Learnability in Inductive Logic Programming: Some Basic Results and Techniques

Michael Frazier* and **C. David Page Jr.**†
Department of Computer Science and Beckman Institute
University of Illinois
Urbana, IL 61801

Abstract

Inductive logic programming is a rapidly growing area of research that centers on the development of inductive learning algorithms for first-order definite clause theories. An obvious framework for inductive logic programming research is the study of the *pac-learnability* of various restricted classes of these theories. Of particular interest are theories that include *recursive* definite clauses. Because little work has been done within this framework, the need for initial results and techniques is great. This paper presents results about the pac-learnability of several classes of simple definite clause theories that are allowed to include a recursive clause. In so doing, the paper uses techniques that may be useful in studying the learnability of more complex classes.

1. Introduction

Inductive logic programming is a rapidly-growing area of research at the intersection of machine learning and logic programming [Muggleton, 1992]. It focuses on the design of algorithms that learn (first-order) definite clause theories from examples. A natural framework for research in inductive logic programming is the investigation of the learnability/predictability of various classes of definite clause theories, particularly in the models of *pac-learnability* [Valiant, 1984] and *learning by equivalence queries* [Angluin, 1988]. Surprisingly little work has been done within this framework, though interest is rising sharply [Džeroski et al., 1992; Cohen and Hirsh, 1992; Ling, 1992; Muggleton, 1992; Page and Frisch, 1992; Arimura et al., 1992].

This paper describes new results on the learnability of several restricted classes of simple (two-clause) definite clause theories that may contain *recursive* clauses; theories with recursive clauses appear to be the most difficult to learn. The positive results are proven for learning by equivalence queries, which implies pac-learnability [Angluin, 1988]. In obtaining the results, we introduce techniques that may be useful in studying the learnability of other classes of definite clause theories with recursion.

The results are presented with the following organization. Section 2 describes the learning model. Section 3 shows that the class $\mathcal{H}_1$, whose concepts are built from unary predicates, constants, variables, and unary functions, is learnable. Section 4 shows that the class $\mathcal{H}_*$, an extension of $\mathcal{H}_1$ that allows predicates of arbitrary arity, is not learnable under a reasonable complexity-theoretic assumption.[1] Nevertheless, Section 4 also shows that each subclass $\mathcal{H}_k$ of $\mathcal{H}_*$, in which predicates are restricted to arity k, is learnable *in terms of* a slightly more general class, and is therefore *pac-predictable*. The prediction algorithm is a generalization of the learning algorithm in Section 3. The results of Section 4 leave open the questions of whether (1) $\mathcal{H}_*$ is *pac-predictable* and (2) $\mathcal{H}_k$ is *pac-learnable*. Section 5 relates our results to other work on the learnability of definite clause theories in the pac-learning or equivalence query models [Džeroski et al., 1992; Page and Frisch, 1992].

2. The Model

The examples provided to algorithms that learn concepts expressed in *propositional logic* traditionally have been truth assignments, or models. Such an example is positive if and only if it satisfies the concept. But concepts in first-order logic may (and almost always do) have infinite models. Therefore, algorithms that learn definite clause theories typically take logical formulas, usually ground atomic formulas, as examples instead. Such an example is positive if and only if it is a logical consequence of the concept. The algorithms in this paper use ground atomic formulas (atoms) as examples in this manner. A concept is used to classify ground atoms according to the atoms'

*mfrazier@cs.uiuc.edu. Supported in part by NSF Grant IRI-9014840, and by NASA Grant NAG 1-613.

†dpage@cs.uiuc.edu. Supported in part by NASA Grant NAG 1-613.

[1] The classes $\mathcal{H}_1$ and $\mathcal{H}_k$ (discussed next) have one other restriction, that variables are *stationary*. This restriction is defined later.

truth value in the concept's least Herbrand model,[2] which is to say, according to whether the atoms logically follow from the concept. For example, the concept $\forall x[p(f(g(x)))] \wedge \forall x[p(f(x)) \rightarrow p(f(h(x)))]$, which is in $\mathcal{H}_1$, classifies $p(f(g(c)))$ and $p(f(h(h(h(g(c))))))$ as true or *positive* while it classifies $p(f(c))$ as false or *negative*. If A and B are two concepts that have the same least Herbrand model, we say they are *equivalent*, and we write $A \cong B$.

