INDUCTIVE LOGIC PROGRAMMING: THEORY AND METHODS

STEPHEN MUGGLETON AND LUC DE RAEDT

▷ Inductive Logic Programming (ILP) is a new discipline which investigates the inductive construction of first-order clausal theories from examples and background knowledge. We survey the most important theories and methods of this new field. Firstly, various problem specifications of ILP are formalised in semantic settings for ILP, yielding a "model-theory" for ILP. Secondly, a generic ILP algorithm is presented. Thirdly, the inference rules and corresponding operators used in ILP are presented, resulting in a "proof-theory" for ILP. Fourthly, since inductive inference does not produce statements which are assured to follow from what is given, inductive inferences require an alternative form of justification. This can take the form of either probabilistic support or logical constraints on the hypothesis language. Information compression techniques used within ILP are presented within a unifying Bayesian approach to confirmation and corroboration of hypotheses. Also, different ways to constrain the hypothesis language, or specify the declarative bias are presented. Fifthly, some advanced topics in ILP are addressed. These include aspects of computational learning theory as applied to ILP, and the issue of predicate invention. Finally, we survey some applications and implementations of ILP. ILP applications fall under two different categories: firstly scientific discovery and knowledge acquisition, and secondly programming assistants. ◁

1. Introduction

Inductive Logic Programming (ILP) has been defined [81] as the intersection of inductive learning and logic programming. Thus *ILP* employs techniques from

*Address correspondence to*Oxford University Computing Laboratory, Wolfson Building, Parks Road, Oxford, OX1 3QD, UK.

*Address correspondence to*Dept. of Computing Science, Katholieke Universiteit Leuven, 200A Celestijnenlaan, B-3001, Heverlee, Belgium.

both machine learning and logic programming.

From inductive machine learning, ILP inherits its goal: to develop tools and techniques to induce hypotheses from observations (examples) and to synthesise new knowledge from experience. By using computational logic as the representational mechanism for hypotheses and observations, inductive logic programming can overcome the two main limitations of classical machine learning techniques, such as the Top-Down-Induction-of-Decision-Tree (TDIDT) family [101]):

1. the use of a limited knowledge representation formalism (essentially a propositional logic), and
2. difficulties in using substantial background knowledge in the learning process.

The first limitation is important because many domains of expertise can only be expressed in a first order logic, or a variant of first order logic, and not in a propositional one. One problem in which this is obvious is the domain of logic program synthesis from examples. Most logic programs cannot be defined using only propositional logic. The use of domain knowledge is also crucial because one of the well-established findings of artificial intelligence is that the use of domain knowledge is essential for achieving intelligent behaviour. Logic offers an elegant formalism to represent knowledge and hence incorporate it in the induction task.

From computational logic, inductive logic programming inherits its representational formalism, its semantical orientation and various well-established techniques. In contrast to most other approaches to inductive learning, inductive logic programming is interested in properties of inference rules, in convergence of algorithms and in the computational complexity of procedures. Many inductive logic programming systems benefit from using the results of computational logic. Additional benefit could potentially be derived from making use of work on termination, types and modes, knowledge-base updating, algorithmic debugging, abduction, constraint logic programming, program synthesis and program analysis.

Inductive logic programming extends the theory and practice of computational logic by investigating induction rather than deduction as the basic mode of inference. Whereas present computational logic theory describes deductive inference from logic formulae provided by the user, inductive logic programming theory describes the inductive inference of logic programs from instances and background knowledge. In this manner, ILP may contribute to the practice of logic programming, by providing tools that assist logic programmers to develop and verify programs.

ILP can be distinguished from traditional investigations of inductive inference in areas such as grammatical induction and induction of finite state automata [76, 13, 3] by its emphasis on the use of a universal representation. Clearly universal representations promise much wider scope of applicability. Logic programs are arguably much easier to manipulate for a machine learning algorithm than other universal representations which have been investigated, such as Universal Turing Machines programs [14] and LISP programs [133, 12]. This is due to the fact that in pure clausal logic changes can be made to a program by simply adding or deleting either complete clauses or literals within a clause without worrying about ordering effects. Since the semantics of logic programs are so closely allied to their syntax, such changes also have a clear and simple effect on the generality of the resulting program. In addition, logic programs allow a single representation for examples,

background knowledge and hypotheses.

In this paper, we provide an introduction to ILP. The introduction focusses on what we believe to be the foundations of the field. This paper is not a bottom-up paper based on describing small differences between many different systems. It is instead a top-down synthetic overview of concepts, terminology and methods. We are not overly concerned with discussing the implementation details of particular systems and approaches because the differences are often quite minor and of not great interest to a general audience. We aim instead at providing a conceptual framework for presenting ILP at four levels of description: a semantic level (defining the problem of ILP), a generic ILP algorithm level, a proof-theoretic level (defining the inference rules used in ILP), and a probabilistic semantics of belief (defining the justification of induced hypotheses).

The paper is organised as follows. In Section 2, we introduce inductive logic programming informally by means of some examples; in Section 3, we formally define the problem of inductive logic programming at the model-theoretic or semantic level; in Section 4, we provide a generic ILP algorithm, in Section 5 we study some inductive inference rules used in ILP, yielding a "proof-theory" for ILP; in Section 6, information compression techniques used within ILP are presented within a unifying Bayesian approach to confirmation and corroboration of hypotheses; in Section 7, we survey some methods to constrain the search-space in ILP (syntactic and semantic bias); in Section 8, the convergence and computationally complexity of ILP (learnability) is investigated; in Section 9, the problem of inventing new predicates is addressed; in Section 10 various ILP implementations are discussed and compared; in Section 11, some applications of ILP in scientific discovery and automatic programming are summarised; finally, in Section 12, we conclude. Appendix A contains a list of symbols and notations used throughout this paper.

2. General setting

Inductive inference is a very common form of everyday reasoning. Consider the following examples, which will be used throughout this paper.

2.1. Family example

Imagine yourself as learning about the relationships between people in your close family circle. You have been told that your grandfather is the father of one of your parents, but do not yet know what a parent is. You might have the following beliefs.

$$B = \left\{ \begin{array}{l} grandfather(X,Y) \leftarrow father(X,Z), parent(Z,Y) \\ father(henry, jane) \leftarrow \\ mother(jane, john) \leftarrow \\ mother(jane, alice) \leftarrow \end{array} \right.$$

You are now given the following facts (positive examples) concerning the relationships between particular grandfathers and their grandchildren.

$$E^+ = \left\{ \begin{array}{l} grandfather(henry, john) \leftarrow \\ grandfather(henry, alice) \leftarrow \end{array} \right.$$

You might be told in addition that the following relationships do not hold (negative examples).

$$E^- = \left\{ \begin{array}{l} \leftarrow grandfather(john, henry) \\ \leftarrow grandfather(alice, john) \end{array} \right.$$

Believing B, and faced with the new facts E^+ and E^- you might guess the following relationship.

$$H = parent(X, Y) \leftarrow mother(X, Y)$$

Note that H is not a consequence of B, and E^-. That is

$$B \wedge E^- \not\models \square \quad (prior\ satisfiability)$$

However, H allows us to explain E^+ relative to B. That is

$$B \wedge H \models E^+ \quad (posterior\ sufficiency)$$

Note that B and H are consistent with E^-. That is

$$B \wedge H \wedge E^- \not\models \square \quad (posterior\ satisfiability)$$

The question arises as to how it is possible to derive (even tentatively) the hypothesis H.

2.2. Another example: Tweety

Suppose that you know the following about birds:

$$B = \left\{ \begin{array}{l} haswings(X) \leftarrow bird(X) \\ hasbeak(X) \leftarrow bird(X) \\ bird(X) \leftarrow vulture(X) \\ carnivore(X) \leftarrow vulture(X) \end{array} \right.$$

Imagine now that an expedition to the upper Zaïre basin comes across a creature, which we shall call for convenience "Tweety". The expedition leader telegraphs you to let you know that Tweety has wings and a beak. This could be represented as the following logic program E^+.

$$E^+ = \left\{ \begin{array}{l} haswings(tweety) \leftarrow \\ hasbeak(tweety) \leftarrow \end{array} \right.$$

Even without any negative examples it would not take a very inspired ornithologist with belief set B to hazard the guess "Tweety is a bird". This can be written as

$$H = bird(tweety) \leftarrow$$

This might be seen by our ornithologist as a working hypothesis about Tweety. It could clearly be refuted if further evidence revealed Tweety to be made of plastic

(though this would require a more sophisticated belief set B'). Note that, as in the grandfather example, H allows us to explain E relative to B. That is

$$B \wedge H \models E^+$$

Note that the ornithologist would be unlikely to entertain the more speculative hypothesis "vulture(tweety)", even though this could also be used to explain all the evidence.

$$H' = vulture(tweety) \leftarrow$$

But how do we know from B and E^+ that H' is more speculative than H?

2.3. Sorting example

Inductive inference can also be viewed as a form of program synthesis. Imagine that a learning program is to be taught the logic program for "quick-sort". The following definitions are provided as background knowledge.

$$B = \left\{ \begin{array}{l} part(X, [], [], []) \leftarrow \\ part(X, [Y|T], [Y|S1], S2) \leftarrow Y =< X, partition(X, T, S1, S2) \\ part(X, [Y|T], S1, [Y|S2]) \leftarrow Y > X, part(X, T, S1, S2) \\ app([], L, L) \leftarrow \\ app([X|T], L, [X|R]) \leftarrow app(T, L, R) \end{array} \right.$$

The program is then provided with a set of positive ground examples of quick-sort, such as

$$E^+ = \left\{ \begin{array}{l} qsort([], []) \leftarrow \\ qsort([0], [0]) \leftarrow \\ qsort([1, 0], [0, 1]) \leftarrow \\ \ldots \end{array} \right.$$

together with some negative examples such as

$$E^- = \left\{ \begin{array}{l} \leftarrow qsort([1, 0], [1, 0]) \\ \leftarrow qsort([0], []) \\ \ldots \end{array} \right.$$

In this case we might hope that the algorithm would, given a sufficient number of examples, suggest the following clauses for "quick-sort".

$$H = \left\{ \begin{array}{ll} qsort([], []) \leftarrow & \\ qsort([X|T], S) \leftarrow & part(X, T, L1, L2), \\ & qsort(L1, S1), \\ & qsort(L2, S2), \\ & app(S1, [X|S2], S) \end{array} \right.$$

Indeed several ILP systems such as Golem [90] and FOIL [102] can learn this definition of quick-sort from as few as 6 or 10 examples. Although much background knowledge is required to learn quick-sort, the mentioned ILP systems are able to select the correct hypothesis from a huge space of possible hypotheses.

In some parts of the paper, we will also employ background theory B'. From B' and some examples it is easy to induce a permutation sort.

$$B' = \left\{ \begin{array}{l} perm([],[]) \leftarrow \\ perm(L,[X|P]) \leftarrow del(X,L,L1), perm(L1,P) \\ del(X,[X|T],T) \leftarrow \\ del(X,[Y|T],[Y|T1]) \leftarrow del(X,T,T1) \\ sorted([]) \leftarrow \\ sorted([X]) \leftarrow \\ sorted([X,Y|T]) \leftarrow X \leq Y, sorted([Y|T]) \end{array} \right.$$

2.4. Inductive Inference and the Philosophy of Science

The form of reasoning demonstrated in the last three examples is known as inductive inference and is very common within the natural sciences. Aristotle first describes it in his "Posterior Analytics". Francis Bacon, in discussing the empiricism of the new natural sciences in the 17th century (in Novum Organum) gave numerous examples of inductive inference as a paradigm for scientific method.

However, despite the efforts of philosophers such as Hume, Mill, Pierce, Popper and Carnap, the foundations of inductive reasoning are still much less clear than those of deductive mathematical logic. Since the 1970's several researchers from within Computer Science have attempted, with varying degrees of success, to find a logical basis for inductive inference. These researchers have included Plotkin [100], Shapiro [125, 126] and the new school of Inductive Logic Programming [81, 83, 107].

In this paper we will describe the theoretical basis of Inductive Logic Programming in the framework of first-order predicate calculus, Bayesian statistics and algorithmic complexity theory. Although the examples used generally only involve definite clauses, most results extend quite naturally to full clausal logic (see Section 3). The theory of ILP will be related to implementations and applications throughout the paper.

2.5. Hypothesis formation and justification

From the examples in Sections 2.1, 2.2 and 2.3 it is clear that the process of hypothesis formation (abduction) and hypothesis justification need further clarification. In this paper it will be assumed that

$$\text{Induction} = \text{Abduction} + \text{Justification}$$

Abduction. According to the philosopher Pierce, abduction is the process of hypothesis formation. This term is used within Logic Programming (eg. [52, 19, 51]) to denote a form of non-monotonic reasoning (see also Section 11.2.5). Pierce describes the basis of abduction as follows: given E and $E \leftarrow H$, hypothesise H. A more extensive definition appropriate for ILP will be given in Section 3.

Justification. The degree of belief ascribed to an hypothesis given a certain amount of evidence. Followers of Carnap talk of the degree of "confirmation", claiming that no absolute justification is possible. On the other hand a follower of Popper would not see there as being a problem of justification, but

rather a problem of deciding between competing hypotheses. They would therefore rather talk of corroboration. The term justification is used here to introduce the whole nexus of related problems. The problem of justification is discussed in detail in Section 6.

In fact scientific theory formation involves much more than the two elements of induction above. Facts must be gathered, experiments must be planned and alternative theories must be tested out. Abduction and justification can be seen as central components of this process. Several ILP applications (Section 11) have involved the discovery of new pieces of scientific knowledge from empirical evidence. ILP potentially also contributes to experimentation and testing of hypotheses [106].

3. Model-Theory of ILP

The logical elements (the semantics) involved in inductive inference will now be described, together with the relationships which should hold between them. We describe two different semantics for ILP: the *normal* and *non-monotonic* semantics, and we also discuss the *definite* semantics, which is – roughly speaking – a special case of the normal semantics.

Throughout the paper, we will employ the notion of syntactic bias (see Section 7). The syntactic bias defines the set of well-formed hypotheses and thus constitutes a parameter of any ILP task. Because the use of a syntactic bias is omni-present in ILP, we will not always write explicitly that we assume the hypotheses are well-formed with regard to this bias.

3.1. Normal semantics

Here, we will use a general setting for ILP and allow examples, background theory and hypotheses to be any (well-formed) logical formula.

The problem of inductive inference is as follows. Given is background (prior) knowledge B and evidence E. The evidence $E = E^+ \wedge E^-$ consists of positive evidence E^+ and negative evidence E^-. The aim is then to find a hypothesis H such that the following conditions hold.

