ELSEVIER

Theoretical Computer Science 185 (1997) 191–213

Theoretical Computer Science

A framework for incremental learning of logic programs[1]

M.R.K. Krishna Rao[a, b, *]

[a] *Max-Planck-Institut für Informatik, Im Stadtwald, 66123 Saarbrücken, Germany*

[b] *Computer Science Group, Tata Institute of Fundamental Research, Colaba, Bombay 400 005, India*

Abstract

In this paper, a framework for incremental learning is proposed. The predicates already learned are used as background knowledge in learning new predicates in this framework. The programs learned in this way have nice modular structure with conceptually separate components. This modularity gives the advantages of portability, reliability and efficient compilation and execution.

Starting with a simple idea of Miyano et al. [21, 22] for identifying classes of programs which satisfy the condition that *all the terms occurring SLD-derivations starting with a query are no bigger than the terms in the initial query*, we identify a reasonably big class of polynomial-time learnable logic programs. These programs can be learned from a given sequence of examples and a logic program defining the already known predicates. Our class properly contains the class of innermost simple programs of [32] and the class of hereditary programs of [21, 22]. Standard programs for `gcd`, `multiplication`, `quick-sort`, `reverse` and `merge` are a few examples of programs that can be handled by our results but not by the earlier results of [21, 22, 32].

1. Introduction

Starting with the seminal work of Shapiro [27, 28], the problem of learning logic programs from examples has attracted a lot of attention in the last ten years (see, e.g. [26, 21, 22, 2, 7, 14, 32]). Many techniques and systems for learning logic programs have been developed and used in many applications [23]. In this paper, we identify a class of polynomial-time learnable logic programs.

Our main emphasis is on incremental (step-by-step) learning of logic programs. The predicates already known (learned) are used as background knowledge in learning new predicates. For example, in learning a program for `reverse`, one can use knowledge about the `append` program already known. Similarly, knowledge about `addition` can

* Current address: School of Computing & Information Technology, Faculty of Science & Technology, Griffith University, Nathan, Brisbane, Australia-4111. E-mail: krishna@cit.gu.edu.au.

[1] This is a revised and extended version of [16].

PII S0304-3975(97)00021-2

be used in learning a program for multiplication. This way, one can learn (or synthesize) programs in an incremental fashion. With the following background knowledge

app([ ], Ys) = Ys

app([X|Xs], Ys) = [X|app(Xs, Ys)]

about append, our algorithm comes up with the following program for reverse.

reverse([ ], [ ]) ←

reverse([X|Xs], app(Ys, [X])) ← reverse(Xs, Ys)

The above reverse program together with the equality theory about append constitutes a functional logic program (cf. [11]) and can be transformed into the following logic program through the flattening operation of [4].

reverse([ ], [ ]) ←

reverse([X|Xs], Z) ← reverse(Xs, Ys), append(Ys, [X], Z)

append([ ], Y, Y) ←

append([X|Xs], Y, [X|Zs]) ← append(Xs, Y, Zs)

The process of learning logic programs in our framework can be summarized as follows.

(1) We are given a set of examples (both positive and negative) of the target predicate to be learned along with a logic program defining the already known predicates. For instance, the following program for addition

addition(0, Y, Y) ←

addition(s(X), Y, s(Z)) ← addition(X, Y, Z)

and the examples for mult

⟨mult(0, s(0), 0), true⟩,

⟨mult(0, s(s(0)), 0), true⟩,

⟨mult(*s*(0), s(s(0)), s(s(0))), true⟩,

⟨mult(s(s(0)), s(s(0)), s(s(s(s(0))))), true⟩ and

⟨mult(s(s(0)), s(s(0)), s(s(s(0)))), false⟩

can be given for learning the predicate mult.

(2) We obtain a background knowledge in the form of an equality theory, from the given logic program. Section 3 discusses how to obtain such a background knowledge.

From the above program, the following background knowledge is obtained.

$\mathtt{add}(0, Y) = Y$

$\mathtt{add}(\mathtt{s}(X), Y) = \mathtt{s}(\mathtt{add}(X, Y))$

(3) Using this background knowledge and the given positive and negative examples, our algorithm obtains a functional logic program in polynomial time. The following functional logic program is obtained from the above background knowledge and examples.

$\mathtt{mult}(0, Y, 0) \leftarrow$

$\mathtt{mult}(\mathtt{s}(X), Y, \mathtt{add}(Y, Z)) \leftarrow \mathtt{mult}(X, Y, Z)$

(4) The functional logic program obtained in the above step is transformed into an equivalent logic program.

$\mathtt{mult}(0, Y, 0) \leftarrow$

$\mathtt{mult}(\mathtt{s}(X), Y, Z)) \leftarrow \mathtt{mult}(X, Y, Z1), \mathtt{addition}(Y, Z1, Z)$

(5) The logic program obtained in step 4 may be transformed into a more efficient program, if needed, using transformations available in the logic programming literature. The logic program is now ready to be used as background knowledge in learning new predicates.

The programs learned in the above framework have a modular structure with conceptually separate components. The benefits of modularity in software engineering are well-known and widely discussed in the literature; they include (1) the divide-and-conquer approach in the analysis and verification, (2) ease of specification, (3) clear description, (4) interchangeability between different "plug-compatible" components, (5) reuse of components across applications, (6) separate analysis, optimization and compilation of components and (7) incremental and parallel evaluation (cf. [8]).

