S. 1993. Multiple predicate learning. In *Proceedings of the 13th International Joint Conference on Artificial Intelligence*, pages 1037–1042. Morgan Kaufmann.

De Raedt, L. 1996. Induction in logic. In R.S. Michalski and Wnek J., editors, *Proceedings of the 3rd International Workshop on Multistrategy Learning*, pages 29–38.

Dehaspe, L. & De Raedt, L. 1996. DLAB: A declarative language bias formalism. In *Proceedings of the International Symposium on Methodologies for Intelligent Systems (ISMIS96)*, volume 1079 of *Lecture Notes in Artificial Intelligence*, pages 613–622. Springer-Verlag.

Dolšak, B. & Muggleton, S. 1992. The application of Inductive Logic Programming to finite element mesh design. In S. Muggleton, editor, *Inductive logic programming*, pages 453–472. Academic Press.

Džeroski, S., Cestnik, B. & Petrovski, I. 1993. Using the m-estimate in rule induction. *Journal of Computing and Information Technology*, 1(1):37 – 46.

Džeroski, S., Dehaspe, L., Ruck, B. & Walley, W. 1994. Classification of river water quality data using machine learning. In *Proceedings of the 5th International Conference on the Development and Application of Computer Techniques to Environmental Studies*.

Džeroski, S. 1995. Inductive logic programming and knowledge discovery in databases. In U. Fayyad, G. Piatetsky-Shapiro, P. Smyth, and R. Uthurusamy, editors, *Advances in Knowledge Discovery and Data Mining*, pages 118–152. The MIT Press.

Emde, W., Habel, C.U. & Rollinger, C.R. 1983. The discovery of the equator or concept driven learning. In *Proceedings of the 8th International Joint Conference on Artificial Intelligence*, pages 455–458. Morgan Kaufmann.

Fayyad, U., Piatetsky-Shapiro, G., Smyth, P. & Uthurusamy, R. editors. 1995. *Advances in Knowledge Discovery and Data Mining*. The MIT Press.

Fensel, D., Zickwolff, M. & Wiese, M. 1995. Are substitutions the better examples? In L. De Raedt, editor, *Proceedings of the 5th International Workshop on Inductive Logic Programming*.

Flach, P. 1992. A framework for inductive logic programming. In S. Muggleton, editor, *Inductive logic programming*. Academic Press.

Flach, P. 1993. Predicate invention in inductive data engineering. In P. Brazdil, editor, *Proceedings of the 6th European Conference on Machine Learning*, Volume 667 of Lecture Notes in Artificial Intelligence, pages 83–94. Springer-Verlag.

Flach, P.R. 1994. Inductive logic programming and philosophy of science. In S. Wrobel, editor, *Proceedings of the 4th International Workshop on Inductive Logic Programming*, volume 237 of *GMD-Studien*, Sankt Augustin, Germany. Gesellschaft für Mathematik und Datenverarbeitung MBH.

Flach, P. 1995. *An inquiry concerning the logic of induction*. PhD thesis, Tilburg University, Institute for Language Technology and Artificial Intelligence.

Genesereth, M. & Nilsson, N. 1987. *Logical foundations of artificial intelligence*. Morgan Kaufmann.

Helft, N. 1989. Induction as nonmonotonic inference. In *Proceedings of the 1st International Conference on Principles of Knowledge Representation and Reasoning*, pages 149–156. Morgan Kaufmann.

Kantola, M., Mannila, H., Raiha, K.J. & Siirtola, H. 1992. Discovering functional and inclusion dependencies in relational databases. *International Journal of Intelligent Systems*, 7(7):561–607.

Kietz, J-U. & Wrobel, S. 1992. Controlling the complexity of learning in logic through syntactic and task-oriented models. In S. Muggleton, editor, *Inductive logic programming*, pages 335–359. Academic Press.

Klösgen, W. 1996. Explora: A multipattern and multistrategy discovery assistant. In U. Fayyad, G. Piatetsky-Shapiro, P. Smyth, and R. Uthurusamy, editors, *Advances in Knowledge Discovery and Data Mining*. The MIT Press.

Lavrač, N. & Džeroski, S. 1994. *Inductive Logic Programming: Techniques and Applications*. Ellis Horwood.