Definition3.1. (normal semantics)

Prior Satisfiability. $B \wedge E^- \not\models \square$
Posterior Satisfiability. $B \wedge H \wedge E^- \not\models \square$
Prior Necessity. $B \not\models E^+$
Posterior Sufficiency. $B \wedge H \models E^+$

The Sufficiency criterion is sometimes named *completeness* with regard to positive evidence and the Posterior Satisfiability criterion is also known as *consistency* with the negative evidence.

In most ILP systems background theory and hypotheses are restricted to being definite. This *definite* setting is simpler than the general setting because a definite clause theory T has a unique minimal Herbrand model $\mathcal{M}^+(T)$, and any logical

formulae is either true or false in the minimal model. This setting is formalised in the definite setting of Definition 2.

Definition3.2. (definite semantics)

Prior Satisfiability. all $e \in E^-$ are false in $\mathcal{M}^+(B)$
Posterior Satisfiability. all $e \in E^-$ are false in $\mathcal{M}^+(B \wedge H)$
Prior Necessity. some $e \in E^+$ are false in $\mathcal{M}^+(B)$
Posterior Sufficiency. all $e \in E^+$ are true in $\mathcal{M}^+(B \wedge H)$

The special case of the definite semantics, where the evidence is restricted to true and false ground facts (examples), will be called the *example* setting. Notice that the example setting is equivalent to the normal semantics, where B and H are definite clauses and E is a set of ground unit clauses. The example setting is the main setting of ILP. It is employed by the large majority of ILP systems; it will also be the most important setting in this paper. The example setting is the one illustrated in Section 2.

The reason for allowing other evidence than examples in the definite semantics, is that it is often useful to allow general clauses as evidence (cf. [110, 107] and Section 11.2). Clausal evidence usually captures more knowledge than factual evidence consisting of only ground facts. For instance, in the family example of Section 2.1, the first positive example could be

$$grandfather(henry, john) \leftarrow father(henry, jane), mother(jane, john)$$

and the (positive) evidence could also include $\leftarrow$ *grandfather(X,X)*, stating that no-one is their own grandfather. Analogously, in the sorting example of Section 2.3, one could use *sorted(Y)* $\leftarrow$ *quicksort(X, Y)* and *quicksort(X,X)* $\leftarrow$ *sorted(X)* as positive evidence when the definition of *sorted* is in the background theory. The use of clausal evidence provides the learner with an incomplete or partial specification of the sorting predicate. This constrains the space of acceptable hypotheses. Positive evidence has to be true in the minimal model of the hypothesis and theory, whereas negative evidence has to be false in this setting.

3.2. The non-monotonic semantics

A *non-monotonic*[1] setting for ILP was introduced by Nicolas Helft [48] and Flach [39]; some variants were later considered by [7, 113, 114]. Here, we define a variant related to the normal setting and [113, 114].

In the non-monotonic setting of ILP, the background theory is a set of definite clauses, the evidence is empty, and the hypotheses are sets of general clauses expressible using *the same alphabet* as the background theory. The reason that the evidence is empty is that the positive evidence is considered part of the background theory and the negative evidence is derived implicitly, by making a kind of closed world assumption (realised by taking the minimal Herbrand model).

[1] The term "non-monotonic" was introduced by Helft in order to make a link with other forms of non-monotonic reasoning, because of the relation to the closed world assumption and its variants.

In the non-monotonic setting, the following conditions should hold for H and B:

Definition3.3. (non-monotonic semantics)

Validity: all $h \in H$ are true in $\mathcal{M}^+(B)$
Completeness: if general clause g is true in $\mathcal{M}^+(B)$ then $H \models g$
Minimality: there is no proper subset G of H which is valid and complete

The Validity requirement assures that all clauses belonging to a hypothesis hold in the database B, i.e. that they are true properties of the data. The Completeness requirement states that all information that is valid in the database should be encoded in the hypothesis. This requirement should also be understood with regard to a given *syntactic bias*, which determines the set of well-formed hypotheses (see Section 7). The Minimality requirement aims at deriving non redundant hypotheses.

To illustrate the non-monotonic setting, consider the following example (taken from [113]) and assume that a hypothesis is well-formed if it consists of clauses containing a single variable:

$$B = \begin{cases} male(luc) \leftarrow \\ female(lieve) \leftarrow \\ human(lieve) \leftarrow \\ human(luc) \leftarrow \end{cases}$$

A possible solution is then:

$$H = \begin{cases} \leftarrow female(X), male(X) \\ human(X) \leftarrow male(X) \\ human(X) \leftarrow female(X) \\ female(X), male(X) \leftarrow human(X) \end{cases}$$

To explain the differences between the example setting and the non-monotonic setting, let us consider

$$B_1 = \begin{cases} bird(tweety) \leftarrow \\ bird(oliver) \leftarrow \end{cases}$$

$$E_1^+ = flies(tweety) \leftarrow$$

An acceptable hypothesis H_1 in the example setting would be *flies(X)* $\leftarrow$ *bird(X)*. Notice that this clause realises an inductive leap as *flies(oliver)* is true in $\mathcal{M}^+(B_1 \wedge H_1)$. On the other hand, H_1 is not a solution in the non-monotonic setting as there exists a substitution $\theta = \{X \leftarrow oliver\}$ which makes the clause false (non-valid) in $\mathcal{M}^+(B_1 \wedge E_1^+)$. This demonstrates that the non-monotonic setting hypothesises only properties that hold in the database. Therefore the non-monotonic semantics realises induction by *deduction*. The induction principle of the non-monotonic setting states that the hypothesis H, which is, in a sense, *deduced* from the set of *observed* examples E and the background theory B (using a kind of closed world and closed domain assumption), holds for *all* possible sets of examples. This produces generalisation beyond the observations. As a consequence, properties derived in

the non-monotonic setting are more conservative than those derived in the normal setting.

The differences between the two settings are related to the closed world assumption. In most applications of the example setting in ILP [58, 91], only the set of positive examples is specified and the set of negative examples is derived from this by applying the closed world assumption, i.e. by taking $E^- = \mathcal{M}^-(B \wedge E^+)$.[2] In our illustration, this results in $E_1^- = \{\textit{flies(oliver)}\}$. Given this modified E_1^-, hypothesis H_1 cannot contribute to a solution in the normal setting. If on the other hand, we ignore the difference between background theory and examples and define $B_2 = \emptyset$, and $E_2^+ = B_1 \wedge E_1^+$ and $E_2^- = E_1^-$, then clause H_2 can also be part of a solution in the normal setting. Intuitively, this shows that solutions to problems in the normal setting, where the closed world assumption is applied, are also valid in the non-monotonic setting.

Theorem 1. Any hypothesis H posterior sufficient and posterior satisfiable for a background theory B, and examples E such that $E^- = \mathcal{M}^-(B \wedge E^+)$, is valid in the non-monotonic setting if $\mathcal{B}_P = \mathcal{B}(B \wedge H) = \mathcal{B}(B \wedge E^+)$.

PROOF. We prove that under these assumptions $\mathcal{M}^+(B \wedge E^+) = \mathcal{M}^+(B \wedge H)$.
Define $\mathcal{B}_P$ as $\mathcal{B}(B \wedge H)$
1) $\mathcal{M}^+(B \wedge E^+) \subset \mathcal{M}^+(B \wedge H)$ because E^+ is true in $\mathcal{M}^+(B \wedge H)$ (posterior sufficiency) and B is true in $\mathcal{M}^+(B \wedge H)$
so $E^+ \wedge B$ is true in $\mathcal{M}^+(B \wedge H)$
so $\mathcal{M}^+(E^+ \wedge B) \subset \mathcal{M}^+(B \wedge H)$
2) $\mathcal{M}^+(B \wedge H) \subset \mathcal{M}^+(B \wedge E^+)$ because $B \wedge H \wedge E^- \not\models \square$ (posterior satisfiability)
so $\mathcal{M}^+(B \wedge H) \cap \mathcal{M}^+(E^-) = \emptyset$
so $\mathcal{M}^+(B \wedge H) \cap \mathcal{M}^-(B \wedge E^+) = \emptyset$
so $\mathcal{M}^+(B \wedge H) \subset \mathcal{B}_P - \mathcal{M}^-(B \wedge E^+)$
so $\mathcal{M}^+(B \wedge H) \subset \mathcal{M}^+(B \wedge E^+)$ $\square$

The opposite does not always hold and this reveals the other main difference between the two settings. In the normal setting, the induced hypothesis can always be used to replace the examples because theory and hypothesis entail the observed examples (and possibly other examples as well). In the non-monotonic setting, the hypothesis consists of a set of properties holding for the example set. When using a language bias (cf. Section 7), which further restricts the (syntactic) form of clauses, there is no explicit guarantee concerning prediction. For instance in the non-monotonic setting (with a language bias restricting hypotheses to single clauses), hypothesis H_2 is a solution for B_1 and E_1^+. Nevertheless, it cannot be used to predict the example in E_1^+.

The non-monotonic semantics do not require the closed domain assumption to hold for the background theory and evidence. Indeed, for example, in a medical application, all patients should be completely specified, which means that all their symptoms and diseases should be fully described. Notice that this is different from requiring that the *complete* universe is described (i.e. all possible patients).

Although the non-monotonic and the normal semantics appear to be quite dif-

[2] $\mathcal{M}^-(T) = \{\overline{f} : f \in (\mathcal{B}(T) - M^+(T))\}$, i.e. the complement of the minimal Herbrand model of T, where $\overline{f}$ denotes the negation of f, where T is a definite clause program, and where $\mathcal{B}(T)$ is the Herbrand base of T.

ferent, it will turn out that some ILP techniques, such as refinement, apply to both frameworks. Also, the two semantics allow for a different kind of application, see also Section 11.2.

4. A generic ILP algorithm

In this section, we present a generic ILP algorithm based on the GENCOL model of [112]. The generic ILP algorithm makes abstraction of specific ILP algorithms and aims at providing the reader with a general understanding of ILP algorithms and implementations.

A first key observation leading towards a generic ILP algorithm, is to regard ILP as a search problem. This view of ILP follows immediately from the model-theory of ILP presented in Section 3. Indeed, in ILP there is a space of candidate solutions, i.e. the set of "well-formed" hypotheses (which constitutes the syntactic bias or the language bias of the problem, cf. Section 7), and an acceptance criterion characterizing solutions to an ILP problem. Following general artificial intelligence principles, one can solve ILP using a naive generate and test algorithm. This approach is known in the literature as the enumeration algorithm. However, as for other artificial intelligence problems, the enumeration algorithm is computationally too expensive to be of practical interest. Therefore, the question arises of how the space of possible solutions can be structured in order to allow for pruning of the search. In concept-learning and ILP [72, 125, 74, 112], the search space is typically structured by means of the dual notions of generalisation and specialisation.

In our view, generalisation corresponds to induction, and specialisation to deduction, implying that induction is viewed here as the inverse of deduction[3].

Definition4.1. A hypothesis G is more general than a hypothesis S if and only if $G \models S$. S is also said to be more specific than G.

In search algorithms, the notions of generalisation and specialisation are incorporated using inductive and deductive inference rules:

Definition4.2. A deductive inference rule $r \in R$ maps a conjunction of clauses G onto a conjunction of clauses S such that $G \models S$; r is called a specialisation rule.

As an example of deductive inference rule, consider resolution. Also, dropping a clause from a hypothesis realises specialisation.

Definition4.3. An inductive inference rule $r \in R$ maps a conjunction of clauses S onto a conjunction of clauses G such that $G \models S$; r is called a generalisation rule.

An example of an inductive inference rule is Absorption:

[3] In this paper, we stick to this – probably controversial – view because it offers a clear and operational framework for induction. This contrasts with alternative frameworks, which mainly rest on philosophical intuitions and have less clear logical formalisations.

$$\textbf{Absorption:}\ \frac{p \leftarrow A, B \qquad q \leftarrow A}{p \leftarrow q, B \qquad q \leftarrow A}$$

In the rule of Absorption the conclusion entails the condition. Notice that applying the rule of Absorption in the reverse direction, i.e. applying resolution, is a deductive inference rule. Other inductive inference rules generalise by adding a clause to a hypothesis, or by dropping a negative literal from a clause. Inductive inference rules, such as Absorption, are clearly not sound. The fact that they cannot be applied in an unrestricted fashion is against the spirit of logical inference.

This soundness problem can be circumvented by associating each hypothesised conclusion H with a label $L = p(H|B \wedge E)$ where L is the probability that H holds given that the background knowledge B and evidence E hold. A Bayesian approach to computing this conditional probability is given in Section 6. Assuming the subjective assignment of probabilities to be consistent, labelled rules of inductive inference are as sound as deductive inference. The conclusions are simply claimed to hold in a certain proportion of interpretations[4].

Generalisation and specialisation form the basis for pruning the search space. This is because :

- when $B \wedge H \not\models e$, where B is the background theory, H is the hypothesis and e is positive evidence, then none of the specialisations H' of H will imply the evidence. Each such hypothesis will be assigned a probability label $p(H'|B \wedge E) = 0$. They can therefore be pruned from the search.
- when $B \wedge H \wedge e \models \square$, where B is the background theory, H is the hypothesis and e is negative evidence, then all generalisations H' of H will also be inconsistent with $B \wedge E$. These will again have $p(H'|B \wedge E) = 0$.

For example, in the family example of Section 2.1, one should not consider specialisations of B as they will not imply the positive examples. On the other hand, in the sorting example of Section 2.3, one should not consider generalisations of the hypothesis *qsort(X,X)* $\leftarrow$ as it is inconsistent with some negative examples.

Given the above key ideas of ILP as search, inference rules and labeled hypotheses, a generic ILP system can now be defined:

Algorithm 1.
QH := `Initialize`
repeat
  `Delete` H *from* QH
  `Choose` *the inference rules* $r_1, ..., r_k \in$ `R` *to be applied to* H
  Apply the rules $r_1, ..., r_k$ *to* H *to yield* $H_1, H_2, ..., H_n$
  Add $H_1, ..., H_n$ *to* QH
  `Prune` QH
until `stop-criterion`(QH) *satisfied*

The algorithm works as follows. It keeps track of a queue of candidate hypotheses QH. It repeatedly deletes a hypothesis H from the queue and expands that hypotheses using inference rules. The expanded hypotheses are then added to the

[4]In the learning literature assignments of degrees of belief are usually more ad hoc than in Section 6 and are known as "inductive bias". Inductive bias is often taken to be a binary (accept/reject) assignment. However, "reject" can simply be viewed as a prior probability of zero.

queue of hypotheses QH, which may be pruned to discard unpromising hypotheses from further consideration. This process continues until the stop-criterion is satisfied.