In deriving a functional logic program consistent with the given examples, we analyze all the terms which can possibly occur in a computation of this program. To avoid the combinatorial explosion and get polynomial-time learnability, we use the concepts of increasing and decreasing functions. The programs in our class have a nice property that the size of the terms occurring in any computation is bounded by the size of the terms in the initial query.

1.1. Related works

Our work has been inspired by the works of Miyano et al. [21, 22, 2] and Yamamoto [32]. Miyano et al. [21, 22, 2] identified a class of elementary formal systems (EFSs) – which are a special kind of logic programs manipulating character strings – and presented a polynomial-time algorithm to learn these programs from examples, without

asking any queries. This class, called *hereditary programs*,[1] contains logic programs with the following property: all the terms appearing in the body of a clause are subterms in the head. This property ensures that each term in any SLD-derivation is a subterm of a term in the initial query. The fact that a term of size n has at most n distinct subterms plays an important role in the polynomial-time learnability of hereditary programs. The standard append program is an example of hereditary programs. The condition that all the terms appearing in the body of a clause are subterms in the head is a bit restrictive from the programming point of view. It is not easy to write hereditary programs even for simple tasks like `reverse`, `merge`, `quick-sort` and `multiplication`.

Yamamoto [32] generalized the results of [21, 22, 2] using generalized unification as the background knowledge in the learning process. Some of the ideas in the framework of incremental learning have been indeed presented in the literature for the first time in [32]. Yamamoto [32] identified a class (called innermost simple programs) of logic programs which can be learned in polynomial time with certain restrictions on the background knowledge employed. The background knowledge about append and `addition` do not satisfy the restrictions of [32] and hence the standard programs for `multiplication`, `quick-sort`, `reverse` cannot be certified as polynomial-time learnable programs. Due to a syntactic condition of innermost simple programs, the standard `merge` program falls beyond the scope of the results of [32].

In this paper, we identify a class of polynomial-time learnable logic programs by relaxing certain syntactic conditions and the requirements on the background knowledge presented in [32]. Our class properly contains the classes of hereditary programs of [21, 22, 2] and innermost simple programs of [32]. Using our results, `multiplication`, `quick-sort`, `reverse` and `merge` programs can be certified as polynomial-time learnable.

The rest of the paper is organized as follows. The next section gives preliminary definitions and results needed. Section 3 explains the notion of regular background knowledge. We identify a class of programs, called hierarchical programs and prove some properties of their computations in Section 4 and establish their polynomial-time learnability in Section 5. A comparison with related works is given in Section 6. Section 7 concludes the paper with a summary.

2. Preliminaries

We assume that the reader is familiar with the basic terminology of logic programming and machine learning and use standard terminology from [20, 23, 24].

In the following, $\mathscr{T}(\Sigma, \mathscr{X})$ denotes the set of terms constructed from the set of function symbols Σ and the set of variables $\mathscr{X}$, $\mathscr{A}(\Pi, \Sigma, \mathscr{X})$ denotes the set of atoms constructed from these terms and the predicate symbols in Π. Throughout the paper, we

[1] Though Miyano et al. only considered EFSs, their definitions and results can be generalized to the usual logic programs.

use Π and Σ to denote the sets of predicate and function symbols under consideration. The size of a term $t \in \mathcal{T}(\Sigma, \mathcal{X})$, denoted by $|t|$, is the number of function symbols and variables occurring in it, and $var(t)$ is the set of variables in t. Terms which do not contain any variable are called ground terms and we use $\mathcal{T}(\Sigma)$ to denote the set of ground terms constructed from Σ. Atoms constructed from ground terms are ground atoms. A context over Σ is a term in $\mathcal{T}(\Sigma \cup \{\Box\}, \mathcal{X})$, where $\Box$ is a special symbol called *hole*. If $C[\Box, \ldots, \Box]$ is a context with n holes, $C[t_1, \ldots, t_n]$ denotes the term obtained by substituting the terms $t_1, \ldots, t_n$ for the n holes in the context C from left to right. We denote arity of a predicate/function symbol f by $arity(f)$.

As can be seen from the examples in the introduction, there are two kinds of function symbols in (functional) logic programs: function symbols like list constructors [ ] and $[\cdot|\cdot]$ are used for data structure building, and function symbols like app and add defined by the background knowledge are used to describe the predicates already learned. Accordingly, we partition Σ into two disjoint sets (i) Γ, the set of constructor symbols (corresponding to data structures) and (ii) Δ, the set of defined symbols (corresponding to predicates already known/defined). A constructor term is a term in $\mathcal{T}(\Gamma, \mathcal{X})$ and a constructor context is a context over Γ. We denote by $B(p)$, the set of atoms $\mathcal{A}(\{p\}, \Gamma, \phi)$.

Definition 1. The size of an atom $p(t_1, \ldots, t_n)$ is defined as $|p(t_1, \ldots, t_n)| = \max(|t_1|, \ldots, |t_n|)$. For a set S of atoms and an integer n, we define $S_n = \{A \in S;\ |A| \leq n\}$.