In a learning problem, a concept C, called the *target*, is chosen from some class of concepts $\mathcal{C}$ and is hidden from the learner. Each concept classifies each possible example element x from a set X, called the *instance space*, as either *positive* or *negative*. We require the existence of an algorithm that, for every $C \in \mathcal{C}$ and every $x \in X$, efficiently (polynomial-time) determines whether C classifies x as positive. (Such an algorithm exists for each concept class introduced in this paper.) The learner infers some concept C' based on information about how the target C classifies the elements of the instance space X. For each of our learning problems, the concept class $\mathcal{C}$ is a class of definite clause theories, and we require that any learning algorithm, $\mathcal{A}$, must for any $C \in \mathcal{C}$ produce a concept $C' \in \mathcal{C}$ such that $C' \cong C$, that is, that C and C' have the same least Herbrand model. (For *predictability* we remove the requirement that C' belong to $\mathcal{C}$, though we still must be able to efficiently determine how C' classifies examples.) The instance space X is the Herbrand universe of C, and the learning algorithm $\mathcal{A}$ is able to obtain information about the way C classifies elements of X only by asking *equivalence queries*, in which $\mathcal{A}$ conjectures some C' and is told by an oracle whether $C' \cong C$. If $C' \ncong C$, $\mathcal{A}$ is provided a *counterexample* x that C' and C classify differently.

We close this section by observing that the union of several classes can be learned by interleaving the learning algorithms for each class.

Fact 1 *Let $p(n)$ be a polynomial in n, and let $\{\mathcal{C}_i : 1 \leq i \leq p(n)\}$ be concept classes with learning algorithms $\{\mathcal{A}_i : 1 \leq i \leq p(n)\}$ having time complexities $\{T_{\mathcal{A}_i} : 1 \leq i \leq p(n)\}$ respectively. Then the concept class $\cup_{i=1}^{p(n)} \mathcal{C}_i$ can be learned in time $max_{1 \leq i \leq p(n)}\{p(n)T_{\mathcal{A}_i}\}$.*

3. The Class $\mathcal{H}_1$

The concept class $\mathcal{H}_1$ is the class of concepts that can be expressed as a conjunction of at most two simple clauses, where a simple clause is a positive literal (an atom) composed of unary predicates and unary or 0-ary functions or an implication between two such positive literals.

As an example, the following is a concept in $\mathcal{H}_1$ that we have seen already.

$$\forall x[p(f(g(x)))] \wedge \forall x[p(f(x)) \rightarrow p(f(h(x)))]$$

Since our conjuncts are always universally quantified, we henceforth leave the quantification implicit. Thus the above concept is written

$$[p(f(g(x)))] \wedge [p(f(x)) \rightarrow p(f(h(x)))]$$

We can divide $\mathcal{H}_1$ into two classes: trivial concepts, which are equivalent ($\cong$) to conjunctions of at most two atoms, and non-trivial, or recursive, concepts. The trivial concepts of $\mathcal{H}_1$ can be learned easily.[3] We next describe an algorithm that learns the non-trivial, or recursive, concepts in $\mathcal{H}_1$. It follows that $\mathcal{H}_1$ is learnable, since we can interleave this algorithm with the one that learns the trivial concepts of $\mathcal{H}_1$.

It can be shown that the recursive concepts in $\mathcal{H}_1$ have the form

$$[p(t_1)] \wedge [p(t_2(x)) \rightarrow p(t_3(x))] \qquad (1)$$

where t_1 is a term, and $t_2(x)$ and $t_3(x)$ are terms ending in the same variable, x.[4] The fact that the functions and predicates are unary leads to a very concise description of a recursive concept in $\mathcal{H}_1$. Specifically, we can drop all parentheses in and around terms. Further, since we are discussing recursive concepts, all predicate symbols are the same and can likewise be dropped. Thus any concept having the form of concept (1) may be written $[\alpha e] \wedge [\beta x \rightarrow \gamma x]$, or

$$\begin{cases} \alpha e \\ \beta x \rightarrow \gamma x \end{cases} \qquad (2)$$

where α, β, and γ are strings of function symbols, x is a variable, and e is either a constant or a variable. Using this notation, determining whether, for example, αe unifies with βx requires only determining whether either α is a prefix of β or β is a prefix of α. For any strings α and β, if α is a prefix of β then we write $\alpha < \beta$.