In the above algorithm, the generic procedures are `type-written`. The algorithm has the following generic parameters:

- `Initialize` denotes the hypotheses started from.
- `R` denotes the set of inference rules applied.
- `Delete` influences the search strategy. Using different instantiations of this procedure, one can realise a depth-first (`Delete` = LIFO), breadth-first (`Delete` = FIFO) or best-first algorithm.
- `Choose` determines the inference rules to be applied on the hypothesis H.
- `Prune` determines which candidate hypotheses are to be deleted from the queue. This is usually realized using the labels (probabilities) of the hypotheses on QH or relying on the user (employing an "oracle"). Combining `Delete` with `Prune` it is easy to obtain advanced search strategies such as hill-climbing, beam-search, best-first, etc.
- The `Stop-criterion` states the conditions under which the algorithm stops. Some frequently employed criteria require that a solution be found, or that it is unlikely that an adequate hypothesis can be obtained from the current queue.

Notice that the above algorithm searches for solutions at the hypotheses level rather than at the clause level, as done by several algorithms such as FOIL [102] and GOLEM [90]. We take the more general approach here.

As an example of an instantiation of this algorithm consider the DUCE and CIGOL algorithms of [79, 89], which realize a hill-climbing search strategy. At the time `Delete` is invoked, the queue always contains a single hypothesis. Initially this hypothesis is $B \wedge E^+$. The inference rules are based on inverting resolution (see Section 5.4 for more details) and include the Absorption rule. In the Pruning phase, only the best hypothesis is kept, the others are discarded from the queue QH. Pruning is realized using a mixture of the minimal description length principle (see Section 6) and relying on the user (the "oracle") to decide whether a clause is true in the intended model or not.

The DUCE and CIGOL systems are representatives of the class of "specific-to-general" systems. These systems start from the examples and background knowledge, and repeatedly generalize their hypothesis by applying inductive inference rules. During the search they take care that the hypothesis remains satisfiable (i.e. does not imply negative examples). Other representatives of this class include ITOU [121], CLINT [107], MARVIN [124], GOLEM [90] and PGA [20].

The dual class of systems, which searches "general-to-specific", starts with the most general hypothesis (i.e. the inconsistent clause $\Box$) and repeatedly specializes the hypothesis by applying deductive inference rules in order to remove inconsistencies with the negative examples. During the search care is taken that the hypotheses remain sufficient with regard to the positive evidence. Systems of this type include FOIL [102], CLAUDIEN [113], MIS [125], MOBAL [54], GRENDEL [24] and ML-SMART [9].

The same search strategies are also valid in the non-monotonic setting (cf. [47, 113]). Indeed, in the non-monotonic setting one is interested in the boundary of maximally general hypotheses, true in the minimal model. Above the boundary,

the hypotheses will be false, and below that boundary they will either be false or non-maximal. To locate the boundary, one can search again specific-to-general or general-to-specific.

In the next two sections of this paper, we will give a detailed overview of the different types of inductive inference rules applied in ILP (the proof-theory of ILP, see Section 5), and provide a unifying framework that makes abstraction of specific labelling schemes employed in ILP (the probabilistic semantics of ILP, see Section 6). These two aspects lie at the heart of ILP. Other implementation aspects (such as search-strategy) usually follow from these two using general artificial intelligence principles.

5. Proof-Theory of ILP

In this section, we give a detailed overview of different frameworks for inductive inference rules. Remember from Section 4 that induction was viewed as the inverse of deduction. Given the formulae $B \wedge H \models E^+$, deriving E^+ from $B \wedge H$ is deduction, and deriving H from B and E^+ is induction. Therefore inductive inference rules can be obtained by inverting deductive ones. Since this "inverting deduction" paradigm can be studied under various assumptions, corresponding to different assumptions about the deductive rule for $\models$ and the format of background theory B and evidence E^+, different models of inductive inference are obtained. In the simplest model, θ-subsumption (see Section 5.2), the background knowledge is supposed to be empty, and the deductive inference rule corresponds to θ-subsumption among single clauses. Since the deductive inference rule based on θ-subsumption is incomplete with regard to implication among clauses, extensions of inductive inference under θ-subsumption have been recently studied under the header "inverting implication" (see Section 5.5). Extensions of θ-subsumption that take into account background knowledge are studied in Section 5.3. Finally, the most attractive but most complicated framework for inductive inference is studied in Section 5.4. This framework takes into account background knowledge and aims at inverting the resolution principle, the best-known deductive inference rule.

Before going into details about these different frameworks, we discuss the difference between inference rules and inference operators, which is important when searching the space of hypotheses.

5.1. Rules of inductive inference and operators

Recall from Section 4 that inference rules basically state what can be inferred from what. A well-known problem in artificial intelligence is that the unrestricted application of inference rules results in combinatorial explosions. To control the application of inference rules, artificial intelligence employs "operators" that expand a given node in the search tree into a set of successor nodes in the search. This together with the above properties of generalisation and specialisation discussed earlier motivates the introduction of specialisation and generalisation operators (see also [112]):

Definition5.1. A specialisation operator maps a conjunction of clauses G onto a set of maximal specialisations of S. A maximal specialisation S of G is a special-

isation of G such that G is not a specialisation of S, and there is no specialisation S' of G such that S is a specialisation of S'.

Definition5.2. A generalisation operator maps a conjunction of clauses S onto a set of minimal generalisations of S. A minimal generalisation G of S is a generalisation of S such that S is not a generalisation of G, and there is no generalisation G' of S such that G is a generalisation of G'.

In the spirit of restricting the application of inference rules, one usually imposes further conditions on the operators. Such conditions (see also below) require for instance that the generated hypotheses satisfy the language bias, that the operators be complete (generate all clauses in the language), etc.

5.2. θ-subsumption

We start discussing the simplest model of deduction for ILP: θ-subsumption as introduced by Plotkin[5].

Definition5.3. ([99, 100]) A clause c_1 θ-subsumes a clause c_2 if and only if there exists a substitution θ such that $c_1\theta \subseteq c_2$. c_1 is a generalisation of c_2 (and c_2 a specialisation of c_1) under θ-subsumption.

In this definition, clauses are seen as sets of (positive and negative) literals.
The θ-subsumption inductive inference rule is thus:

$$\textbf{θ-subsumption:}\ \frac{c_2}{c_1} \text{ where } c_1\theta \subseteq c_2.$$

For example, *father(X,Y)* ← *parent(X,Y), male(X)* θ-subsumes *father(jef,paul)* ← *parent(jef,paul), parent(jef,ann), male(jef), female(ann)* with $\theta = \{X = jef,\ Y = ann\}$.

5.2.1. Properties Some properties of θ-subsumption include (see [100, 99]):

Implication. If c_1 θ-subsumes c_2 then $c_1 \models c_2$. The opposite does not hold for self-recursive clauses: let $c_1 = p(f(X)) \leftarrow p(X); c_2 = p(f(f(Y))) \leftarrow p(Y)$; $c_1 \models c_2$ but c_1 does not θ-subsume c_2. Therefore deduction using θ-subsumption is not equivalent to implication among clauses, see also Section 5.5.

Infinite Descending Chains. There exist infinite descending chains,
e.g. *h(*X_1, X_2*)* ←
h(X_1, X_2*)* ← *p(*X_1, X_2*)*
h(X_1, X_2*)* ← *p(*X_1, X_2*),p(*X_2, X_3*)*
h(X_1, X_2*)* ← *p(*X_1, X_2*),p(*X_2, X_3*),p(*X_3, X_4*)*
...
This series is bounded from below by *h(X,X)* ← *p(X,X)*.

[5] A simplified form of θ-subsumption has been studied by Steven Vere [142].

Equivalence. There exist different clauses that are equivalent under θ-subsumption, e.g. *parent(X,Y)* $\leftarrow$ *mother(X,Y), mother(X,Z)* θ-subsumes *parent(X,Y)* $\leftarrow$ *mother(X,Y)* and vice versa. Because two clauses equivalent under θ-subsumption are also logically equivalent (implication), ILP systems should generate at most one clause of each equivalence class. For an extended discussion of equivalence see [69].

Reduction. To get around this problem, Plotkin defined equivalence classes of clauses, and showed that there is a unique representative (up to variable renamings) of each clause, which he named the *reduced* clause. The reduced clause r of a clause c is a minimal subset of literals of c such that r is equivalent to c. An algorithm to reduce clauses follows from this. ILP systems can get around the problem of equivalent clauses when working with reduced clauses only.

Lattice. The set of reduced clauses form a lattice, i.e. any two clauses have unique *lub* (the least general generalisation $-$ *lgg*, see also below) and any two clauses have a unique *glb*.

5.2.2. Operators Let us first discuss specialisation under θ-subsumption. Shapiro [125] introduced the notion of a refinement operator ρ for clauses, which corresponds to our notion of a specialisation rule under θ-subsumption with the restriction that G and S contain a single clause. Refinement operators basically employ two operations on a clause:

1. apply a substitution θ to the clause,
2. add a literal (or a set of literals) to the clause.

There are several issues in designing refinement operators. In the next definition, we assume a specific language bias $\mathcal{L}$ is used (see Section 7). Without loss of generality, we assume $\mathcal{L}$ has a most general element $\top$.

Definition5.4. (properties of refinement operators)

Global completeness. A refinement operator ρ (with transitive closure ρ^*) is globally complete for a language $\mathcal{L}$ if and only if $\rho^*(\top) = \mathcal{L}$, where $\top$ is the most general element in $\mathcal{L}$.

Local completeness. A refinement operator ρ (with transitive closure ρ^*) is locally complete for a language $\mathcal{L}$ if and only if $\forall c \in \mathcal{L} : \rho(c) = \{c' \in \mathcal{L} \mid c'$ is a maximal specialisation of c$\}$.

Optimality. A refinement operator ρ (with transitive closure ρ^*) is optimal for a language $\mathcal{L}$ 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 $c_2 \in \rho^*(c_1)$.

First, for reasons discussed above, it is desirable that only reduced clauses are generated by the refinement operators; such a refinement operator for full clausal logic was recently developed by Patrick van der Laag [61]. Secondly, to consider all hypotheses, operators should be globally complete (preferably, for a language containing only reduced clauses). Thirdly, if a heuristic general-to-specific search strategy (such as hill climbing in FOIL [102]) is employed, the operator should

be locally complete. If the operator is not locally complete, not all successors of a node (hypothesis) in the search space are considered. On the other hand, if a complete search strategy is used (such as breadth-first [125] or depth first iterative deepening [113]), it is desirable that the operator be optimal, because they generate each candidate clause exactly once. Non-optimal refinement operators, such as in Shapiro's MIS [125], generate all candidate clauses more than once, getting trapped in recomputing the same things again and again. Recently, an optimal refinement operator for full clausal logic was developed by Wim Van Laer [141] for use in the non-monotonic setting of CLAUDIEN [113].

The definitions of the properties of generalisation operators (for θ-subsumption and single clauses) can be derived from those of refinement operators. Neither a locally nor a globally complete generalisation rule for full clausal logic (and also definite clause logic) exists because of the infinite descending chains. Indeed, without additional assumptions about the language bias, the most specific generalisation of *h(X,X)* $\leftarrow$ *p(X,X)* under θ subsumption contains an infinite number of literals. Generalisation operators thus depend very much on the language bias employed. Therefore we do not discuss them any further here.

Although generalisation operators under θ-subsumption for single clauses under θ-subsumption do not exist for full clausal logic, a generalisation rule that starts from pairs of clauses does exist. This is the well-known least general generalisation rule of Plotkin [99], which computes the greatest lower bound of the two input clauses under θ-subsumption. To compute the *lgg* of two clauses, consider the following. The *lgg* of the terms $f(s_1, ..., s_n)$ and $f(t_1, ..., t_n)$ is $f(lgg(s_1, t_1), ..., lgg(s_n, t_n))$. The *lgg* of the terms $f(s_1, ..., s_n)$ and $g(t_1, ..., t_m)$ where $f \neq g$ is the variable v where v represents this pair of terms throughout. The *lgg* of two atoms $p(s_1, ..., s_n)$ and $p(t_1, ..., t_n)$ is $p(lgg(s_1, t_1), ..., lgg(s_n, t_n))$, the *lgg* being undefined when the sign or the predicate symbols are unequal. Finally, the *lgg* of two clauses c_1 and c_2 is then $\{lgg(l_1, l_2) \mid l_1 \in c_1 \text{ and } l_2 \in c_2\}$. For example, the *lgg* of *father(tom,ann)* $\leftarrow$ *parent(tom,ann), male(tom), female(ann)* and *father(jef,paul)* $\leftarrow$ *parent(jef,paul), male(jef), male(paul)* is *father(X,Y)* $\leftarrow$ *parent(X,Y), male(X), male(Z).*

5.3. Relative subsumption

Plotkin [100] extended the notion of θ-subsumption to that of relative subsumption as follows. First he defines c-derivations, which defines the deductive inference rule, i.e. the way $\vdash$ is implemented.

Definition5.5. A resolution-based derivation D of the clause c from the conjunction of clauses T is called a c-derivation if and only each clause in T appears at most once in D.

Plotkin then defines relative subsumption as follows.

Definition5.6. The conjunction of clauses T relatively subsumes the clause c if and only if there exists a c-derivation of a clause d from T such that d θ-subsumes c.

Like θ-subsumption, it is straightforward to define relatively reduced clauses using a straightforward definition of relative clause equivalence. Relative subsump-

tion forms a lattice over relatively reduced clauses. Plotkin defines the relative least general generalisation (rlgg) as follows.

Definition5.7. The least general generalisation of clauses c and d relative to T is the lub of c and d within the relative subsumption ordering.

Plotkin shows that the rlgg of two clauses is not necessarily finite. However, under the language bias of ij-determinacy introduced in [90], a unique, finite rlgg can be constructed.

Buntine [21] defined a special case of relative subsumption which he called *generalised subsumption*. Generalised subsumption is only applicable to definite clauses.

5.4. Inverting Resolution

As stated in Section 5.1 inductive inference rules can be viewed as the inverse of deductive rules of inference. Since the deductive rule of resolution is complete for deduction an inverse of resolution should be complete for induction. This idea of "inverse resolution" was first introduced for first-order logic in [89]. Several authors have expanded on these ideas [144, 49, 121, 136]. Four rules of inverse resolution were introduced in [79].

Absorption:

$$\frac{q \leftarrow A \qquad p \leftarrow A, B}{q \leftarrow A \qquad p \leftarrow q, B}$$

Identification:

$$\frac{p \leftarrow A, B \qquad p \leftarrow A, q}{q \leftarrow B \qquad p \leftarrow A, q}$$

Intra-construction:

$$\frac{p \leftarrow A, B \qquad\qquad p \leftarrow A, C}{q \leftarrow B \qquad p \leftarrow A, q \qquad q \leftarrow C}$$

Inter-construction:

$$\frac{p \leftarrow A, B \qquad\qquad q \leftarrow A, C}{p \leftarrow r, B \qquad r \leftarrow A \qquad q \leftarrow r, C}$$

In these rules lower-case letters are atoms and upper-case letters are conjunctions of atoms. Both Absorption and Identification invert a single resolution step. This is shown diagrammatically in Figure 1 as a 'V' with the two premises on the base and one of the arms. The new clause in the conclusion is then the clause found on the other arm of the V. For this reason Absorption and Identification were called collectively V-operators.