Definition 2. A logic Program P is a finite set of definite clauses of the form $H \leftarrow B_1, \ldots, B_n$, where $H, B_1, \ldots, B_n$ are atoms. The atom H is the head of this clause and $B_1, \ldots, B_n$ is the body of this clause. The length of a program P, denoted by $length(P)$, is defined as follows:

- $length(t) = 0$ if t is a constructor term,
- $length(a) = 1$ if a is a defined/predicate symbol with $arity(a) = 0$,
- $length(f(t_1, \ldots, t_m)) = m + length(t_1) + \cdots + length(t_m)$ if $f \in \Delta \cup \Pi$ and $m > 0$,
- length of a clause $C = H \leftarrow B_1, \ldots, B_n$ is defined as $length(C) = length(H) + length(B_1) + \cdots + length(B_n)$ and
- $length(P)$ is maximum over the lengths of clauses in P.

Informally, length of a clause is the sum of the arities of predicate/defined symbols in it plus the number of predicate/defined symbols of arity 0 in it.

We recall the following notions from [24, 21, 32, 16]. In the following, we use a special predicate symbol c to denote the target predicate to be learned.

Definition 3. A *concept* is a subset I of $B(c)$ and a *concept class* $\mathscr{C}$ is a subset of $2^{B(c)}$. For a concept class $\mathscr{C}$, we define $\mathscr{C}_n = \{I \cap B(c)_n;\ I \in \mathscr{C}\}$.

Definition 4. An *example* is a tuple $\langle A, a \rangle$ where $A \in B(c)$ and $a = \texttt{true}$ or $\texttt{false}$. It is *positive* if $a = \texttt{true}$ and *negative* otherwise. A concept I is *consistent* with a

sequence of examples $\langle A_1, a_1\rangle, \ldots, \langle A_m, a_m\rangle$ if $A_i \in I$ is equivalent to $a_i = \texttt{true}$ for each $i \in [1, m]$.

3. Background knowledge

In this paper, we use logic programs defining the already known predicates as background knowledge in learning a new predicate. This background knowledge is presented in the form of an equality theory (either conditional or unconditional) $\mathscr{B}$ over the defined and constructor symbols, Δ and Γ respectively. There are many techniques to derive a (conditional or unconditional) term rewrite system (i.e., a set of directed equations) from a given logic program (see e.g. [17, 29, 18, 10, 1]). The derived rewrite systems have a nice property that *in the left-hand sides, defined symbols occur only at the outermost level and constructor symbols do not occur at the outermost level.* Such rewrite systems are called *constructor systems.* For an equality theory $\mathscr{B}$ presented by a confluent and terminating constructor system, *narrowing* serves as a sound and complete E-unification method [12]. Throughout the paper, we consider equality theories presented by confluent and terminating constructor systems. Any equality theory $\mathscr{B}$ presented by such a rewrite system has the property: for every pair of constructor terms s and t, $\mathscr{B} \models s = t$ if and only if s and t are syntactically identical.

Definition 5. A *term rewriting system* (TRS, for short) $\mathscr{R}$ is a pair $(\mathscr{F}, R)$ consisting of a set $\mathscr{F}$ of function symbols and a set R of rewrite rules of the form $l \to r$ satisfying:

(i) $l, r \in \mathscr{T}(\mathscr{F}, \mathscr{X})$,
(ii) left-hand-side l is not a variable and
(iii) $Var(r) \subseteq Var(l)$.

A rule $l \to r$ applies to term t in $\mathscr{T}(\mathscr{F}, \mathscr{X})$, if a subterm s of t matches with l through some substitution σ, i.e. $s \equiv l\sigma$, and the rule is applied by replacing the subterm s in t by $r\sigma$ resulting in a new term u. This is formalized in the following definition.

Definition 6. The *rewrite relation* $\Rightarrow_{\mathscr{R}}$ induced by a TRS $\mathscr{R}$ is defined as follows: $s \Rightarrow_{\mathscr{R}} t$ if there is a rewrite rule $l \to r$ in $\mathscr{R}$, a substitution σ and a context $C[\ ]$ such that $s \equiv C[l\sigma]$ and $t \equiv C[r\sigma]$.

We say that s *reduces to* t *in one rewrite* (or reduction) *step* if $s \Rightarrow_{\mathscr{R}} t$ and say s *reduces to* t if $s \Rightarrow^*_{\mathscr{R}} t$, where $\Rightarrow^*_{\mathscr{R}}$ is the reflexive-transitive closure of $\Rightarrow_{\mathscr{R}}$.

Termination and confluence are the most important properties of term rewriting systems. Termination ensures that each computation is of finite length, whereas confluence ensures that the nondeterministic choice of where to apply the rewrite step in any term has no effect on the result of a computation, that is, all the computations results in the same value.

Definition 7. A term rewriting system $\mathscr{R}$ is *terminating* if there is no infinite rewriting derivation $t_1 \Rightarrow_{\mathscr{R}} t_2 \Rightarrow_{\mathscr{R}} t_3 \Rightarrow_{\mathscr{R}} \cdots$.

Definition 8. A term rewriting system $\mathscr{R}$ is *confluent* if there exists a term v such that $t \Rightarrow^*_{\mathscr{R}} v$ and $u \Rightarrow^*_{\mathscr{R}} v$ whenever $s \Rightarrow^*_{\mathscr{R}} t$ and $s \Rightarrow^*_{\mathscr{R}} u$.

Both termination and confluence properties are undecidable in general. However, a lot of techniques and tools are available in the literature to prove termination of many classes of term rewriting systems. For terminating systems, it is relatively simple to verify the confluence property. In particular, it is enough to prove that all the critical pairs are joinable to ensure the confluence property of a term rewriting system.