Since we are speaking now of recursive concepts only, we refer to the two parts of the concept as the *base atom* and the *recursive clause*. The atoms in the least Herbrand model that are not instances of the base

[2] The least Herbrand model of a set of definite clauses (every such set has one) is sometimes referred to as *the model* or the *unique minimal model* of the set. The set of definite clauses entails a logical sentence if and only if the sentence is true in this model. It is often useful to think of this model as a set, namely, the set of ground atoms that it makes true.

[3] The basic idea is that the learning algorithm, by using equivalence queries, is able to obtain one example for each of the (at most two) atoms in the concept. Only a few atoms are more general than (that is, have as instances) each example, and the algorithm conjectures all combinations of these atoms, one for each of the (at most two) examples.

[4] The proof of this is omitted for brevity, as are some other proofs; see [Frazier and Page, 1993; Frazier, 1993] for missing proofs.

atom in the concept are generated by applying the recursive clause. A concept is equivalent to a conjunction of two atoms if its recursive clause can be applied at most once. For the recursive clause to apply at all, α must unify with β, and for it to apply more than once, β must unify with γ. Hence a concept is non-trivial only if (1) either $\alpha<\beta$ or $\beta<\alpha$ and (2) either $\beta<\gamma$ or $\gamma<\beta$. In light of Fact 1, to show that the non-trivial concepts can be learned in polynomial time, we need only show that the class of non-trivial concepts can be partitioned into a polynomial number of concept classes, each of which can be learned in polynomial time. Therefore, we carve the class of non-trivial concepts into five different subclasses defined by the prefix relationships among α, β, and γ. The five possible sets of prefix relationships that can yield recursive concepts, based on our earlier discussion, are (1) $\alpha<\beta$ and $\beta<\gamma$, (2) $\alpha<\beta$, $\gamma<\beta$, and $\alpha<\gamma$, (3) $\alpha<\beta$, $\gamma<\beta$, and $\gamma<\alpha$, (4) $\beta<\alpha$ and $\beta<\gamma$, (5) $\beta<\alpha$ and $\gamma<\beta$. (We do not need to divide case (4) into two cases because the relationship between α and γ is insignificant here.) The approach for each subclass is similar—generalize the first positive example in such a way that the oracle is forced to provide a positive example containing whatever pieces of α, β, or γ are missing from the first example. For brevity, we present only the proof of the most difficult case.

$\beta<\alpha$ and $\gamma<\beta$. This class consists of concepts having the form

$$\begin{cases} \phi\psi\omega e \\ \phi\psi x \rightarrow \phi x \end{cases} \tag{3}$$

Concepts of this form generate smaller and smaller atoms by deleting copies of ψ at the front of ω; if ψ is not a prefix of ω, the concept can delete only one copy of ψ. Any concept of this form has the least Herbrand model described by

$$\phi\psi^k\zeta e \quad \text{for } 1 \leq k \leq n+1, \text{ where } \omega = \psi^n\zeta$$

Lemma 2 *This class can be learned in polynomial time.*

Proof: To learn this class an algorithm needs to obtain an example that contains ϕ, ψ, ω, and ζ. It then must determine n. We give an algorithm that makes at most two equivalence queries to obtain ϕ, ψ, ω, and ζ. It then guesses larger and larger values for n until it guesses the correct value. This value of n is linearly related to the length of the base atom, so overall the algorithm takes polynomial time.