The rules of Inter- and Intra-construction introduce a new predicate symbol. Inductive inference rules which introduce new predicates are said to carry out "predicate invention" (see Section 9). When constructing logic programs such as "insertion sort", ILP systems such as CIGOL [89] use Intra-construction to introduce a new predicate "insert". The new predicate can then be generalised using a V-operator. Diagrammatically (see Figure 2) the construction operators can be shown as two linked V's, or a W, each respresenting a resolution. The premises are placed at the two bases of the W and the three conclusions at the top of the W. One of the clauses is shared in both resolutions. Intra- and Inter-construction are

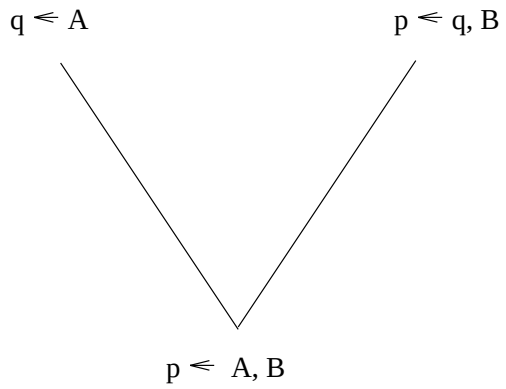

Figure 1. Absorption as a V-operator

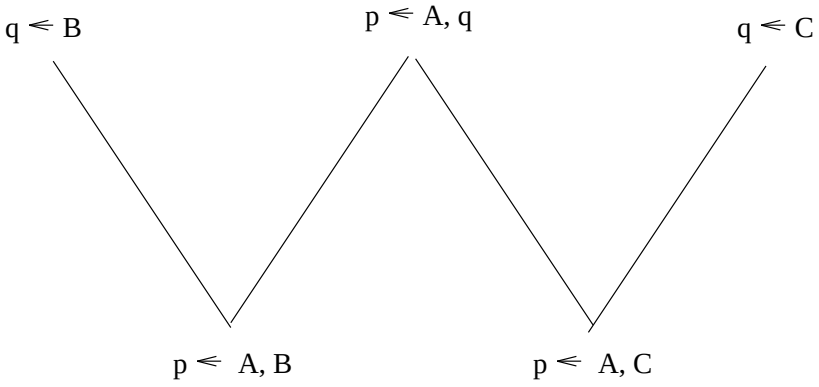

Figure 2. Intra-construction as a W-operator

collectively called W-operators.

The V and W operators have most specific forms (see Definition 2) as shown below (see also [81]).

Absorption↓:
$$\frac{q \leftarrow A \qquad p \leftarrow A, B}{q \leftarrow A \qquad p \leftarrow q, A, B}$$

Identification↓:
$$\frac{p \leftarrow A, B \qquad p \leftarrow A, q}{q \leftarrow A, B \qquad p \leftarrow A, q}$$

Intra-construction↓:
$$\frac{p \leftarrow A, B \qquad\qquad p \leftarrow A, C}{q \leftarrow A, B \qquad p \leftarrow A, q \qquad q \leftarrow A, C}$$

Inter-construction↓:
$$\frac{p \leftarrow A, B \qquad\qquad q \leftarrow A, C}{p \leftarrow r, A, B \qquad r \leftarrow A \qquad q \leftarrow r, A, C}$$

Note that in this form the V-operators realise both generalisation and specialisation since the conclusions entail the premises. Use of most specific operators is

usually implemented [122, 90] by having a two stage operation. In the first phase, inverse resolution operators are applied to examples (this is called *saturation* in [122]). In the second phase clauses are reduced by generalisation through the θ-subsumption lattice (see Section 5.2).

In [81] it was shown that the lgg of two examples e_1 and e_2 saturated relative to background knowledge B is equivalent to the rlgg of e_1 and e_2 relative to B. This result established a relationship between generalisations based on subsumption and those based on inverse resolution.

5.4.1. Matching subclauses Just as resolution requires unification to match terms, inverse resolution operators require a matching operation. In [122] all clauses, including the examples are "flattened". This involves introducing a new (n+1)-ary predicate for every n-ary function symbol. Thus the clause $member(a, [a, b]) \leftarrow$ becomes

$$member(U, V) \leftarrow a(U), dot(V, U, X), dot(X, Y, Z), b(Y), nil(Z).$$

Each new predicate symbol is then separately defined. For instance

$$dot([X|Y], X, Y) \leftarrow$$

After flattening the problem of matching clauses when applying the inverse resolution operators reduces to one-sided matching of clause bodies. In [81] saturation using most specific operations is shown to be complete with respect to Plotkin's c-derivations (see Section 6). This kind of completeness result was demonstrated independently in [121]. However, c-derivations do not cover all cases in which $B \models c$. The latter problem is known as *inverting implication*.

5.5. Inverting implication

Plotkin [100] was the first to show that θ-subsumption and implication between clauses are not equivalent. The difference between the two is important since almost all inductive algorithms which generalise first-order clauses invert θ-subsumption rather than implication. This inevitably leads to a form of incompleteness in these algorithms. In this section methods of constructing the inverse implicants of clauses are explored. In Section 5.5.4 it is shown how these methods can be extended to the problem of inverting implication in the presence of background knowledge. First the difference between Plotkin's θ-subsumption and implication between clauses will be reviewed.

Remember from Section 5.2, that whenever clause c θ-subsumes clause d it also implies d. However the converse does not hold. For instance Plotkin shows that with clauses

$$\begin{aligned} c &= p(f(X)) \leftarrow p(X) \\ d &= p(f(f(X))) \leftarrow p(X) \end{aligned}$$

c implies d, since d is simply c self-resolved. However c does not θ-subsume d. In discussing this problem Niblett [93] proves various general results. In particular he shows that there is not always a unique least generalisation under implication

of an arbitrary pair of clauses. For instance, the clause d above and the clause d' = $p(f(f(f(X)))) \leftarrow p(X)$ have both c and the clause $p(f(X)) \leftarrow p(Y)$ as least generalisations. Although Niblett claims that implication between Horn clauses is decidable, this has since been shown to be false by Marcinkowski and Pacholski [70].

Gottlob [42] also proves a number of properties concerning implication between clauses. Notably let c^+, c^- be the positive and negative literals of c and d^+, d^- be the same for d. Now if $c \models d$ then c^+ θ-subsumes d^+ and c^- θ-subsumes d^-.

5.5.1. Sub-unification The problem of inverting implication is discussed in a paper by Lapointe and Matwin [63]. They note that inverse resolution (Section 5.4) is incapable of reversing SLD derivations in which the hypothesised clause is used more than once. In fact Plotkin [100] showed that the same problem appears in the use of relative least general generalisation of clauses (see definition of c-derivations). Lapointe and Matwin go on to describe sub-unification, a process of matching sub-terms. They demonstrate that sub-unification is able to construct recursive clauses from fewer examples than would be required by ILP systems such as Golem [90] and FOIL [102]. For instance, given the atoms $append([], X, X)$ and $append([a, b, Y], [1, 2], [a, b, Y, 1, 2])$ sub-unification can be used to construct the recursive clause

$$append([U|V], W, [X|Y]) \leftarrow append(V, W, Y)$$

Unlike the approach taken originally with inverse resolution [89], Lapointe and Matwin do not derive sub-unification from resolution. Instead sub-unification is based on a definition of most general sub-unifiers. Although the operations described by Lapointe and Matwin are shown to work on a number of examples it is not clear how general the mechanism is.

A complete though non-deterministic algorithm is given for inverting implication in [84]. A complete and deterministic method is given by Idemstam-Almquist [50]. A new and simple inverse implication technique called "forced simulation" is described in [26].

5.5.2. Implication and resolution In this section the relationship between resolution and implication between clauses is investigated. Below a definition equivalent to Robinson's [120] resolution closure is given. The function $\mathcal{RL}$ below contains only the linear derivations of Robinson's function $\mathcal{R}$. However, the closure is equivalent up to renaming of variables given that linear derivation (as opposed to input derivation) is known to be complete.

Definition5.8. **(Resolution closure)**
Let T be a set of clauses. The function $\mathcal{RL}$ is recursively defined as

$$\begin{aligned} \mathcal{RL}^1(T) &= T \\ \mathcal{RL}^n(T) &= \{c \mid c_1 \in \mathcal{RL}^{n-1}(T), c_2 \in T, c \text{ is the resolvent of } c_1 \text{ and } c_2\} \end{aligned}$$

the resolution closure $\mathcal{RL}^*(T)$ is $\mathcal{RL}^1(T) \cup \mathcal{RL}^2(T) \cup \ldots$

5.5.3. Nth powers and nth roots of clauses The set of clauses constructed by self-recursing c, $\mathcal{RL}^*(\{c\})$, is partitioned into levels by the function $\mathcal{RL}$. By viewing resolution as a product operation Muggleton and Buntine [89] stated the problem of finding the inverse resolvent of a pair of clauses as that of finding the set of quotients of two clauses. Following the same analogy the set $c^2 = \mathcal{RL}^2(\{c\})$ might be called the squares of the clause c and $c^3 = \mathcal{RL}^3(\{c\})$ the cubes of c. The following definition from [84] captures this idea.

Definition5.9. **(nth powers of a clause)**
Let c and d be clauses. For $n \geq 1$, d is an nth power of c if and only if d is an alphabetic variant of a clause in $\mathcal{RL}^n(\{c\})$.

Taking the analogy a bit further one might also talk about the nth roots of a clause.

Definition5.10. **(nth roots of a clause)**
Let c and d be clauses. d is an nth root of c if and only if c is an nth power of d.

We now have: in terms of nth roots of a clause.

Corollary 1. **(Implication between clauses in terms of nth roots)** *Let c be an arbitrary clause and d be a non-tautological clause. $c \models d$ if and only if for some positive integer n, c is an nth root of a clause e which θ-subsumes d.*

It is fairly straightforward to enumerate the set of clauses which θ-subsume a given clause. Therefore the problem of finding the set of clauses which imply a given clause c reduces to that of enumerating the set of nth roots of clauses which θ-subsume c. The special case of clauses which immediately θ-subsume c occurs with $n = 1$. An algorithm for constructing nth roots is given in [84].

5.5.4. Implication and background knowledge In the normal setting of Inductive Logic Programming (Section 3.1) generalisation is carried out in the presence of background knowledge. In this section the solution to inverting implication between clauses is extended to the case in which background knowledge is present.

Assume a background clausal theory B and a clause (or example) c which is not entailed by B. Assume that there is a single clause d such that

$$B \wedge d \models c$$

This problem can be transformed to one involving implication between single clauses as follows.

$$\begin{aligned} B \wedge d &\models c \\ d &\models (B \rightarrow c) \\ &\models d \rightarrow (B \rightarrow c) \\ &\models d \rightarrow \overline{(B \wedge \overline{c})} \\ &\models d \rightarrow \overline{(l_1 \wedge l_2 \wedge \ldots)} \end{aligned}$$

In the last line $(B \wedge \overline{c})$ is replaced with a conjunction of all ground literals which can be derived from $(B \wedge \overline{c})$. This can be viewed as replacing the formula with a model of the formula. Since $(l_1 \wedge l_2 \wedge \ldots)$ is a conjunction of literals, the last line above represents implication between two clauses. The clause $(\overline{l_1} \vee \overline{l_2} \vee \ldots)$ can be constructed to be of finite length if B is range-restricted or generative (see [90]) and elements of the model are only constructed to a finite depth of resolution. This clause can then be used to construct c using an algorithm for constructing nth-roots.

6. Probabilistic semantics: confirmation and belief

According to Utgoff and Mitchell [139], bias is anything which influences how the concept-learner draws inductive inferences based on the evidence. There are two fundamentally different forms of bias: *declarative* bias, which defines the space of hypotheses to be considered by the learner, i.e. *what* to search, and *preference* bias, which determines *how* to search that space, which hypotheses to focus on, and which ones to prune, etc. In this section, we will discuss the probabilistic semantics of ILP, which underly any preference bias. The next section presents different forms of declarative bias.

Since there will generally be more than one candidate hypothesis which explains all the examples we need a sound basis for grading hypotheses, i.e. a preference bias. Many ILP algorithms, such as FOIL [102] use information based techniques to guide search. In this section the information compression techniques described in [80, 92, 27] are presented within a unifying Bayesian approach to confirmation and corroboration of hypotheses. The relationship between the probabilistic view and information view are shown from first principles. This general approach has the advantage of being applicable even when only positive examples are available.

6.1. Probability calculus

Unlike deductive inference, the conclusions of inductive inference are not assured to follow from what is known. Thus, each inductively inferred logical statement is accompanied by a degree of belief, or probability value (see Section 4).

The probability calculus, like the predicate calculus, has its basis in set theory. Figure 1 is a Venn diagram depicting the intersecting sets P and Q within the universal set U.

The probability of a randomly chosen element of U being in P, written $p(\mathrm{P})$ is defined as follows.

$$p(\mathrm{P}) = \frac{|\mathrm{P}|}{|\mathrm{U}|}$$

Similarly for Q. Given that a randomly chosen element of U is found within Q, the probability that it is also found within P, written $p(\mathrm{P}|\mathrm{Q})$, is

$$p(\mathrm{P}|\mathrm{Q}) = \frac{p(\mathrm{P} \cap \mathrm{Q})}{p(\mathrm{Q})}$$

$p(\mathrm{P}|\mathrm{Q})$, or the probability of P given Q, is known as a conditional probability. Noting that

$$p(\mathrm{P} \cap \mathrm{Q}) = p(\mathrm{P}|\mathrm{Q}).p(\mathrm{Q}) = p(\mathrm{Q}|\mathrm{P}).p(\mathrm{P})$$

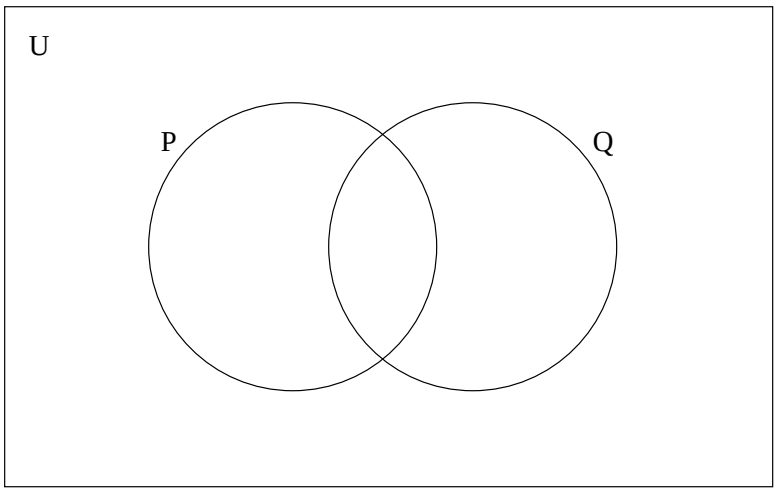

Figure 1. Venn diagram for probabilities

and rearranging gives Bayes' theorem,

$$p(\mathrm{P}|\mathrm{Q}) = \frac{p(\mathrm{P}).p(\mathrm{Q}|\mathrm{P})}{p(\mathrm{Q})}$$

Suppose that in Figure 1 P represents the set of all Herbrand models of the wff P, Q the same for the wff Q and U= $2^{\mathcal{B}(P \wedge Q)}$ the set of all Herbrand interpretations of $P \wedge Q$. Then $p(\mathrm{P}) = p(P)$ is simply the proportion of interpretations of P which are models of P. This is also the probability that a randomly chosen interpretation is a model of P. $p(P|Q)$ is the proportion of models of Q which are models of P. This probabilistic interpretation of first-order predicate calculus was suggested by the Philosopher of Science Carnap [22, 78] in the 1950s. It has the properties that

$p(\square) = 0$,
$p(\blacksquare) = 1$,
$p(P \wedge Q) = p(\mathrm{P} \cap \mathrm{Q})$,
$p(P \vee Q) = p(\mathrm{P} \cup \mathrm{Q})$,
$p(\overline{P}) = 1 - p(P)$ and
$p(P) \leq p(Q)$ if $P \models Q$.