Definition 9. Let $l_1 \to r_1$ and $l_2 \to r_2$ be renamed versions of rules in a rewrite system $\mathscr{R}(\mathscr{F}, R)$ such that they have no variables in common. Suppose a nonvariable subterm s of l_1 unifies with l_2 through a most general unifier σ, i.e. $s\sigma \equiv l_2\sigma$ and let C be a context such that $l_1 \equiv C[s]$. *The pair of terms* $\langle C[r_2]\sigma,\ r_1\sigma\rangle$ *is called a critical pair of* $\mathscr{R}(\mathscr{F}, R)$. If $l_1 \to r_1$ and $l_2 \to r_2$ are renamed versions of the same rewrite rule, we do not consider the case $C = \square$.

For checking confluence property of a terminating term rewriting system, one can rewrite the two terms t_1 and t_2 of each critical pair $\langle t_1,\ t_2\rangle$ until they cannot be further rewritten and check for equality. That is, (1) rewrite t_i to t_i' such that (a) $t_i \Rightarrow^*_{\mathscr{R}} t_i'$ and (b) t_i' is irreducible and (2) check whether $t_1' \equiv t_2'$. The following lemma ensures correctness of this procedure.

Lemma 1. *A terminating term rewriting system* $\mathscr{R}$ *is confluent if each of its critical pairs is joinable, i.e.* $[\exists v\ t_1 \Rightarrow^*_{\mathscr{R}} v$ *and* $t_2 \Rightarrow^*_{\mathscr{R}} v]$ *for each critical pair* $\langle t_1,\ t_2\rangle$ *of* $\mathscr{R}$.

Example 1. The equality theory presented by the following rewrite system is a background knowledge $\mathscr{B}$ about append.

```
app([ ], Ys) → Ys

app([X|Xs], Ys) → [X|app(Xs, Ys)]
```

One can see the similarity of this term rewrite system and a logic program defining append.

It is easy to note that this rewrite system is terminating as the recursive argument of app gets smaller after each application of the second rewrite rule. The system is confluent as it is terminating and has no critical pair.

Definition 10. A *unifier* of two terms s and t in $\mathscr{T}(\Sigma, \mathscr{X})$ *w.r.t. an equality theory* $\mathscr{B}$ is a substitution θ such that $\mathscr{B} \models s\theta = t\theta$. A *unifier* of two atoms $p(s_1, \ldots, s_n)$ and $p(t_1, \ldots, t_n)$ *w.r.t.* $\mathscr{B}$ is a substitution θ such that $\mathscr{B} \models s_1\theta = t_1\theta \wedge \cdots \wedge s_n\theta = t_n\theta$. Two terms (or atoms) are *unifiable* if there exist a unifier for them.

A unifier $\theta = \{X_1/s_1, \ldots, X_n/s_n\}$ of two terms (or atoms) s and t is *relevant* if $\{X_1, \ldots, X_n\} \subseteq var(s) \cup var(t)$.

Example 2. The following are some of the relevant unifiers of terms $\mathtt{s} = \mathtt{app(X, Y)}$ and $\mathtt{t} = [1,2,3]$ with respect to the background knowledge $\mathscr{B}$ given in the above example.

$\theta_1 = \{\mathtt{X}/[\ ],\ \mathtt{Y}/[1,2,3]\}$; $\theta_2 = \{\mathtt{X}/[1],\ \mathtt{Y}/[2,3]\}$;

$\theta_3 = \{\mathtt{X}/[1,2],\ \mathtt{Y}/[3]\}$; $\theta_4 = \{\mathtt{X}/[1,2,3],\ \mathtt{Y}/[\ ]\}$.

As mentioned above, *narrowing* serves as a sound and complete E-unification method for an equality theory $\mathscr{B}$ presented by a confluent and terminating constructor system [12]. In the following, we only consider the background knowledges presented by confluent and terminating constructor systems. This is a reasonable requirement as the transformations which derive term rewriting systems from logic programs (defining the already known predicates) are becoming more and more powerful, and often derive terminating rewrite systems from terminating logic programs.

To establish polynomial-time learnability of hierarchical programs, we need to put some restrictions on the background knowledge employed. We explain these restrictions in the following and by $\mathscr{B}$, we denote the background knowledge under consideration.

Definition 11. A defined symbol $f \in \Delta$ is *increasing* if $|t| \geqslant |t_i|$ for each $i \in [1,n]$ whenever $\mathscr{B} \models f(t_1, \ldots, t_n) = t$ and $t_1, \ldots, t_n, t$ are constructor terms.

A defined symbol $f \in \Delta$ is *decreasing* if $|t| \leqslant |t_i|$ for some $i \in [1,n]$ whenever $\mathscr{B} \models f(t_1, \ldots, t_n) = t$ and $t_1, \ldots, t_n, t$ are constructor terms.

Example 3. Consider the background knowledge $\mathscr{B}$ about `add` presented by the following rewrite system.

$\mathtt{add(0, Y)} \rightarrow \mathtt{Y}$

$\mathtt{add(s(X), Y)} \rightarrow \mathtt{s(add(X, Y))}$

It is very easy to see that the defined symbol `add` is increasing. Similarly, the defined symbol `app` presented in Example 1 is increasing.