1. Conjecture the false concept to obtain counterexample ρe
2. Dovetail the following algorithms
   - Assuming $\rho = \phi\psi^j\zeta\xi e$ for some $j \geq 1$
   (a) Select ϕ, ψ, ζ, ξ from ρ
   (b) Guess the value of n
   (c) Halt with output
   $$\begin{cases} \phi\psi^{n+1}\zeta e \\ \phi\psi x \rightarrow \phi x \end{cases}$$
   - Assuming $\rho = \phi\zeta\xi e$
   (a) Select ϕ, ζ, ξ from ρ
   (b) Conjecture
   $$\{ \ \phi\zeta e$$
   to obtain counterexample $\rho' e'$. Note that ρ' necessarily contains ψ as a substring.
   (c) Select ψ from ρ'
   (d) Guess n
   (e) Halt with output
   $$\begin{cases} \phi\psi^{n+1}\zeta e \\ \phi\psi x \rightarrow \phi x \end{cases}$$

When the algorithm *selects* substrings from a counterexample, it is in reality dovetailing all possible choices; nevertheless, we observe that there are only $O(|\rho|^5)$ (respectively $O(|\rho'|)$) choices to try. Similarly, when the algorithm guesses the value of n, it is actually making successively larger and larger guesses for n and testing whether it is correct with an equivalence query. It will obtain the correct value for n in time polynomial in the size of the target concept. At that point it outputs the necessarily correct concept and halts. □

It is worth noting that the algorithm above uses only two of the counterexamples it receives, though it typically makes more than two equivalence queries. This is the case with the algorithms for the other subclasses as well. It is also worth noting that when the algorithm above guesses the value of n, it is guessing the number of times the recursive clause is applied to generate the earliest generated atom of which either example is an instance.

By the preceding arguments, we have Theorem 3.

Theorem 3 *The concept class $\mathcal{H}_1$ is learnable.*

4. Increasing Predicate Arity

It is often useful to have predicates of higher arity, but otherwise maintain the form of the concepts in $\mathcal{H}_1$. For example

$$\begin{cases} plus(x, 0, x) \\ plus(x, y, z) \rightarrow plus(x, s(y), s(z)) \end{cases}$$

$$\begin{cases} greater(s(x), 0) \\ greater(x, y) \rightarrow greater(s(x), s(y)) \end{cases}$$

In this section we remove the requirement that predicates be unary. Specifically, let $\mathcal{H}_*$ be the result of allowing predicates of arbitrary arity but requiring functions to remain unary, with the additional restriction, which we define next, that variables be *stationary*. Notice that because functions are unary, each

argument has at most one variable (it may have a constant instead), and that variable must be the last symbol of the argument. A concept meets the *stationary variables* restriction if for any variable x, if x appears in argument i of the consequent of the recursive clause then x also appears in argument i of the antecedent. This class does include the preceding arithmetic concepts built with the successor function and the constant 0, but does not include the concept $[p(a,b,c)] \wedge [p(x,y,z) \rightarrow p(z,x,y)]$ because variables "shift positions" in the recursive clause. Unfortunately, we have the following result for the class $\mathcal{H}_*$.

Theorem 4 *$\mathcal{H}_*$ is not learnable, assuming $R \neq NP$.*[5]

The result follows from a proof that the *consistency problem* [Pitt and Valiant, 1988] for $\mathcal{H}_*$ is NP-hard. Our conjecture is that the class is not even predictable (that is, learnable *in terms of* any other class), though this is an open question.

Nevertheless, we now show that if we fix the predicate arity to any integer k, then the resulting concept class $\mathcal{H}_k$ is learnable in terms of a slightly more general class, called $\mathcal{H}_k{}'$, and is therefore predictable (the question of learnability of $\mathcal{H}_k$ in terms of $\mathcal{H}_k$ itself remains open). Concepts in $\mathcal{H}_k{}'$ may be any union of a concept in $\mathcal{H}_k$ and two additional atoms built from variables, constants, unary functions, and predicates with arity at most k. An example is classified as positive by such a concept if and only if it is classified as positive by the concept in $\mathcal{H}_k$ or is an instance of one of the additional atoms. The learning algorithm is based on the learning algorithm for $\mathcal{H}_1$, and central to it are the following definition and lemma. In the lemma, and afterward, we use G_0 to denote the base atom and, inductively, G_{i+1} to denote the result of applying the recursive clause to G_i.