However, $p(P)$ is undefined when P has an infinite set of Herbrand models.

6.2. Justification

Suppose we are attempting to induce a definition of the predicate p^n from positive examples only. Abduction will have two extreme solutions.

$$H = \top = p(x_1, .., x_n) \leftarrow$$
$$H = \bot = E^+$$

When negative examples are present, application of the posterior satisfiability condition (Section 3) will replace the unique topmost element by a set of topmost elements. In the following let

$$T = B \wedge H$$

Let us assume that our degree of belief in a formula can be represented as a subjective probability. We can therefore make use of Bayes' Theorem as follows.

$$p(T|E) = \frac{p(T).p(E|T)}{p(E)}$$

Below we assume that the evidence is correct and therefore $p(E) = 1$. As already mentioned, Carnap took the view that $p(T)$ is the proportion of interpretations which are models of T. This leads to the paradox that if T has a finite set of models among an infinite set of interpretations then $p(T) = 0$, i.e. T is necessarily false. Solomonoff [128] took an alternative approach to these probabilities by re-interpreting them in information terms. Any recursively enumerable set must have finite information since it can be denoted by a finite formula. However, a theory T for which $p(T) = 0$ has infinite information. In all other ways Solomonoff's syntactically-oriented approach provides a usable approximation to Carnap's probabilistic interpretation of logic formulae.

Like Carnap's interpretation Solmonoff's approach can be used to ascribe prior probabilities to logic programs. However, in Solomonoff's case $p(P) = 2^{-\sigma(P)}$ where $\sigma(P)$ is the number of bits in the minimum encoding of P (the information content of the formula P). In both Carnap and Solomonoff's case, since the number of logic programs is large and prior probabilities must sum to 1, the prior probability of any particular logic program will be very small. Larger logic programs can be composed of smaller logic programs by conjunction. When the models of two logic programs P and Q are independent (the 'average' case) $p(P \wedge Q) = p(P).p(Q)$. Even when P and Q are not independent $p(P \wedge Q)$ must be less in Carnap's interpretation than both $p(P)$ and $p(Q)$ (see Figure 1). According to Shannon's information theory, the information content of logic program P is $I(P) = -log_2 p(P)$. Using this definition we have the following properties for the information content of logic programs.

Empty program. $(I(\blacksquare) = 0)$ since $(p(\blacksquare) = 1)$.
Empty clause. $(I(\square) = \infty)$ since $(p(\square) = 0)$.
Additive composition. $(p(P \wedge Q) = p(P).p(Q))$ implies $(I(P \wedge Q) = I(P) + I(Q))$.

Note that additive composition assumes independence of P and Q. We have the following corollary of Bayes' Theorem.

Corollary 1. **Information Bayes.** *Let E represent the evidence for theory T. Then if $T \models E$ then*

$$I(T|E) = I(T) + I(E|T) - I(E)$$

Proof. Simply the log form of the Bayes' formula. □

It is possible that for certain T, $I(T|E) \geq I(B \wedge E)$. In this case we might say that T does not "compress" the examples, since it has greater information

content than the examples themselves. Random data (sometimes called noise) cannot be compressed. The principle of choosing the theory which minimises $I(T|E)$ is known as Rissanen's *minimum description length principle* (MDL)[119][6]. The MDL principle has been made use of in machine learning by Quinlan and Rivest [105]. MDL has been used in ILP in [80, 92, 27] and [102]. It is a generalisation of other Bayesian confirmation techniques such as those used in [32].

The following result shows that the choice of the theory with minimum description is equivalent to choosing the theory which has maximum Bayes' posterior probability. This is the same as Fisher's *maximum likelihood principle* (maximise $I(E|T)$) when the prior probability $p(T)$ is assumed to be the same for all T.

Theorem 1. **Equivalence of minimum description and maximum posterior probability.** *Let E be evidence for a set of potential theories chosen from Σ.*

$$min_{T \in \Sigma} I(T|E) = -log_2 max_{T \in \Sigma} p(T|E)$$

PROOF. Follows trivially from the fact that log_2 is monotone and $I(T|E) = -log_2 p(T|E)$.

Solmonoff's σ function is not computable due to halting. However a variety of good approximations to this approach are given in [102, 80, 92, 27]. The fine details of functions used are beyond the scope of this paper.

7. Declarative Bias

In this section, we will briefly discuss the most important forms of declarative bias. Current ILP systems, distinguish two kinds of declarative bias: *syntactic* bias (sometimes also called language bias) and *semantic* bias. Syntactic bias imposes restrictions on the form (syntax) of clauses allowed in hypothesis. To illustrate syntactic bias, let us consider abduction as it is usually perceived in logic programming. Roughly speaking, abduction can be considered the special case of the normal setting in inductive logic programming, where the syntactic bias restricts the hypotheses to positive ground unit clauses, where the positive evidence is a true ground fact, and the negative evidence a set of integrity constraints. Semantic bias imposes restrictions on the meaning, or the behaviour of hypotheses. To illustrate semantic bias, consider types and modes.

7.1. Syntactic Bias

Formally speaking, a syntactic bias defines the set of well-formed hypotheses $\mathcal{H}$. The set of well-formed hypotheses $\mathcal{H}$ is usually defined from a language bias $\mathcal{L}$, which is the set of syntactically acceptable clauses.

Since the syntactic bias of an ILP system determines the actual result, it is a very important parameter of an ILP system. Whereas previously, most ILP systems employed an implicit built-in syntactic bias, there is a growing interest in general

[6] Rissanen's principle is a variant of Jayne's *maximum entropy principle* but more sophisticated than William of Ockham's (1290-1349) *razor principle* which advocates minimising $I(T)$ rather than $I(T|E)$.

formalisms to specify syntactic bias. The advantage of such general formalisms is that language bias can be decoupled from particular ILP implementations. Hence, it becomes a true portable parameter of the system, which facilitates the comparison of different systems. In the remainder of this section, we first present four different frameworks for bias specification, and then briefly study the link between syntactic bias and the efficiency of ILP algorithms.

7.1.1. General frameworks for bias-specification At present there exist four more or less general frameworks to specify language bias, i.e. to specify the set of clauses allowed in hypotheses. This includes: the inductive logic programming language of Bergadano [8, 10], the antecedent description grammars of Cohen [24, 23], the schemata of the BLIP-MOBAL team [35, 54] and their variants [111, 127, 135]. The fourth framework, parametric languages as defined by [90, 107, 20, 25], will be presented when discussing the link to the complexity of learning.

Bergadano's inductive logic programming language uses a notation close to PROLOG as it aims mainly at applications in programming. It extends PROLOG by means of clause sets and predicate sets. As an example, consider the following expression:

$$\{father(X,Y) \leftarrow \{male(X),female(X)\},\ parent(X,Y);$$
$$mother(X,Y) \leftarrow \{male(X),female(X)\},\ parent(X,Y)\ \}$$

Set expressions, denoted using {}, are used to express that a subset of the literals or clauses may be present in the final hypothesis. The above expression denotes the hypotheses space consisting of all subsets of the following set of clauses:

$\{father(X,Y) \leftarrow male(X),female(X),\ parent(X,Y);$
$father(X,Y) \leftarrow female(X),\ parent(X,Y);$
$father(X,Y) \leftarrow male(X),\ parent(X,Y);$
$father(X,Y) \leftarrow parent(X,Y);$
$mother(X,Y) \leftarrow male(X),female(X),\ parent(X,Y);$
$mother(X,Y) \leftarrow female(X),\ parent(X,Y);$
$mother(X,Y) \leftarrow male(X),\ parent(X,Y);$
$mother(X,Y) \leftarrow parent(X,Y)\}$

Whereas the framework introduced by Bergadano aims at readability, the framework of Cohen aims at generality and computing power. Cohen employs a kind of definite clause grammar, which he calls antecedent description grammars, to describe the set of well-formed clauses. The above clauses can be encoded in this formalism as follows:

$goal\_formula(father(X,Y))).$
$goal\_formula(mother(X,Y))).$
$body(father(X,Y) \rightarrow m(X),f(X),\ [\ parent(X,Y)\ ]$
$body(mother(X,Y) \rightarrow m(X),f(X),\ [\ parent(X,Y)\ ]$
$m(X) \rightarrow []$
$m(X) \rightarrow [male(X)]$
$f(X) \rightarrow []$
$f(X) \rightarrow [female(X)]$

In this notation, *goal_formula* defines the predicates to be learned, and *body(P)* is the starting symbol of a grammar for learning clauses with as head P. As for definite clause grammars, square brackets enclose terminal symbols.

Another type of syntactic bias that is often used in inductive logic programming is a form of second-order schemata. Here, we present the formalism first introduced by Emde *et al.* [35] and later adapted or employed by [77, 147, 137, 111, 127, 54, 135]. Slightly different but related formalisms have been considered by [149, 38]. A second-order schema is basically a clause, where some of the predicate names are (existentially quantified) predicate variables. One such second order schema is e.g.

$$S = \exists p, q, r : p(X, Y) \leftarrow q(X, XW), q(YW, Y), r(XW, YW)$$

A set of second order schemata defines a language bias as the set of all clauses that can be obtained by instantiating a second order schema with a second order substitution. A second order substitution is a substitution that replaces predicate-variables by predicate-names.

For schema S, $\Theta = \{$ p = *connected*, q = *part-of*, r = *touches*$\}$ is a second order substitution. The instantiated schema $S\Theta$ yields :

connected(X, T) $\leftarrow$ *part-of(X,XW), part-of(YW, Y), touches(XW, YW)*

The above three ways of specifying bias, have the advantage that the specification is closely connected to the structure on the search space under θ-subsumption. Indeed, Kietz and Wrobel [54] showed, both theoretically and also in their MOBAL system, that second order schemata can be partially ordered and effectively searched using an extension of θ-subsumption, Cohen showed that generality can be determined at the sentential level (which are sentences containing both terminals and non-terminals) and effectively used to guide the search, and for Bergadano's formalism the structure of the search-space follows directly from the set notation. The three formalisms can therefore be easily used in the general-to-specific framework under θ-subsumption.

Finally, let us note that Cohen's formalism is the most powerful but least declarative framework, and that both Bergadano's framework and that of BLIP-MOBAL are complementary. Indeed, using Bergadano's framework it is easy to make abstraction of the number of literals in a clause whereas a language bias having a fixed number of literals would result in a huge number of expressions. Schemata are complementary in that the opposite is true. Therefore, it might be interesting to consider a straightforward generalisation of both models, where the set expressions also allow for predicate variables.

7.1.2. Syntactic bias and the complexity of the search. Earlier approaches [107, 90, 20] to bias specification employed a parametric approach, where a number of parameters determined the syntax of clauses in the hypotheses. The parametric approach has the advantage that it is easy to implement a *shift of bias* [107], which occurs when the learner changes the language bias. Changing the language bias may be necessary when there exists no solution within a certain syntactic bias. Using a parametric approach, shifting the bias can be realized by modifying the parameters in such a way that the language becomes more expressive.

In the parametric approach various parameters have been employed; many of them are rather straightforward and include criteria such as restrictions on the

maximum number of variables in a clause, the maximum number of literals in a clause, the predicates allowed in the hypotheses, etc.

Before presenting some of the more advanced notions, we introduce "linked" clauses [48].

Definition 7.1. A clause is linked if all of its variables are linked. A variable v is linked in a clause c if and only if v occurs in the head of c, or there is a literal l in c that contains the variables v and w (v $\neq$ w) and w is linked in c.

The linkage requirement is meant to exclude usually useless clauses such as, for instance, *p(X)* $\leftarrow$ *r(Z)*. A linked clause is for instance, *p(X)* $\leftarrow$ *q(X,Y),r(Y,Z),t(Z,W)*.

The following parameters are important as they determine the computational complexity of the learning.

Definition 7.2. (depth of term) The depth $d(V)$ of a variable V is 0. The depth $d(c)$ of a constant c is 1. The depth $d(f(t_1, ..., t_n))$ of a term $f(t_1, ..., t_n)$ is $1 + \max d(t_i)$.

Limiting the depth of terms in hypotheses to 1, corresponds to working with functor free clauses.

Definition 7.3. (level of a term) The level $l(t)$ of a term t in a linked clause c is 0 if t occurs as an argument in the head of c; and $1 + \min l(s)$ where s and t occur as arguments in the same literal of c.

The variable F in *father(F,C)* $\leftarrow$ *male(F), parent(F,C)* has level 0, the variable C in *father(F)* $\leftarrow$ *male(F), parent(F,C)* has level 1, the variable G in *grandfather(F)* $\leftarrow$ *male(F), parent(F,C),parent(C,G)* has level 2, etc. The level of a term corresponds to Muggleton and Feng's i parameter [90] and De Raedt's level of existential quantification [107].

Both the level and the depth of terms are frequently employed by ILP learners to define language restrictions, see for example [90, 107, 109, 25, 56]. The two notions are especially important in the context of specific-to-general ILP systems such as ITOU [121], GOLEM [90], CLINT [107] and PGA [20], because this class of learners starts learning from a so-called starting clause. The starting clause $SC(B, \mathcal{L}, e)$ is a function of the background theory B, the language bias $\mathcal{L}$ and a positive example e. $SC(B, \mathcal{L}, e)$ yields a most specific clause $c \in \mathcal{L}$ such that $B \wedge c \models e$.