Example 4. Consider the background knowledge $\mathscr{B}$ about `mod` and `subtract` functions presented by the following rewrite system.

$\mathtt{mod}(X, Y) \rightarrow X$ if $\mathtt{X} < \mathtt{Y}$ and $\mathtt{Y} > 0$

$\mathtt{mod(X, Y)} \rightarrow \mathtt{mod(subtract(X, Y), Y)}$ if $\mathtt{X} \geqslant \mathtt{Y}$ and $\mathtt{Y} > 0$

$\mathtt{subtract(X, 0)} \rightarrow \mathtt{X}$

$\mathtt{subtract(s(X), s(Y))} \rightarrow \mathtt{subtract(X, Y)}$

The defined symbol `mod` is decreasing as the value of $\mathtt{mod(X, Y)}$ is less than or equal to `X` when $\mathtt{mod(X, Y)}$ is defined (i.e., when $\mathtt{Y} > 0$). Similarly, `subtract` is decreasing as

the value of subtract(X, Y) is less than or equal to X when subtract(X, Y) is defined (i.e., when $X \geqslant Y$).

The algorithms presented in the literature for generating linear predicate inequalities from given logic programs – such as the ones presented in Ullman and van Gelder [30] and Plümer [25] – can be used in proving that the defined functions in the background knowledge corresponding to the predicates already known are increasing or decreasing.

To capture the set of all terms that occur in a computation, we introduce the following notion of *dependent set*. In the following, we only consider background knowledge such that Δ can be partitioned as $\Delta_1 \uplus \Delta_2$ with Δ_1 containing increasing functions and Δ_2 containing decreasing functions.

Remark 1. Note that the above definition of increasing and decreasing functions allows a function to be both increasing and decreasing. When a defined function f is both increasing and decreasing, we put f in either Δ_1 or Δ_2 depending on whether f occurs in the heads or the bodies of the clauses in the program under consideration. If f only occurs in the heads, it is placed in Δ_1 and if f only occurs in the bodies, it is placed in Δ_2. The motivation for this will become clear in the sequel.

Definition 12. The *dependent set* $D(t)$ of a ground constructor term t is defined as

(1) $t \in D(t)$ and
(2) the ground constructor terms $s_1, \ldots, s_n \in D(t)$ if $\mathscr{B} \models f(s_1, \ldots, s_n) = s$ for some $f \in \Gamma \cup \Delta_1$ and $s \in D(t)$.
(3) the ground constructor term $s \in D(t)$ if $\mathscr{B} \models f(s_1, \ldots, s_n) = s$ for some $f \in \Delta_2$ and $s_1, \ldots, s_n \in D(t)$.

It is easy to see that $D(t)$ is closed under subterms, i.e., every subterm of s is in $D(t)$ if $s \in D(t)$. The following lemma shows that the size of the terms in $D(t)$ is bounded by the size of t.

Lemma 2. *For each term* $u \in D(t)$, $|u| \leqslant |t|$.

Proof (*Induction*). Let us consider the 3 cases in the above definition. (1) Lemma holds trivially for $t \in D(t)$. (2) The ground constructor terms $s_1, \ldots, s_n \in D(t)$ if $\mathscr{B} \models f(s_1, \ldots, s_n) = s$ and $s \in D(t)$ for some $f \in \Gamma \cup \Delta_1$. By induction hypothesis, $|s| \leqslant |t|$. If $f \in \Gamma$, each s_i is a subterm of s and hence $|s_i| < |t|$. If $f \in \Delta_1$, each $|s_i| \leqslant |s|$ as f is increasing. Hence $|s_i| \leqslant |t|$. (3) The ground constructor term $s \in D(t)$ if $\mathscr{B} \models f(s_1, \ldots, s_n) = s$ and $s_1, \ldots, s_n \in D(t)$ for some $f \in \Delta_2$. By induction hypothesis, each $|s_i| \leqslant |t|$. It follows that $|s| \leqslant |t|$ from the fact that f is decreasing and each $|s_i| \leqslant |t|$. □

Definition 13. We say that a background knowledge $\mathscr{B}$ has *polynomial dependency property* if $|D(t)|$ is bounded by a polynomial in $|t|$.

Example 5. Consider the background knowledge $\mathscr{B}$ about app given in Example 1. For a list L, $D(L)$ is the set of sublists of L. The number of sublists of a list L of length n is $(n+1)_{C_2}$, which is of the order $O(n^2)$. Therefore, $\mathscr{B}$ has polynomial dependency property.

Basically, a sublist of L can be identified by its two end-points. So to compute the number of sublists, we need to compute the number of possible ways of choosing two points on a line with $n+1$ points. That is, the number of sublists of a list L of length n is $(n+1)_{C_2}$.

Similarly, the background knowledge $\mathscr{B}$ about add presented in Example 3 has polynomial dependency property as $D(n)$ is the set of natural numbers less than or equal to n.

Example 6. Consider the background knowledge $\mathscr{B}$ about mod and subtract functions presented in Example 4. This has polynomial dependency property as $D(n)$ is the set of natural numbers less than or equal to n.