Definition 5 *Let*

$$\begin{cases} p(\alpha_1 e_1, ..., \alpha_k e_k) \\ p(\beta_1 e_1', ..., \beta_k e_k') \rightarrow p(\gamma_1 e_1'', ..., \gamma_k e_k'') \end{cases}$$

be a concept in $\mathcal{H}_k$. Then we say the *subconcept at argument i*, for $1 \leq i \leq k$, of this concept is

$$\begin{cases} \alpha_i e_i \\ \beta_i e_i' \rightarrow \gamma_i e_i'' \end{cases}$$

For example the subconcepts at arguments 1 through 3, respectively, of the concept *plus* are:

$$\begin{cases} x \\ x \rightarrow x \end{cases} \qquad \begin{cases} 0 \\ y \rightarrow s(y) \end{cases} \qquad \begin{cases} x \\ z \rightarrow s(z) \end{cases}$$

Notice that an atom $G_i = plus(t_1, t_2, t_3)$ is the ith atom generated by the concept *plus* if and only if for all $1 \leq k \leq 3$, the ith term generated by subconcept k of *plus* is t_k. This is the case because the argument positions in the definition of *plus* never disagree on the binding of a variable. Not all concepts in $\mathcal{H}_k$ have this property; concepts that do are said to be *decoupled*.

Lemma 6 (Decoupling Lemma) *Let*

$$\begin{cases} p(\alpha_1 e_1, ..., \alpha_k e_k) \\ p(\beta_1 e_1', ..., \beta_k e_k') \rightarrow p(\gamma_1 e_1'', ..., \gamma_k e_k'') \end{cases}$$

be a concept in $\mathcal{H}_k$. For any $1 \leq j \leq k$, if e_j' is a variable x then for any $n \geq 2$: if $G_n = p(t_1, ..., t_k)$ unifies with $p(\beta_1 e_1', ..., \beta_k e_k')$, the binding generated for x by this unification is the same as the binding generated for x by unifying t_j with $\beta_j x$. Equivalently, for any $1 \leq i \neq j \leq k$, if e_i' and e_j' are the same variable, x, then for any $n \geq 2$: unifying t_i with $\beta_i x$ yields the same binding for x as does unifying t_j with $\beta_j x$.

The lemma says, in effect, that every concept in $\mathcal{H}_k$ behaves in a decoupled manner after the generation of G_2. Therefore, by the Decoupling Lemma, any concept C in $\mathcal{H}_k$ that generates $G_0, G_1, ...$ can be rewritten as the union of three concepts (only polynomially larger than C), two of which are the atoms G_0 and G_1. The third concept is some $\tilde{C}$ in $\mathcal{H}_k$ whose base atom $\tilde{G}_0$ is G_2 and whose recursive clause (if any) generates $\tilde{G}_1 = G_3, \ldots, \tilde{G}_m = G_{m+2}, \ldots$ meeting the following condition: for all $m \geq 0$ and any variable x in the antecedent of the recursive clause, no two argument positions impose different bindings on x when generating $\tilde{G}_{m+1}$ from $\tilde{G}_m$. In other words, $\tilde{C}$ is decoupled, so the behavior of $\tilde{C}$ can be understood as a simple composition of the independent behaviors of the subconcepts at arguments 1 through k. As an example, the concept *plus* can be rewritten as the union of $G_0 = plus(x, 0, x)$, $G_1 = plus(x, s(0), s(x))$, and $\tilde{C} =$

$$\begin{cases} plus(x, s(s(0)), s(s(x))) \\ plus(x, y, z) \rightarrow plus(x, s(y), s(z)) \end{cases}$$

Of course, the definition of *plus* is such that even without this rewriting, the concept is decoupled. Consider instead the concept

$$\begin{cases} p(z, s(w), z) \\ p(x, x, y) \rightarrow p(x, s(s(s(0))), s(y)) \end{cases}$$

In generating G_1 from G_0, the first argument binds x to z while the second binds x to $s(w)$, and the third binds y to z. Thus the concept is not decoupled. The result is that x and y are bound to $s(w)$, so G_1 is $p(s(w), s(s(s(0))), s(s(w)))$. Furthermore, in generating G_2 from G_1, the first argument binds x to $s(w)$, while the second binds x to $s(s(s(0)))$, and the third binds y to $s(s(w))$. The result is that x is bound to $s(s(s(0)))$, and y is bound to $s(s(s(s(0))))$. (But from this point on, the first and second arguments always agree on the binding for x.) We would like the concept, instead, to be such that the bindings