For linked languages with maximum depth 1 and level > 1, the starting clause is unique, but the number of literals can grow exponential with its level, see Example 1.

Example 7.1. Let B be defined as follows:

$$B = \begin{cases} parent(jef, paul) \leftarrow \\ parent(jef, ann) \leftarrow \\ male(paul) \leftarrow \\ female(ann) \leftarrow \end{cases}$$

let e = is-a-father-of-son(jef), and let the clauses in the languages have a maximum

depth 1 and maximum level 2. The only starting clause is then:

is-a-father(jef) $\leftarrow$ *parent(jef,ann), parent(jef,paul),female(ann),male(paul)*

Therefore, specific-to-general systems being complete for these languages – without using additional (semantic) restrictions – are necessarily inefficient, cf. [56].

Also, starting clauses are not necessarily unique and the number of starting clauses can be exponential in the maximum number of variables allowed in clauses. This is illustrated in Example 2.

Example 7.2. Given the same backgroundknowledge and example as in Example 1 and clauses having a maximum of two variables, the following clauses are legal starting clauses:

is-a-father(F) $\leftarrow$ *parent(F,C), male(C)*
is-a-father(F) $\leftarrow$ *parent(F,C), female(C)*

It is easy to extend this example and show that the number of starting clauses can grow exponentially in the number of variables.

7.2. Semantic Bias

Although modes and types are usually employed to optimise the efficiency of Prolog compilers [71, 73, 18], they are also relevant to bias the set of acceptable hypotheses in inductive logic programming. Indeed, since Shapiro's MIS [125] it has become quite standard in inductive logic programming to provide the learner with *type* and *mode* declarations (cf. e.g. [65, 90, 130, 54, 145]).

Since modes and types are well-known in logic programming, we do not formalise them here, but rather illustrate their use on an example.

For example, the ILP system Progol [88] allows the user to specify declarations of the predicates in the background theory such as:

$mode(1, append(+list, +list, -list)$
$mode(*, append(-list, -list, +list)$
$list(nil) \leftarrow$
$list([X|T]) \leftarrow integer(X), list(T)$

The first mode states that the predicate *append* will succeed once (1) when the first two arguments are instantiated with lists and on return the third argument will be instantiated by a list. Types such as *list* are user-defined as monadic background predicates. The second declaration states that *append* will succeed finitely many times ($*$) when the third argument is instantiated by a list. The specified limit on the degree of indeterminacy of the call can be any natural number or $*$.

Modes are useful for inductive logic programming for two reasons. First, if we are in a single-predicate learning context, and the background predicates and their modes are correctly specified, the learner can guarantee termination by assuring that the queries it generates are mode-conform. Secondly, the learner can optimise its search when answering queries. Indeed, given the first declaration for *append*,

the learner does not need to backtrack after having found a first solution to a query matching the declaration.

Given type declarations of the predicate to be learned, the learner need only consider the type-conform subset of its hypothesis space. This can drastically reduce the computation needed.

Another semantic bias, employed by the ILP systems GOLEM [90], FOIL [104], and LINUS [65], is the notion of determinate clauses. Here, we adopt the simpler definition of [33] instead of the original one of [90].

Definition 7.4. (adapted from [33]) A definite clause $h \leftarrow l_1, ..., l_n$ is determinate (with respect to background knowledge B and examples E) if and only if for every substitution θ for h that unifies h to a ground instance $e \in E$, and for all $i = 1, \ldots, n$ there is a unique substitution θ_i such that $(l_1 \wedge ... \wedge l_i)\theta\theta_i$ is both ground and true in $\mathcal{M}^+(B)$.

Roughly speaking, a clause is determinate if all of its literals are determinate; and a literal is determinate if each of its variables that does not appear in preceding literals has only one possible binding given the bindings of its variables that appear in preceding literals.

To illustrate determinacy, reconsider the background theory B of Example 2. Here, the clause *has-father(Y)* $\leftarrow$ *parent(F, Y)* is determinate as given a Y there is a unique instantiation of F that is true. On the other hand, the clause *is-father(F)* $\leftarrow$ *parent(F, Y)* is not determinate as there exist two true instantiations of Y given F. Notice also that none of the clauses shown in Example 2 is determinate.

Determinate clauses are one way to get around some of the problems indicated in Examples 2 and 1. Indeed, some of the results in computational learning theory show that certain classes of determinate clauses can be learned efficiently (cf. [33] and Section 8). This however at the cost of losing completeness.

8. Learnability

The discussion in the sections so far has revolved around the process of hypothesis formation and justification. However it was noted in Section 2.5 that this is only a part of a larger scientific setting in which facts are gathered, experiments planned, and alternative theories tested. A simplified scenario of this kind is studied in the theory of "learnability". Learnability concerns itself with the convergence properties of a process of forming and revising predictive hypotheses. Two main approaches to learnability will be discussed in this chapter. These are

- Gold's [41] identification in the limit. This approach is derived from computability theory. It deals with finite time convergence of a computational learning procedure.
- Valiant's [140] Probably-Approximately-Correct(PAC) learning. This is derived from computational complexity theory and deals with the expected rate of convergence.

Current learnability results address only the definite and the example settings of inductive logic programming.

8.1. Identification-in the-limit

Both identification-in the-limit and PAC-learnability assume a predefined class of hypothesised theories $\mathcal{H}$, derived from the syntactic bias $\mathcal{L}$ which defines the clauses that can be part of a hypothesis. Here $\mathcal{H}$ will be assumed to contain only sets of definite clauses. A presentation of a definite clause theory T is defined as follows. Let $\mathcal{Q}^+(T)$ be the set of clauses true in $\mathcal{M}^+(T)$ and using the same alphabet as T; let $\mathcal{Q}^-(T)$ be the set of clauses false in $\mathcal{M}^+(T)$ and using the same alphabet as T; $\mathcal{Q} = \mathcal{Q}^- \cup \mathcal{Q}^+$.

Definition8.1. $E_\infty = \langle E_0, E_1, E_2, \ldots \rangle$ is a presentation of T if and only if $E_0 = \emptyset$ and for all $i \geq 1$, $E_i = E_{i-1} \cup \{e_i\}$ for an $e_i \in (\mathcal{Q}^+(T) \cup \mathcal{Q}^-(T))$ such that $\mathcal{M}^+(E_\infty) = \mathcal{M}^+(T)$[7].

The following is an ILP-oriented variant of Gold's definition.

Definition8.2. Let B be a definite clause background theory and $\mathcal{H}(B)$ be a class of definite clause theories. Let A be an ILP algorithm which given positive and negative evidence $E = E^+ \cup E^-$ returns an hypothesis $H' = A(B, E)$ such that posterior satisfiability and posterior sufficiency holds. Algorithm A identifies the class $\mathcal{H}(B)$ in the limit if and only if for each H in $\mathcal{H}(B)$ and presentation $E_\infty = \langle E_0, E_1, \ldots \rangle$ of H there is a finite i such that $\mathcal{M}^+(B \wedge H) = \mathcal{M}^+(B \wedge (A(B, E_j)))$ for all $j \geq i$.

The intuition behind Gold's formalism is that a certain class of learning tasks is "learnable" when there exists an algorithm that will find a correct hypothesis in finite time for all of these learning tasks if the algorithm is provided with enough evidence.

Gold gives various results showing that certain classes of theories cannot be identified in the limit. One such class is illustrated in Example 1, the example is adapted from [11].

Example8.1. Suppose $\mathcal{H} = \{H_1, H_2, H_3\}$, where the H_i are defined as follows:

$$H_1 = p(a) \leftarrow$$

$$H_2 = \begin{cases} p(b) \leftarrow \\ p(c) \leftarrow \end{cases}$$

$$H_3 = \begin{cases} p(b) \leftarrow \\ p(c) \leftarrow \\ p(d) \leftarrow \end{cases}$$

Given the presentation $\langle p(b), p(c), p(b), p(c), \ldots \rangle$ one cannot distinguish between H_2 and H_3, implying that even a finite class of finite ground unit clauses cannot be identified in the limit from *positive* examples only.

[7] Here, we implicitly assume that a unique minimal model of E_∞ exists, even though E_∞ may contain general clauses.

The main results in identification in the limit in ILP are due to Shapiro [125] and De Raedt [107, 110]. Shapiro proves that his MIS system (equipped with the eager search strategy) identifies any h-easy definite clause theory from a presentation consisting of (positive and negative) examples and an oracle to answer membership and existential questions. Roughly speaking, an h-easy definite clause theory is a definite clause theory for which there exists a function h from the Herbrand base to the natural numbers, which returns for a given fact, the maximum depth of the SLD-proof tree needed to prove that the fact is true. The value returned by h is used as a depth bound on the proof of the fact, in order to guarantee termination. A membership question asks the oracle for the truth-value of a ground fact; and an existential question asks the oracle for the truth-value of a non-ground fact. For membership questions, the oracle has to answer true or false. For existential questions, the oracle must answer with ground substitutions for which the fact is true, or with false, meaning that no instantiation of the fact is true.

De Raedt and Bruynooghe [110] upgraded Shapiro's result towards presentations using any presentation containing positive *clausal* evidence only[8]. In their adaptation of Shapiro's MIS, they restrict their attention towards functor free clauses. In [107], this restriction is – under certain conditions – lifted for the CLINT system. Other results in identification in the limit are due to Plotkin [100] and Banerji [4].

8.2. PAC-learnability

The following is a variant of Valiant's definition of PAC-learnability.

Definition8.3. Let B be a definite clause theory and $\mathcal{H}(B)$ be a class of definite clause theories. Let A be an algorithm which given positive and negative examples $E = E^+ \cup E^-$ returns an hypothesis $H' = A(B, E)$ in $\mathcal{H}(B)$ such that posterior sufficiency and posterior satisfiability holds. Let error$(B \wedge H', B \wedge H)$ be the probability that an example drawn from $\mathcal{B}(H)$ (see Section 3.2) according to distribution D is true in $\mathcal{M}^+(B \wedge H')$ and false in $\mathcal{M}^+(B \wedge H)$ or vice versa. Algorithm A PAC-learns the class $\mathcal{H}(B)$ if and only if for each H in $\mathcal{H}(B)$ and every probability distribution D of $\mathcal{B}(H)$, and all ϵ and δ, $0 < \epsilon, \delta < 1$, there is a polynomial function f such that for a random sample of examples $E \subseteq \mathcal{B}(H)$ of size at least $f(1/\epsilon, 1/\delta)$ drawn from distribution D, the probability that $H' = A(B, E)$ has error$(B \wedge H', B \wedge H) \leq \epsilon$ is at least $1 - \delta$.

8.3. PAC-learnability results in ILP

Learning-in-the-limit results are well-established in the ILP literature both for full-clausal logic [100] and definite clause logic [125, 4, 110, 107]. These results tell one little about the efficiency of learning. In contrast, Valiant's [140] PAC (Probably-Approximately-Correct) framework is aimed at providing complexity results for machine learning algorithms. Furthermore, the PAC-framework does not require convergence to a correct hypothesis, but rather to a hypothesis that is with high

[8] Notice that a negative example n in the definite setting can be expressed as positive evidence $\leftarrow n$.

probability $(1-\delta)$ approximately correct $(1-\epsilon)$, hence resulting in a more realistic framework.

Haussler's [46] negative PAC result concerning existentially quantified formulae seemed initially to exclude the possibility of PAC results for first-order logic. The situation has been improved by recent positive results in significant sized subsets of definite clause logic. These results have been possible for particular language biases (see Section 7). Namely, single constrained Horn clauses [94] (depth = 0, level = 0 in Section 7.1) and k-clause ij-determinate non-recursive single-predicate logic programs [33] under simple distributions. (k denotes maximum number of clauses in hypotheses, i denotes level, j denotes the maximum arity of predicates in the background knowledge and simple distributions are limited to those which are computable). Recursive ij-determinate predicates were shown to be PAC-learnable when membership queries are allowed. Thus the definition of quick-sort is PAC-learnable using membership queries.

Kietz [56] showed that the following languages are not PAC-learnable

- one-clause j-determinate programs, even without recursion
- one-clause ij-indeterminate programs, even without recursion

The second result disables the learning of the following simple non-determinate clause.

$$male(X) : -brother(X, Y).$$

However, Cohen [25] recently showed that single definite clauses with bounded indeterminacy and polynomial literal support are PAC-predictable (the same as PAC-learnable except that hypotheses do not have to be within $\mathcal{H}(B)$). Cohen's restriction on the indeterminacy of a single clause hypothesis is as follows.

Definition8.4. (l-indeterminate) A clause $h \leftarrow b_1, .., b_r$ is called l-indeterminate (with respect to background knowledge B and E) if and only if for every possible substitution σ of h to some ground instance $e \in E$ and for all $i = 1, .., r$ there are at most l distinct substitutions θ such that $(b_1 \wedge .. \wedge b_i)\theta\sigma$ is both ground and true in $\mathcal{M}^+(B)$.

Thus the clause above for defining male could be learned if a bound could be put on the maximum number of brothers and sisters any individual might be expected to have.

9. Predicate invention

The following theoretical characterisation of predicate invention follows that in [87]. If P is a logic program then the set of all predicate symbols found in the heads of clauses of P is called the definitional vocabulary of P or $\mathcal{P}(P)$. ILP has the following three definitional vocabularies.

Observational vocabulary: $\mathcal{O} = \mathcal{P}(E^+ \cup E^-)$
Theoretical vocabulary: $\mathcal{T} = \mathcal{P}(B) - \mathcal{O}$
Invented vocabulary: $\mathcal{I} = \mathcal{P}(H) - (\mathcal{T} \cup \mathcal{O})$

The learner carries out predicate invention whenever $I \neq \emptyset$.

9.1. Necessary predicate invention

Ling [68] discusses the conditions under which predicate invention is necessary. This requires the following addition to the satisfiability, necessity and sufficiency requirements of Section 3.

Necessary invention: $\mathcal{I} \neq \emptyset$ for each H which provides sufficiency and consistency.

In other words predicate invention is only necessary when there does not exist a finite axiomatisation of the predicates in $\mathcal{O}$ containing only predicate symbols from $\mathcal{T} \cup \mathcal{O}$. The following theorem is from Stahl [129].

Theorem 1. **Decidability with fixed vocabulary.** *Given a recursively enumerable, deductively closed set of formulas C in a first order language $\mathcal{L}$ it is undecidable whether C is finitely axiomatisable in $\mathcal{L}$.*

Stahl's proof is based on an application of Rice's Theorem [117] on the undecidability of non-trivial index sets being recursively enumerable. This result means that the necessity of invention must by needs be heuristic in the general case. However, if constraints on the language and depth of inference such as those discussed in Sections 7 and 8 are applied, this problem becomes decidable.