Definition 14. A defined symbol $f \in \Delta_1$ is *solvable* if there exists an algorithm which takes a term $t \in \mathscr{T}(\Gamma)$ as input and outputs all the tuples $\langle t_1, \ldots, t_n \rangle$ of ground constructor terms satisfying $\mathscr{B} \models f(t_1, \ldots, t_n) = t$ if exists and reports failure otherwise. Further, f is *polynomial-time solvable* if the algorithm runs in polynomial-time of $|t|$.

Similarly, a defined symbol $f \in \Delta_2$ is *solvable* if there exists an algorithm which takes a tuple $\langle t_1, \ldots, t_n \rangle$ of ground constructor terms and outputs a term $t \in \mathscr{T}(\Gamma)$ satisfying $\mathscr{B} \models f(t_1, \ldots, t_n) = t$ if exists and reports failure otherwise. Further, f is *polynomial-time solvable* if the algorithm runs in polynomial-time of $|t_1| + \cdots + |t_n|$.

Remark 2. Any function symbol defined by a confluent and terminating term rewriting system is solvable, as narrowing serves as the E-unification algorithm.

The following example gives two polynomial-time solvable functions.

Example 7. Consider the background knowledge $\mathscr{B}$ presented in Example 1. The defined symbol app is polynomial-time solvable, as list of length $n \geqslant 1$ can be broken into two sublists in $n+1$ different ways. Similarly, the defined symbol add defined by the background knowledge $\mathscr{B}$ presented in Example 3 is polynomial-time solvable, as a natural number n can be split into two natural numbers in $n+1$ different ways.

The defined symbols mod and subtract of the background knowledge presented in Example 4 are well-known to be polynomial-time solvable.

Definition 15. A background knowledge $\mathscr{B}$ is *regular* if it has the polynomial dependency property and each defined symbol in it is polynomial-time solvable.

Example 8. The background knowledge $\mathscr{B}$ about (1) app given in Example 1, (2) add given in Example 3 and (3) mod and subtract given in Example 4 are all regular as we have proved both the above requirements.

4. Hierarchical programs

In this section, we define a class of logic programs and study certain properties of their computations.

Definition 16. Let $\mathscr{B}$ be a regular background knowledge with defined symbols $\Delta_1 \uplus \Delta_2$. A definite clause $H \leftarrow B_1, \ldots, B_n$ is *hierarchical* w.r.t. $\mathscr{B}$ if

(a) all the arguments of defined symbols occurring in it are constructor terms,
(b) no function symbol in Δ_2 occur in H and no function symbol in Δ_1 occur in $B_1, \ldots, B_n$ and
(c) each argument in $B_1, \ldots, B_n$ is either a constructor term occurring in H or of the form $f(t_1, \ldots, t_n)$ such that $f \in \Delta_2$ and each t_k is a constructor term occurring in H.

A logic program is *hierarchical* w.r.t. $\mathscr{B}$ if each clause in it is hierarchical.

Informally, the arguments of H are either constructor terms or terms containing both constructor and defined symbols in Δ_1 but without any nesting of defined symbols. The arguments in the body are either constructor terms or terms containing both constructor and defined symbols in Δ_2 but with defined symbols only at the topmost level and all the constructor terms in the body also occur in H.

Example 9. The following program for `greatest common divisor`

$$\texttt{gcd(X,0,X)} \leftarrow \texttt{X} > \texttt{0}$$
$$\texttt{gcd(X,Y,Z)} \leftarrow \texttt{gcd(Y,mod(X,Y),Z)}$$

is hierarchical w.r.t. the background knowledge $\mathscr{B}$ given in Example 4 as `mod` is a decreasing function and occurs only in the body.
Similarly, the programs for `multiplication` and `reverse` given in the introduction are hierarchical w.r.t. the background knowledges about `add` and `app` respectively, as the functions `add` and `app` are increasing and occur only in the heads.

All the three conditions in the definition of hierarchical programs are purely syntactical conditions and can be checked by scanning each clause in the program once.

Lemma 3. *It is decidable whether a given program is* hierarchical *w.r.t. a given regular background knowledge.*

A computation of a hierarchical program w.r.t. a regular background knowledge involves repeated application of the following computation step. We say a goal is a ground constructor goal if every term in it is a ground constructor term. The selection rule is fixed a priori.

Definition 17. A *computation step* derives a ground constructor goal

$$G' = \leftarrow A_1, \ldots, A_{l-1}, \overline{B_1\theta}, \ldots, \overline{B_j\theta}, A_{l+1}, \ldots, A_k$$

from (1) a clause $H \leftarrow B_1, \ldots, B_j$ in a hierarchical program P w.r.t. a regular background knowledge $\mathscr{B}$ and (2) a ground constructor goal $G = \leftarrow A_1, \ldots, A_k$ as follows:

(i) let θ be a relevant most general unifier w.r.t. $\mathscr{B}$ of H and the selected atom A_l in G,
(ii) for each $i \in [1, j]$, let $\overline{B_i\theta}$ be the *ground constructor* atom obtained from $B_i\theta$ by reducing each term of the form $f(t_1, \ldots, t_m)$, $f \in \Delta_2$ in it to the normal form w.r.t. the rewrite system representing $\mathscr{B}$,
(iii) if no such *ground constructor atom* $\overline{B_i\theta}$ occurs for an atom $B_i\theta$, *no computation step is possible* from G with input clause $H \leftarrow B_1, \ldots, B_j$.