[5] That is, $\mathcal{H}_*$ is not pac-learnable, and therefore is not learnable in polynomial time by equivalence queries. R is the class of problems that can be solved in random polynomial time.

generated by the arguments are *independent*, that is, the concept is decoupled. The following, equivalent concept in $\mathcal{H}_k{}'$ has this property: $G_0 = p(z, s(w), z)$, $G_1 = p(s(w), s(s(s(0))), s(s(w))$, and $\tilde{C} =$

$$\begin{cases} p(s(s(s(0))), s(s(s(0))), s(s(s(s(s(0))))) \\ p(x, x, y) \rightarrow p(x, x, s(y)) \end{cases}$$

These observations motivate an algorithm that learns $\mathcal{H}_k$ in terms of $\mathcal{H}_k{}'$ and is therefore a prediction algorithm for $\mathcal{H}_k$.

Theorem 7 *For any constant k, the class $\mathcal{H}_k$ is predictable from equivalence queries alone.*

Proof: (Sketch) Any target concept C is equivalent to some concept in $\mathcal{H}_k{}'$ that consists of G_0, G_1 and $\tilde{C} =$

$$\begin{cases} p(\alpha_1 e_1, ..., \alpha_k e_k) \\ p(\beta_1 e_1', ..., \beta_k e_k') \rightarrow p(\gamma_1 e_1'', ..., \gamma_k e_k'') \end{cases}$$

that is only polynomially larger than C, where $\tilde{C}$ is decoupled. Our algorithm will obtain such an equivalent concept. $\tilde{C}$ generates some sequence of atoms $\tilde{G}_0, \tilde{G}_1, ...$ (this sequence may or may not be finite). Notice that the subconcepts, S_1, ..., S_k, of $\tilde{C}$ are, respectively:

$$\begin{cases} \alpha_1 e_1 \\ \beta_1 e_1' \rightarrow \gamma_1 e_1'' \end{cases} \quad \cdots \quad \begin{cases} \alpha_k e_k \\ \beta_k e_k' \rightarrow \gamma_k e_k'' \end{cases}$$

Let $g_{i,j}$ be the jth term generated by subconcept i. Then because $\tilde{C}$ is decoupled, for all $j \geq 0$ we have $\tilde{G}_j = p(g_{1,j}, ..., g_{k,j})$.

At the highest level of description, the prediction algorithm poses equivalence queries in such a way that it obtains, as examples, instances of G_0, G_1, and $\tilde{G}_i$ and $\tilde{G}_j$ for distinct i and j. The algorithm determines G_0, G_1, $\tilde{G}_i$, and $\tilde{G}_j$ from their examples, and it determines $\tilde{C}$ from $\tilde{G}_i$ and $\tilde{G}_j$. To determine $\tilde{C}$ the algorithm uses the learning algorithm for $\mathcal{H}_1$ to learn the subconcepts $S_1, ..., S_k$ of $\tilde{C}$. The only subtlety is that some subconcept S_i may be equivalent to a conjunction of two terms, and the learning algorithm for $\mathcal{H}_1$ might return such a concept. Such a concept cannot serve as a subconcept for a member of $\mathcal{H}_k$. But it is straightforward to verify that the only case in which this can occur is the case in which all $g_{i,j}$, $j \geq 1$, are identical. And the examples the learning algorithm receives do not include $g_{i,0}$ (it receives only examples extracted from G_2, G_3, ...). Therefore, if the concept returned by the learning algorithm is a conjunction of at most two atoms, it is in fact a single atom—$g_{i,1}$ ($= g_{i,j}$, for all $j \geq 1$)—in which case we may use the concept in $\mathcal{H}_1$ whose base atom, recursive clause antecedent, and recursive clause consequent are all $g_{i,1}$. We now fill in the details of the algorithm.