Lloyd, J.W. 1987. *Foundations of logic programming*. Springer-Verlag, 2nd edition.

MacDonald, I.G. 1979. *Symmetric functions and Hall polynomials*. Clarendon Oxford.

Mannila, H. 1995. Aspects of data mining. In Y. Kodratoff, G. Nakhaeizadeh, and G. Taylor, editors, *Proceedings of the MLnet Familiarization Workshop on Statistics, Machine Learning and Knowledge Discovery in Databases*, pages 1–6, Heraklion, Crete, Greece.

Manthey, R. & Bry, F. 1988. SATCHMO: a theorem prover implemented in Prolog. In *Proceedings of the 9th International Conference on Automated Deduction (CADE88)*, pages 415–434. Springer-Verlag.

Michalski, R.S. 1983. A theory and methodology of inductive learning. In R.S Michalski, J.G. Carbonell, and T.M. Mitchell, editors, *Machine Learning: an artificial intelligence approach*, volume 1. Morgan Kaufmann.

Mitchell, T.M. 1982. Generalization as search. *Artificial Intelligence*, 18:203–226.

Morik, K. & Brockhausen, P. 1996. A multistrategy approach to relational discovery in databases. In R.S. Michalski and Wnek J., editors, *Proceedings of the 3rd International Workshop on Multistrategy Learning*, pages 17–28.

Muggleton, S. & De Raedt, L. 1994. Inductive logic programming : Theory and methods. *Journal of Logic Programming*, 19,20:629–679.

Muggleton, S. & Feng, C. 1990. Efficient induction of logic programs. In *Proceedings of the 1st conference on algorithmic learning theory*, pages 368–381. Ohmsma, Tokyo, Japan.

Muggleton, S. 1995. Inverse entailment and Progol. *New Generation Computing*, 13.

Piatetsky-Shapiro, G. 1991. Discovery, analysis, and presentation of strong rules. In G. Piatetsky-Shapiro and W. Frawley, editors, *Knowledge Discovery in Databases*, pages 229–248. The MIT Press.

Plotkin, G. 1970. A note on inductive generalization. In *Machine Intelligence*, volume 5, pages 153–163. Edinburgh University Press.

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

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

Savnik, I. & Flach, P.A. 1993. Bottom-up induction of functional dependencies from relations. In *Proceedings of the AAAI'93 Workshop on Knowledge Discovery in Databases*, pages 174–185. AAAI Press. Washington DC.

Schlimmer, J. 1991. Learning determinations and checking databases. In *Proceedings of the AAAI'91 Workshop on Knowledge Discovery in Databases*, pages 64–761. Washington DC.

Shen, W.M. 1992. Discovering regularities from knowledge bases. *International Journal of Intelligent Systems*, 7(7).

Srinivasan, A., Muggleton, S. & Bain, M. 1992. Distinguishing exceptions from noise in non-monotonic learning. In *Proceedings of the 2nd International Workshop on Inductive Logic Programming*, 1992.

Srinivasan, A., Muggleton, S.H. & King, R.D. 1995. Comparing the use of background knowledge by inductiv e logic programming systems. In L. De Raedt, editor, *Proceedings of the 5th International Workshop on Inductive Logic Programming*. IOS Press.

Srinivasan, A., Muggleton, S.H., Sternberg, M.J.E. & King, R.D. 1995. Theories for mutagenicity: a study in first-order and feature-based induction. *Artificial Intelligence*. To appear.

Sterling, L. & Shapiro, E. 1986. *The art of Prolog*. The MIT Press.

van der Laag, P.R.J. & Nienhuys-Cheng, S.-H. 1994. Existence and nonexistence of complete refinement operators. In F. Bergadano and L. De Raedt, editors, *Proceedings of the 7th European Conference on Machine Learning*, volume 784 of *Lecture Notes in Artificial Intelligence*, pages 307–322. Springer-Verlag.

Wrobel, S. & Džeroski, S. 1995. The ILP description learning problem: Towards a general model-level definition of data mining in ILP. Technical report, Presented at the 1995 Workshop of the GI Special Interest Group Machine Learning (FGML-95).

Received December 8, 1995
Accepted June 5, 1996
Final Manuscript January 7, 1997