Definition 18. A *computation* of a hierarchical program P w.r.t. a regular background knowledge $\mathscr{B}$ starting with a ground constructor goal G is a sequence $G_0, \ldots, G_n$ of ground constructor goals satisfying the following:
(i) $G_0 = G$ and
(ii) for each $i \in [1, n]$, G_i is derived from G_{i-1} and a clause in P by a computation step.

Now, we prove an important property of computations of hierarchical programs.

Lemma 4. *If P is a hierarchical program w.r.t. a regular background knowledge $\mathscr{B}$ and $\leftarrow p(t_1, \ldots, t_n)$ is a ground constructor goal, then all the terms occurring in any computation of P starting with $\leftarrow p(t_1, \ldots, t_n)$ are ground constructor terms in $S = D(t_1) \cup \cdots \cup D(t_n)$.*

Proof (*Induction on the length l of the computation*).

Basis: $l = 0$. The lemma follows from the fact that $t \in D(t)$ for any term t.

Induction hypothesis: Assume that the lemma holds for all computations of length $l < k$.

Induction step: Now, we establish that it holds for $l = k$. Let A be the selected atom and $H \leftarrow B_1, \ldots, B_n$ be the input clause used in the last computation step. By the induction hypothesis, all the arguments of A are ground constructor terms belonging to S. We have to show that all the arguments of $\overline{B_1\theta}, \ldots, \overline{B_n\theta}$ are ground constructor terms in S, where θ is a most general unifier of A and H.

Consider an argument $t\theta$ in $B_1\theta, \ldots, B_n\theta$. Since P is a hierarchical program, t is a constructor subterm of an argument (say, s) in H if t is a constructor term. Let s' be the term in the corresponding argument-position of A. By the induction hypothesis, $s' \in S$. If s is a constructor term it is obvious that $t\theta$ is a subterm of $s\theta \equiv s'$. Since $s' \in S$, all its subterms are in S as well and hence $t\theta \in S$. If s contains defined symbols, it can be written as $C[s_1, \ldots, s_m]$, where C is a context of constructor

symbols and variables and $root(s_i)$ is a defined symbol for each $i \in [1,m]$. It is clear that s' can be written as $C[s'_1,\ldots,s'_m]$ and each $s'_i \in S$. Now, t is either a subterm of C or one of s_i. If t is a subterm of C, it can be proved $t\theta \in S$ as above. Let us now consider the case that t is a subterm of s_i. Since, P is a hierarchical program, s_i is of the form $f(t_1,\ldots,t_h)$, where $f \in \Delta_1$ and each t_j is a constructor term. Since θ is a unifier of A and H, it follows that $\mathscr{B} \models f(t_1\theta,\ldots,t_h\theta) = s'_i$. Since $s'_i \in S$ it follows from Definition 12 that each $t_j\theta \in S$. Since t is a constructor subterm of s_i, it must be a subterm of some t_j. Therefore, $t\theta$ is a subterm of $t_j\theta$ and hence $t\theta \in S$.

Now consider the case that t is not a constructor term. Since P is a hierarchical program, t is of the form $f(t_1,\ldots,t_n)$ such that $f \in \Delta_2$ and each t_k is a constructor subterm of an argument in H. As in the above case, each $t_k\theta \in S$ and by Definition 12, the normal form of $f(t_1\theta,\ldots,t_n\theta)$ w.r.t. the rewrite system defining $\mathscr{B}$ belongs to S. Therefore the argument in $\overline{B_1\theta},\ldots,\overline{B_n\theta}$ corresponding to t is in S. This completes the proof. □

From this lemma and Lemma 2, we get the following theorem.

Theorem 1. *If P is a hierarchical program w.r.t. a regular background knowledge $\mathscr{B}$ and $\leftarrow p(t_1,\ldots,t_n)$ is a ground constructor goal, then all the atoms occurring in any computation starting with $\leftarrow p(t_1,\ldots,t_n)$ are of size less than or equal to $|p(t_1,\ldots,t_n)|$.*

5. Polynomial-time learnability

In this section, we prove polynomial-time learnability of hierarchical programs w.r.t. regular background knowledge.

Definition 19. For a hierarchical program P, we define the semantics[2] $M(P)$ as $\{A \in B(\Gamma) |$ there is a computation of P starting with a ground constructor goal $\leftarrow A$ ending in the empty goal$\}$.

We need the following lemma.

Lemma 5. *Let H be the head of a hierarchical clause w.r.t. a regular background knowledge $\mathscr{B}$. If H unifies with a ground constructor atom of A w.r.t. $\mathscr{B}$, the size of each constructor subterm in H and each constructor context in H is less than or equal to the size of A.*

[2] The semantics $M(P)$ can also be defined as the least fixpoint of a monotonic operator (similar to the T_P operator in [24]) in a straightforward way.