The algorithm begins by conjecturing the empty theory, and it necessarily receives a positive counterexample in response. This counterexample is an instance of some more general atom, A_1, that is either G_0, G_1, or some $\tilde{G}_i$. The algorithm guesses A_1 and guesses whether A_1 is G_0, G_1, or some $\tilde{G}_i$. It then conjectures A_1 in an equivalence query and, if A_1 is not the target, necessarily receives another positive example. (As earlier, by *guess* we mean that the algorithm dovetails the possible choices.) This example is also an instance of G_0, G_1, or some $\tilde{G}_i$, but it is *not* an instance of A_1. Again, the algorithm guesses that atom—call it A_2—and guesses whether it is an instance of G_0, G_1, or some $\tilde{G}_i$. Following the second example, and following each new example thereafter, the algorithm has at least two of of G_0, G_1, $\tilde{G}_i$, and $\tilde{G}_j$ (some $i \neq j$). It conjectures the union of those that it has, with the following exception: if it has both $\tilde{G}_i$, and $\tilde{G}_j$ (any $i \neq j$), it uses a guess of $\tilde{C}$ in place of $\tilde{G}_i$ and $\tilde{G}_j$. Again, in response to such a conjecture, either the algorithm is correct or it receives a positive counterexample. It remains only to show how (1) the atoms G_0, G_1, $\tilde{G}_i$, and $\tilde{G}_j$ are "efficiently guessed", and (2) $\tilde{C}$ is "efficiently guessed" from $\tilde{G}_i$ and $\tilde{G}_j$.

Given *any* example that is an instance of G_0, the number of possibilities for G_0 is small (because no function has arity greater than 1). For example, if A is $plus(s(s(s(0))), s(0), s(s(s(s(0)))))$, then the first argument of G_0 must be $s(s(s(0)))$, $s(s(s(x)))$, $s(s(x))$, $s(x)$, or x. Similarly, the other arguments of G_0 must be generalizations, or prefixes, of the other arguments of A. Finally, there are at most 5 possible choices of variable co-references, or patterns of variable names, for the three arguments (all three arguments end in the same variable, all three end in different variables, etc.). By this reasoning, there are fewer than $O((2|A|)^k)$ generalizations of A to consider, all of which can be tried in parallel. Note that, because k is fixed, the number of possible generalizations is polynomial in the size of the example. G_1, $\tilde{G}_i$, and $\tilde{G}_j$ are efficiently guessed in the same way.

To learn $\tilde{C}$, the algorithm first determines the "high-level structure" of $\tilde{C}$; specifically, it guesses which of $e_1, ..., e_k, e_1', ..., e_k', e_1'', ..., e_k''$ are variables, and which of these are the *same* variable, e.g. that e_1'' and e_2'' are the same variable x. (In other words, it guesses variable co-references; since $\tilde{C}$ is decoupled, these are only important for ensuring proper co-references in generated atoms that contain variables.) There are fewer than k^{3k} possibilities, where k is fixed. The algorithm then is left with the task of precisely determining the subconcepts S_1, ..., S_k of $\tilde{C}$. Because it has two examples of distinct atoms generated by $\tilde{C}$—$\tilde{G}_i$ and $\tilde{G}_j$—it has two examples of each subconcept. For example, where $\tilde{G}_j$ is $p(g_{1,j}, ..., g_{k,j})$, for each $1 \leq i \leq k$ we know $g_{i,j}$ is an example of subconcept S_i. The algorithm uses the learning algorithm for $\mathcal{H}_1$ to learn the subconcepts from these examples, with the following slight modification. While the learning algorithm

for $\mathcal{H}_1$ would ordinarily conjecture a concept in $\mathcal{H}_1$, the present learning algorithm must conjecture a concept in $\mathcal{H}_k'$. Therefore, the algorithm conjectures every concept in $\mathcal{H}_k'$ that results from any combination of conjectures, by the learning algorithm for $\mathcal{H}_1$, for the subconcepts; that is, it tries all combinations of subconcepts. Because k is fixed, the number of such combinations is polynomial in the sizes of the counterexamples seen thus far.