The following result due to Kleene [60] shows the importance of the introduction of new predicates in constructing finite axiomatisations.

Theorem 2. **Finite axiomatisation given additional vocabulary.** *Any recursively enumerable, deductively closed set C of formulas in a first order language $\mathcal{L}$ is finitely axiomatisable using additional predicate symbols other than those in $\mathcal{L}$.*

Although Kleene's proof is constructive it introduces new predicates regardless of whether they are necessary. Clearly any one of a potentially infinite set of new predicates could be introduced. It seems reasonable that when it is necessary to extend the vocabulary this should be done in as conservative a manner as is possible. To do so requires a notion of ordering over invented predicates.

In [87] a lattice of utility of invented predicates is introduced. The lattice has a unique topmost and bottom-most element. An equivalence class over the set of all possible invented predicates allows one to investigate only one of a set of invented predicates which are equivalent up to re-ordering of arguments and removal of redundant arguments. By making use of least-upper-bound and greatest-lower-bound operators, this utility lattice should provide a sound and complete approach to searching for invented predicates.

9.2. Predicate invention techniques

Most ILP systems which carry out predicate invention [79, 89, 122, 5, 68] are based on use of the inverse resolution W-operators (see Section 5.4). This necessarily involves a specific-to-general search.

An exception to this approach is found in [145] and [147] in which a general-specific search is employed. The search is guided by the use of mode declarations in [145] (see Section 7.2).

In [62], the authors use W-operators to introduce new predicates. The auxiliary sub-predicates are then generalised using inverse implication (see 5.5). This allows certain sub-predicates to be learned which could not have been learned otherwise.

10. ILP implementations

Up till now, we discussed – what are in our view – the foundations of the field of inductive logic programming, in particular, the model-theory, the proof-theory, the probabilistic semantics, the bias, the notions of predicate invention and learnability. The underlying assumption is that these foundations lie at the heart of ILP and are sufficient for understanding ILP. As a consequence, we ignored several other issues in ILP, mainly because they are closely connected to particular ILP implementations and applications. This includes for instance the use of an oracle, theory revision, and the handling of numerical data. At the same time, we also did not study any particular ILP system in detail. In this section, we will briefly touch on these two matters. First, we will discuss some dimensions and issues of ILP as perceived by users of ILP systems. Secondly, we will give a short overview of some selected ILP systems.

10.1. Characteristics of ILP systems

Practical ILP systems can be classified along different dimensions as perceived by users of ILP systems. Obvious characteristics, studied earlier in this paper, include the types of bias employed, the ability to invent new predicates, and the heuristics employed to handle imperfect data and noise.

10.1.1. Incremental/non-incremental This dimension describes the way the evidence E (examples) is obtained. In non-incremental or empirical ILP, the evidence is given at the start and not changed afterwards, in incremental ILP, the examples are input one by one by the user, in a piecewise fashion. Non-incremental systems search typically either specific-to-general or general-to-specific. Incremental systems usually employ a mixture of these strategies as they may need to correct earlier induced hypotheses. Incremental ILP systems include MIS [125], CLINT [107], MOBAL [54], FORTE [118], RX [134], LFP [144], and CIGOL [89]. Non-incremental systems include GOLEM [90], FOIL [102], FOCL [95], GRENDEL [24], CLAUDIEN [113], mFOIL [32], and LINUS [66].

10.1.2. Interactive/ Non-interactive In interactive ILP, the learner is allowed to pose questions to an oracle (i.e. the user) about the intended interpretation. Usually these questions query the user for the intended interpretation of an example or a clause. The answers to the queries allow to prune large parts of the search space (in the generic algorithm queries would normally be generated in the procedure `Prune`). Obviously, interactiveness implies incrementality. Most systems are non-interactive. Interactive systems include CIGOL [89], MIS [125], and CLINT [107].

10.1.3. Single/Multiple Predicate Learning/Theory Revision Suppose $\mathcal{P}(F)$ represent the predicate symbols found in formula F. In single predicate learning from examples, the evidence E is composed of examples for one predicate only, i.e. $\mathcal{P}(E)$ is a singleton. In multiple predicate learning, $\mathcal{P}(E)$ is not restricted as the aim is to learn a set of possibly interrelated predicate definitions. Theory revision is usually a form of incremental multiple predicate learning, where one starts from an initial approximation of the theory. Although theory revision systems have been around ever since MARVIN [124], MIS [125], followed by Banerji [4], BLIP-MOBAL [147], ML-SMART [9], CIGOL [89] and CLINT [108], there has recently been a renewed interest in theory revision and multiple predicate learning, cf. [2, 1, 6, 115, 118, 146, 28, 10, 134, 113, 148, 110]. These newer approaches differ from the previous ones in the sense that they try to learn without requiring an oracle. Note that also ML-SMART and BLIP-MOBAL did not require an oracle. Although it is commonly believed that theory revision and multiple predicate learning algorithms are fundamentally different from single predicate learners, both types of systems fit in a natural way in the generic algorithm outlined in Section 4. The main differences between theory revision systems and single predicate learners are the following. Theory revision systems typically use a variety of deductive and inductive inference rules, e.g. combining abduction, with specialisation and generalisation. Secondly, as for incremental systems, they can both generalise and specialise. Specialisation occurs when a negative example is implied by the hypothesis, and generalisation when a positive example is not implied. Finally, in theory revision it is important to modify the theory as little as possible, and to stay as close to the original theory as possible. This issue is formalised in the recent work of Stefan Wrobel [148].

10.1.4. Numerical data The mesh domain (Section 11.1.5) involves predicting the number of sections that an edge of a CAD object should be broken into for efficient finite-element analysis. The rules developed by GOLEM [90] have the following form.

$$mesh(Obj, 8) \leftarrow connected(Obj, Obj1), \ldots$$

With a small number of examples it is hard to get enough examples in which the prediction is an exact number, such as 8. Instead we would like the rules to predict an interval such as

$$mesh(Obj, X) \leftarrow 7 \leq X \leq 9, connected(Obj, Obj1), \ldots$$

This kind of construction is not handled elegantly by existing systems (though LINUS [66] and more recently FOIL [104] can use TDIDT-extensions [101] to introduce tests such as $X \leq 9$). In statistics this problem of numerical prediction is known as regression. Many efficient statistical algorithms exist for handling numerical data. ILP system designers are starting to look at smoothly integrating such approaches into their systems. Recent work on introducing linear inequalities into inductively constructed definite clauses [75, 53] provides an elegant logical framework for this problem. This approach also allows the introduction of Constraint Logic Programming (CLP) techniques into ILP.

10.2. ILP Systems

In this section, we give an overview of a number of important inductive logic programming systems. It is clear that a complete overview of all systems is outside the scope of this paper, given the very large number of ILP systems and implementations. Instead, the overview centres around the following 6 systems: MIS [125], MOBAL-BLIP [54], CIGOL [89], GOLEM [90], FOIL [102], and CLAUDIEN [113]. These systems were selected because they are fundamentally different, and contributed significantly to inductive logic programming. Furthermore, most of the other systems are very much related to these 6.

One of the first real inductive logic programming systems[9] in the sense that it was related to I as well as LP and involved both theory and implementation, is the MIS system of Ehud Shapiro [125]. The MIS system introduced several important techniques in inductive logic programming. These include refinement graphs (see Section 5.2.2) for general to specific search, the backtracing algorithm to locate incorrect clauses in programs, identification in the limit of h-easy programs (see Section 8), the handling of multiple predicates (realizing theory revision) and coping with functors in definite clause programs (i.e. realizing program synthesis from examples). Many other systems and techniques are related to MIS, e.g. LFP [143], CLINT [107], SIERES [145], FORTE [118], AUDREY [146], MIST [59], TR [1], those of [4, 110], RX [134], MARKUS [43] and others. Important developments in MIS type systems include: the introduction of the definite semantics (using clausal evidence instead of merely examples) by [110, 107], relating the MIS to intensional knowledge-base updating [110] (see also Section 11.2), the introduction of predicate invention techniques in [145], the elimination of the questions to the user (oracle) in [118, 146, 1, 134], and the introduction of specific to general search in [107].

Whereas the MIS originated from an interest in (logic) program synthesis and computational learning theory (or inductive inference), the MOBAL system [54], and its predecessors BLIP [77, 147, 137, 34] and METAXA [35], originates from a knowledge acquisition and knowledge discovery perspective. The main contribution of this line of research is the introduction of second order schemata and the associated theory of model driven learning, which now yields practical knowledge acquisition tools. Although schemata were originally only meant to specify syntactic bias, schemata (and their variants) have proven to be useful for other purposes as well. This includes the learning of syntactic bias, i.e. higher order logic learning [38, 111], predicate invention [111], intelligent (general to specific) search aids [127], and analogical reasoning mechanisms [149].

The first real "inverse resolution" operator (see Section 5.4), was the absorption operator, employed by the MARVIN system [124]. However, in MARVIN, the underlying theory of inverting resolution was not yet formalised. This was first done for propositional logic in DUCE [79] and later for definite clause logic in CIGOL [89] and LFP [143]. This paradigm was further explored by [144, 122, 121, 81]. However, in many ways the most innovative extension introduced in DUCE and CIGOL was the concept of predicate invention. This was not present in the earlier

[9] A full historic overview of inductive logic programming, is outside the scope of this paper. However, a personal view (by Claude Sammut) of the developments that lead to inductive logic programming is contained in [123].

frameworks of either Plotkin [100] or Shapiro [125]. One of the new departures in this line of research has come from the LOPSTER system [63] which was the first to make use inverting implication (see Section 5.5).

The GOLEM system [90] was based on a special case of inverse resolution which corresponds to the *rlgg* operator of Gordon Plotkin [98, 100], see also [81]. This special case of inverse resolution restricted to *determinate* clauses (cf. Section 7) proved to be much more tractable than the more complicated inverse resolution paradigm. It is in part because of the increased efficiency that GOLEM could be applied to real scientific discovery tasks [58, 91]. The determinacy restriction proved also to be relevant for the computational complexity of the learner [56, 33, 25].

The FOIL system [102] is based on traditional concept-learning techniques. In particular, it relates to the greedy TDIDT-algorithms [101] and the AQ family of algorithms of Michalski [72]. As a matter of fact, the use of relations to express background knowledge, when learning concepts was already present in the Induce algorithm of [72], and in the ML-Smart system of [9]. However, these algorithms – adopting the classical concept-learning framework – produced rules for a fixed number of classes only. As a consequence, AQ and Induce learned a kind of functor free definite clauses with propositions in the condition part. Furthermore, AQ and Induce employed a non-standard logic to represent concepts and examples. The main contribution of Quinlan in FOIL was to recognize the power of logic programming as a representation language for inductive learning and to upgrade machine learning techniques towards the much more expressive DATALOG representation. Furthermore, the FOIL system was the first widely known demonstration that first order learning could really work, in the sense that it works efficiently on a broad range of problems involving large and noisy datasets. More recently [103] FOIL incorporated the ij-determinate constraint introduced first in [90]. Many variants and refinements of FOIL have been developed, including FOCL [17], mFOIL [32], and CHAM [57]. Related to traditional concept-learning techniques and FOIL, is also LINUS of [65], which transforms certain classes of ILP problems into attribute value form, then runs classical algorithms and transforms the result back into logical clauses.

The CLAUDIEN system [113] is the first efficient inductive logic programming working in the non-monotonic setting deriving full clausal theories from databases. CLAUDIEN is based on a simple general to specific iterative deepening search using refinement under θ-subsumption. At the same time, it offers a natural approach to empirically learning multiple predicates, which requires interaction with the user, or "good" presentations in the normal setting (see [114, 115]). Indeed, in the non-monotonic setting it is easy to learn multiple predicates, because if two clauses c_1 and c_2 are valid, then their conjunction is also valid. This is in contrast to the normal setting, where the conjunction of two clauses (contributing individually to a solution) may violate the posterior satisfiability requirement. It remains however to be seen whether this new approach will yield as successful applications as GOLEM, MOBAL and FOIL.

11. Application areas

Other computational techniques, such as neural networks, are said to mimic human learning. In a sense neural networks, along with techniques such as statistical

regression, can be viewed as making use of a form of inductive inference. However, unlike neural networks, ILP algorithms output rules which are easily understood by people. This makes ILP particularly appropriate for scientific theory formation tasks in which the comprehensibility of the generated knowledge is essential to the advancement of scientific subjects.

The use of a relational logic formalism has allowed successful application of ILP systems in a number of domains in which the concepts to be learned cannot easily be described in an attribute-value language. These applications include structure-activity prediction for drug design [58, 131], protein secondary-structure prediction [91], finite element mesh design [31] and automatic construction of qualitative models [16]. It is worth noting that the results produced by ILP 1) did produce new knowledge publishable in refereed journals of the application area (as for the drug design and protein folding); 2) are understandable and meaningful for scientists in the application domain; 3) were realized using general purpose ILP systems. There are very few other examples within AI where this combination has been achieved.

Programming assistants are tools that assist a programmer in the design and implementation of software. The most straightforward application of ILP to this area is program synthesis from examples, bias and partial specifications (see e.g. [125, 104, 59, 107, 40]). Other applications include algorithmic debugging [125], program testing and verification [36, 116], the automatic derivation of properties of programs and/or databases [15, 113], reverse engineering [15] and knowledge-base updating [110].

11.1. Scientific Discovery and Knowledge Acquisition

11.1.1. Drug design The majority of pharmaceutical R&D is based on finding slightly improved variants of patented active drugs (292 out of 348 US drugs introduced between 1981 and 1988 were of this kind). This involves laboratories of chemists synthesising and testing hundreds of compounds almost at random. The ability to automatically discover the chemical properties which affect the activity of drugs could provide a great reduction in pharmaceutical R&D costs. The average cost of developing a single new drug is $230 million.

In [58] it was shown that ILP techniques are capable of constructing rules which predict the activity of untried drugs. Rules are constructed from examples of drugs with known medicinal activity. The accuracy of the rules was found to be higher than for traditional statistical methods. More importantly the easily understandable rules can provide key insights, allowing considerable reductions in the numbers of compounds that need to be tested.

11.1.2. Protein primary-secondary shape prediction Predicting the three-dimensional shape of proteins from their amino acid sequence is widely believed to be one of the hardest unsolved problems in molecular biology. It is also of considerable interest to pharmaceutical companies since shape generally determines the function of a protein. ILP techniques developed at the Turing Institute have recently had considerable success within this area. Over the last 20 years many attempts have been made to apply methods ranging from statistical regression to decision tree and neural net learning to this problem. Published accuracy results for the general prediction problem have ranged between 50 and 60 %, very close to random prediction. In [91] it was found that the ability to make use of biological background knowledge

together with the ability to describe structural relations boosted the predictivity for a restricted sub-problem from around 70% to around 80% on an independently chosen test set.