Finally, recall that for concepts having the form of concept 3 the learning algorithm for $\mathcal{H}_1$ must guess the value for n, where n is the number of times the recursive clause was applied to generate the earlier of the two examples it has. Therefore, the present learning algorithm may have to guess n. (The n sought is necessarily the same for all subconcepts. Thus only one n needs to be found.) This is handled by initially guessing $n = 1$ and guessing successively higher values of n until the correct n is reached. This approach succeeds provided that the target truly contains a type 3 subconcept. In the case that the target does not contain a type 3 subconcept, the potentially non-terminating, errant search for a non-existent n halts because we are interleaving the steps from *all* pending guesses of *all* forms—including the correct (not type 3) form—of the subconcept. □

5. Related Work

The concept classes studied in this paper are incomparable to—that is, neither subsume nor are subsumed by—the other classes of definite clause theories whose learnability, in the *pac-learning* and/or *equivalence query* models, has been investigated [Džeroski et al., 1992; Page and Frisch, 1992].[6] Page and Frisch investigated classes of definite clauses that may have predicates and functions of arbitrary arity but explicitly do *not* have recursion [Page and Frisch, 1992]. In that work, a *background* theory was also allowed; allowing such a theory in the present work is an interesting topic for future research. Džeroski, Muggleton, and Russell [Džeroski et al., 1992] investigated the learnability of classes of function-free *determinate* k-clause definite clause theories under simple distributions, also in the presence of a background theory.[7] This class includes recursive concepts; to learn recursive concepts, the algorithm requires two additional kinds of queries (*existential queries* and *membership queries*). Rewriting definite clause theories that contain functions to function-free clauses allows their algorithm to learn in the presence of functions. Nevertheless, the restriction that clauses be *determinate* effectively limits the depth of function nesting; their algorithm takes time exponential in this depth. So, for example, while the algorithm can easily learn the concept *even integer*, or *multiple of 2*, from $\mathcal{H}_1$ — $[even(0)] \wedge [even(x) \rightarrow even(s(s(x)))]$—the time it requires grows exponentially in moving to a concept such as *multiple of 10* or *multiple of 1000*, also in $\mathcal{H}_1$. It is easy to show that the classes $\mathcal{H}_1$, $\mathcal{H}_k$, and $\mathcal{H}_*$, rewritten to be function-free, are not determinate.

[6] Theoretical work on the learnability of definite clauses in other models includes Shapiro's [Shapiro, 1983] work on learning in the limit, Angluin, Frazier, and Pitt's [Angluin et al., 1992] work with propositional definite clause theories, and Ling's [Ling, 1992] investigation of learning from *good examples*.

[7] The restriction to simple distributions is a small one that quite possibly can be removed.

References

Angluin, D. (1988). Queries and concept learning. *Machine Learning*, 2:319–342.

Angluin, D., Frazier, M., and Pitt, L. (1992). Learning conjunctions of horn clauses. *Machine Learning*, 9:147–164.

Arimura, H., Ishizaka, H., and Shinohara, T. (1992). Polynomial time inference of a subclass of context-free transformations. In *COLT-92*, pages 136–143, New York. The Association for Computing Machinery.

Cohen, W. W. and Hirsh, H. (1992). Learnability of description logics. In *COLT-92*, pages 116–127, New York. The Association for Computing Machinery.

Džeroski, S., Muggleton, S., and Russell, S. (1992). Pac-learnability of logic programs. In *COLT-92*, pages 128–135, New York. The Association for Computing Machinery.

Frazier, M. (1993). *Forthcoming*. PhD thesis, University of Illinois at Urbana-Champaign.

Frazier, M. and Page, C. D. (1993). Learnability of recursive, non-determinate theories: Some basic results and techniques. Submitted to the Third International Workshop on Inductive Logic Programming (ILP93).

Ling, C. X. (1992). Logic program synthesis from good examples. In Muggleton, S. H., editor, *Inductive Logic Programming*, pages 113–129. Academic Press, London.

Muggleton, S. H. (1992). Inductive logic programming. In Muggleton, S. H., editor, *Inductive Logic Programming*, pages 3–27. Academic Press, London.

Page, C. D. and Frisch, A. M. (1992). Generalization and learnability: A study of constrained atoms. In Muggleton, S. H., editor, *Inductive Logic Programming*, pages 29–61. Academic Press, London.

Pitt, L. and Valiant, L. G. (1988). Computational limitations on learning from examples. *Journal of the ACM*, 35(4):965–984.

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

Valiant, L. G. (1984). A theory of the learnable. *Communications of the ACM*, 27(11):1134–1142.