11.1.3. Satellite diagnosis ILP techniques have been applied to problems within the Aerospace industry. In this case a complete and correct set of rules for diagnosing power supply failures was developed by generating examples from a qualitative model of the power sub-system of an existing satellite [37]. The resulting rules are thus guaranteed complete and correct for all single faults since all examples of these were generated from the original model. Rules were described using a simple temporal formalism in which each predicate had an associated time variable.

11.1.4. Rheumatology An application [67] of the LINUS system [66] to the learning of medical rules for the early diagnosis of rheumatic diagnosis showed that relational background knowledge provided by a domain expert substantially improved the quality of the induced rules as compared to results with attribute value learning techniques.

11.1.5. Finite element meshes Successes [31] achieved in applying Golem to Finite Element Mesh design have drawn interest from industry in applying these techniques within state-of-the-art CAD packages. Finite element methods are used extensively by engineers and modelling scientists to analyse stresses in physical structures. These structures are represented quantitatively as finite collections of elements. The deformation of each element is computed using linear algebraic equations. In order to design a numerical model of a physical structure it is necessary to decide the appropriate resolution for modelling each component part. Considerable expertise is required in choosing these resolution values. Too fine a mesh leads to unnecessary computational overheads when executing the model. Too coarse a mesh produces intolerable approximation errors. ILP techniques have been used to develop rules for deciding on appropriate resolution values inductively from expert provided examples.

In this case a relational language was required to reflect the relations between elements of the physical structure being modelled.

11.2. Programming Assistants

Here, we study the relation between interests in logic programming and ILP. At several places, we argue for a tighter interpretation of the $ILP = I \cap LP$ paradigm. Such an interpretation allows us to import ILP into LP, and to export LP to inductive techniques in general; thus permitting cross-fertilization. We discuss the application of this claim in logic program synthesis, reverse engineering, algorithmic debugging, deductive databases and program testing and verification.

11.2.1. Logic program synthesis Logic program synthesis and transformation [30, 64] tries to develop techniques to derive efficient programs from a specification (synthesis) or an inefficient implementation (transformation). Usually logic program synthesis and transformation employ deductive techniques to achieve this aim. Here, we will show that alternatively, one could use induction. This has the advantage that logic program synthesis from incomplete specifications is plausible.

Let us briefly illustrate this on a the sorting example. In a logic program synthesis or transformation setting, the predicate *sort* might be specified by :

$S = sort(X,Y) \leftrightarrow permutation(X,Y),\ sorted(Y)$

with corresponding definitions of *permutation* and *sorted* (see Section 2.3) in the background theory B. The aim would then be to improve the definition of permutation sort towards a more efficient sorting predicate such as quicksort, insertion sort or bubble sort. The techniques to achieve this aim in logic program synthesis and transformation are basically deductive, for instance using fold-unfold or mathematical induction techniques.

In the ILP setting, one could tackle the same problem by taking:

$$E^+ = \begin{cases} permutation(X,Y) \leftarrow sort(X,Y) \\ sorted(Y) \leftarrow sort(X,Y) \\ sort(X,Y) \leftarrow permutation(X,Y), sorted(Y) \end{cases}$$

B then includes sorted and permutation and possibly other predicates such as partition, append, member, etc.; and the language bias is such that permutation and sorted are not to be used in hypotheses. Any definition of sort satisfying the requirements will be equivalent to the specification (i.e. permutation sort) and therefore correct. Also, depending on the predicates in the background theory and the bias, different definitions for sort could be derived. For example, if partition is in the background theory, one could induce quicksort. This shows that ILP can be used to derive logic programs from complete specifications. On the other hand – and this is shows the flexibility of ILP – by relaxing the evidence, ILP can also induce programs from incomplete specifications, which is not possible by most synthesis approaches (but see [40]). For instance the third clause in E^+ could be replaced by a few positive examples. A disadvantage of using ILP techniques for logic program synthesis, is that there is no guarantee that the induced hypothesis will be more efficient in use than the original specification. This should be verified empirically.

An extreme case of the application of ILP to this area is programming synthesis from examples only. Although such automatic programming has been used by many ILP researchers (cf. [90, 125, 107, 104]) to test and illustrate their techniques, we do not believe program synthesis from examples *only* to be a promising direction for ILP. This is because too many examples are needed before the correct definition can be induced (cf. e.g. [104]). Therefore, automatic programming from examples only will never be practical because it is much easier to program manually than to specify hundreds (thousands) of examples. At the same time, it follows that upgrading the representation of examples as ground facts to more general formulae should be one of the prime concerns in ILP (cf. [110, 107]).

11.2.2. Inducing properties of programs/databases Given a database or a program, one is often interested in the regularities in the database or program. High-level regularities satisfying a program can be regarded as (partial) specifications of that program. Such specifications can then be used to judge the correctness of the program. Regularities satisfying a database can be relevant as integrity constraints

and as new knowledge discovered (cf. [97], and higher). Inducing properties of programs or databases thus corresponds to a form of reverse engineering.

Although in principle, one could use the normal setting of ILP to discover high-level regularities in programs or databases, the non-monotonic setting of ILP is more appropriate (cf. [114]). In this setting the given database or program is B and the induced hypothesis H contains the high level regularities one is interested in. One example in a database context was given in Section 3.2. To illustrate the programming context, consider the sorting example once more, and assume that B contains quicksort, sorted and permutation. Systems working in the non monotonic setting, such as CLAUDIEN [113], could then induce a hypothesis containing the clauses E^+ listed in the previous section.

11.2.3. Program testing and debugging The relation between algorithmic debugging and inductive inference is well known since Shapiro's influential work on the MIS [125]. Basically, debugging a program corresponds to credit-assignment problem in inductive inference. Furthermore, once the bug has been located, one may try to repair it by using incremental inductive inference techniques.

Whereas algorithmic debugging starts from a known bug in a program, program testing and verification tries to discover whether there exist bugs in the program. To this aim they generate a test set of example behaviours of the program, which can then be judged on correctness by the user. To generate a test set from a program or knowledge-base, satisfying certain desirable properties, one can employ ILP techniques (cf. [36, 116]). Indeed, suppose one starts from a program P to test and an ILP system (in the example setting). Roughly speaking, the ILP approach to test generation computes a minimal set of examples E of the program's behaviour such that E is sufficient to induce a program P' equivalent to P (or to uniquely distinguish P from a set of alternatives). The underlying assumption states that if P behaves correctly on E, it is correct. The computation of E is done incrementally. Initially, E is empty and P' is a program generated by the ILP system and correct with regard to E. If P and P' are not equivalent, an example e can be generated that is true in one program but not in the other. The example e (with the truthvalue in P) is then added to E and the process of inducing P' from E, and generating examples is repeated until P is equivalent to P'. The example set E is then the required test set.

11.2.4. Knowledge base updating Roughly speaking, the problem of knowledge-base updating (see [29, 19, 44, 45, 138, 96, 52]) can be specified as follows. Given is a deductive database D, satisfying a set of integrity constraints I and a formula f not explained by the database. The aim is then to find an updated database which explains the formula f such that all integrity constraints remain satisfied. To illustrate this, let D be

$$D = \left\{ \begin{array}{l} grandparent(x,y) \leftarrow father(x,z), parent(z,y) \\ parent(x,y) \leftarrow mother(x,y) \\ father(Henry, Jane) \leftarrow \\ mother(Jane, John) \leftarrow \\ mother(Jane, Alice) \leftarrow \end{array} \right.$$

Let the integrity theory I be

$$I = \left\{ \begin{array}{l} \leftarrow father(x,x) \\ \leftarrow mother(x,x) \end{array} \right.$$

An update request f_1 could then ask to make *grandparent(George,Henry)* $\leftarrow$ true. Typical knowledge-base updating methods realise the update requests by adding and deleting ground facts to the database (using a mixture of abduction and techniques to shortcut proofs). For example, the above update request could be realised by adding the fact *father(George,Henry)* $\leftarrow$.

The problem of knowledge-base updating as formulated above corresponds to an *incremental* ILP problem (in the definite setting) where $B \wedge H = D$ and $E^+ = I$, and the update request is considered positive evidence (cf. [110]). The advantage of reformulating knowledge-base updating in terms of ILP is that this allows us to extend the allowed transactions. None of the existing knowledge-base updating methods allow the induction of non-factual clauses; few techniques can delete non-factual clauses from the database. In contrast, in the ILP setting this is very natural. Given the above database, integrity theory and appropriate evidence, incremental ILP techniques could induce the missing clauses for *parent* and *grandparent*. On the other hand, ILP techniques could also benefit from the work on knowledge-base updating, which has spent a lot of effort to cope with normal program clauses in an SLDNF setting. In ILP, few techniques handle negation in a general and sound manner (but see [136]).

11.2.5. Abduction Abduction, as it is currently perceived in Logic Programming [51], can be considered the special case of the example setting in inductive logic programming, where the hypotheses are restricted to sets of ground facts and the evidence to single positive examples[10]. This statement reveals an important difference among the two techniques: in ILP, the facts (examples) are usually assumed to be stable as the clauses are to be learned; in contrast, in abductive logic programming, the clauses are stable and the facts are to be learned. Therefore, these two techniques should not be considered opposite, but rather complementary. Indeed, many ILP systems include an abductive component (e.g. MIS [125], CLINT [107], abduction is also a special case of inverse resolution, etc.). Also, applications of abduction, may be extendable towards inductive logic programming. One such application was discussed above: intensional knowledge-base updating.

12. Conclusion and future directions

Plotkin [100] in the early 1970's and Shapiro [125] in the early 1980's set the scene for the recent upsurge of interest in the area of learning first-order formulae. However, since 1990 ILP has grown from a theoretical backwater to a mainstream area of research, as evidenced by three annual international workshops [82, 85, 86]. Many of the problems encountered on the way can make use of solutions developed in Machine Learning, Statistics and Logic Programming.

[10] It has to be mentioned that abduction has considered more complicated representations for background theories, including normal program clauses.

Many future advances of ILP are likely to come from well-established techniques drawn from Logic Programming. For instance, at present most ILP systems (with the exceptions of MOBAL [55] and the system of [130]) require that all mode and type information concerning predicates in the background knowledge be provided by the user. However, both type and mode declarations could be derived automatically from analysis of the background knowledge. In addition benefit could potentially be derived from making use of work on termination, knowledge-base updating, algorithmic debugging, abduction, constraint logic programming, program synthesis and program analysis.

It should be clear from Section 5 that logical theorem-proving is at the heart of all ILP methods. For this reason it must be worth asking whether the technology of Prolog interpreters is sufficient for all purposes. Reconsider the Tweety example in Section 2.2. Implementing a general system that carried out the inference in this example would require a full-clausal theorem prover. However, most ILP systems merely use a Prolog interpreter to carry out theorem-proving. Is it worth going to more computationally expensive techniques? In learning full-clausal theories, De Raedt and Bruynooghe [113] have made use of Stickel's [132] efficient full-clausal theorem-prover. Stickel's theorem prover compiles full clauses into a set of definite clauses. These definite clauses are then executed by a Prolog interpreter using iterative deepening. This technique maintains most of Prolog's efficiency while allowing full theorem-proving. Learning full-clausal theories is a largely unexplored new area for ILP.

The problem of dealing efficiently and effectively with numerical data is an important challenge to ILP. Earlier systems such as LINUS [66] dealt with the problem by allowing simple inequalities, such as $X > 7$, in the hypothesis language. Recent work on introducing more general linear inequalities into inductively constructed definite clauses [75, 53] provides an elegant logical framework for this problem. This approach also allows the introduction of Constraint Logic Programming (CLP) techniques into ILP.

ILP research has many issues to deal with and many directions to go. By maintaining strong connections between theory, implementations and applications, ILP has the potential to develop into a powerful and widely-used technology.

Acknowledgements

The authors would especially like to thank Maurice Bruynooghe of the Katholieke Universiteit Leuven and Lincoln Wallen of Oxford University Computing Laboratory for useful input during this research. They would also like to thank the reviewers for their constructive comments and suggestions. This work was supported partly by the Esprit Basic Research Action ILP (project 6020), an SERC Advanced Research Fellowship held by Stephen Muggleton, and the Belgian National Fund for Scientific Research. Stephen Muggleton is a Research Fellow of Wolfson College Oxford and Luc De Raedt is a post-doctoral researcher of the Belgian National Fund for Scientific Research.

Appendix A: Notational conventions

□ *: false;*

$\blacksquare$: *true;*
$\models$: *logical entailment;*
$\wedge$: *conjunction;*
$\leftarrow$: *implication;*
$\leftrightarrow$: *double implication;*
$\subset$: *proper subset;*
$\subseteq$: *subset;*
$\top$: *maximally general element;*
$\bot$: *maximally specific element;*
$\overline{f}$: *complement of* f*;*
$d(t)$: *depth of term* t*;*
glb : *greatest lower bound;*
$l(t)$: *level of term* t *in clause;*
lub : *least upper bound;*
$p(X)$: *prior probability of* X*;*
$p(X|Y)$: *prior probability of* X *given* Y*;*
$I(X)$: *prior information content of* X*;*
$I(X|Y)$: *information content of* X *given* Y*;*
ρ : *refinement operator;*
ρ^* : *transitive closure of* ρ*;*
$\mathcal{B}(T)$: *the Herbrand base of* T*, where* T *is a conjunction of clauses;*
$\mathcal{H}$: *set of well-formed hypotheses, contains set of sets of clauses in* $\mathcal{L}$*];*
$\mathcal{L}$: *language bias, i.e. set of clauses;*
$\mathcal{M}^+(T)$: *the minimal Herbrand model of* T*, where* T *is a definite clause program;*
$\mathcal{M}^-(T)$ $= \{\overline{f} : f \in (\mathcal{B}(T) - M^+(T))\}$*, i.e. the complement of the minimal Herbrand model of* T*, where* T *is a definite clause program;*
$\mathcal{M}_h^+(T)$: *the finite success-set (a subset of* $\mathcal{M}^+(T)$*) which can be derived with proofs of at most depth* h*;*
$\mathcal{M}_h^-(T)$: *the finite failure-set (a subset of* $\mathcal{M}^-(T)$*) which can be derived with failed proofs of at most depth* h*,*
$\mathcal{P}(F)$: *set of predicate symbols occurring in* F*, where* F *is any logical formula;*
$\mathcal{Q}^+(T)$: *the set of clauses true in* $\mathcal{M}^+(T)$ *and using the same alphabet as* T*;*
$\mathcal{Q}^-(T)$: *the set of clauses false in* $\mathcal{M}^+(T)$ *and using the same alphabet as* T*;*
$\mathcal{RL}^n(T)$: *nth linear resolution of definite clause theory* T*;*