Proof. Let $|A|=n$. By definition, each argument s of A has $|s|\leqslant n$. Now, consider an argument t of H and the corresponding argument s of A. If t is a constructor term, it is obvious that $|t|\leqslant n$ as it is unifying with a ground constructor term s with $|s|\leqslant n$. Otherwise, t can be written as $C[f_1(t_{11},\ldots,t_{1n_1}),\ldots,f_m(t_{m1},\ldots,t_{mn_m})]$ such that C is a context of constructor symbols and variables, $f_i\in\Delta_1$ for each $i\in[1,m]$ and each t_{ij} is a constructor term. It is obvious that $|C|\leqslant n$ as s should be of the form $C[\ldots]$. Each $f_i(t_{i1},\ldots,t_{in_i})$ unifies with a subterm s_i of s and $|s_i|\leqslant n$. That is, $\mathscr{B}\models f_i(t_{i1}\theta,\ldots,t_{in_i}\theta)=s_i$. Since f_i is an increasing function, $|t_{ij}\theta|\leqslant|s_i|\leqslant n$ for each $j\in[1,n_i]$. Hence $|t_{ij}|\leqslant n$ for each $j\in[1,n_i]$. This completes the proof. □

Definition 20. For $\mathtt{k},\mathtt{l},\mathtt{m}\geqslant 0$, $\mathtt{Hier}(\mathscr{B},\mathtt{k},\mathtt{l},\mathtt{m})$ is the class of concepts definable by hierarchical programs P w.r.t. $\mathscr{B}$ having at most $\mathtt{m}$ variable occurrences in the head of any clause and satisfying $|P|\leqslant\mathtt{k}$ and $length(P)\leqslant\mathtt{l}$. We also use $\mathtt{Hier}(\mathscr{B},\mathtt{k},\mathtt{l},\mathtt{m})$ to denote the above class of hierarchical programs.

Note that the number of defined symbols in a clause is bounded by its length.

Lemma 6. *Let $\mathscr{B}$ be a regular background knowledge, A be a ground constructor atom $p(s_1,\ldots,s_h)$ and C be the set of all clauses in $\mathtt{Hier}(\mathscr{B},\mathtt{k},\mathtt{l},\mathtt{m})$ which* (1) *have at most k distinct predicate symbols and* (2) *their heads unify with A. Then $|C|$ is bounded by a polynomial in the size of A.*

Proof. Let $H\leftarrow B_1,\ldots,B_{n_1}$ be a clause satisfying the above conditions. By the above lemma, the size of each constructor subterm in H and each constructor context in H is less than or equal to the size of A (say, n). By the definition of length of a clause, there are at most $\mathtt{l}$ such constructor contexts or maximal constructor subterms in H. That is, the number of symbols in H is at most $n.\mathtt{l}$. Further, there are at most $\mathtt{l}-1$ Δ_1-symbols in H and the arities of all these symbols add up to at most $\mathtt{l}-1$.

- The $\mathtt{l}-1$ Δ_1-symbols in H can occur in $\mathbf{choose}(n.\mathtt{l},(\mathtt{l}-1))$ possible positions. This is in the order of $(n.\mathtt{l})^{\mathtt{l}-1}$.
- There are $|\Delta_1|^{(\mathtt{l}-1)}$ ways of choosing $\mathtt{l}-1$ Δ_1-symbols.
- The arguments of Δ_1-symbols in $H\theta$ are in $\bigcup_{i\in[1,h]}D(s_i)\}$, where θ is an mgu of A and H. There are $|\bigcup_{i\in[1,h]}D(s_i)\}|^{(\mathtt{l}-1)}$ ways of choosing arguments for Δ_1-symbols in $H\theta$.
- H has at most $\mathtt{m}$ distinct variables which can occur in $\mathbf{choose}(n.\mathtt{l},\mathtt{m}).\mathtt{m}^{\mathtt{m}}$ ways in H. This is in the order of $(n.\mathtt{l}.\mathtt{m})^{\mathtt{m}}$
- Therefore, there are at most $(n.\mathtt{l})^{\mathtt{l}}.|\Delta_1|^{\mathtt{l}}.|\bigcup_{i\in[1,h]}D(s_i)\}|^{\mathtt{l}}.(n.\mathtt{l}.\mathtt{m})^{\mathtt{m}}$ ways of choosing H. A polynomial in n (note that $|\bigcup_{i\in[1,h]}D(s_i)\}|$ is bounded by a polynomial in n by the polynomial dependency property of $\mathscr{B}$).

By the definition of length of a clause, there are at most $\mathtt{l}-2$ predicate or Δ_2-symbols in the body, i.e., $(\mathtt{k}+|\Delta_2|)^{(\mathtt{l}-2)}$ possibilities. Further, there are at most $\mathtt{l}-2$ constructor subterms in the body and these subterms occur in H, i.e., $(n.\mathtt{l})^{\mathtt{l}-2}$

possibilities. Therefore, there are at most $(\mathtt{k}+|\Delta_2|)^{\mathtt{l}}.(n.\mathtt{l})^{\mathtt{l}}$ ways of choosing the body of a clause with head H. A polynomial in n. Hence $|C|$ is bounded by a polynomial in n. □

Now, we are in a position to establish our main result. Our assumption in the above lemma that there are at most k distinct predicate symbols is justified by the following. If there is a predicate p which occurs in a clause c in a program P but does not occur in the head of any clause in P, then $M(P)=M(P-\{c\})$. So, it suffices to consider programs in $\mathtt{Hier}(\mathscr{B},\mathtt{k},\mathtt{l},\mathtt{m})$ which have at most k distinct predicate symbols.

Theorem 2. *If the background knowledge $\mathscr{B}$ is regular, then the class of concepts* $\mathtt{Hier}(\mathscr{B},\ \mathtt{k},\ \mathtt{l},\ \mathtt{m})$ *is polynomial-time learnable for any* $\mathtt{k},\mathtt{l},\mathtt{m}\geqslant 0$.

Proof. We prove that one can find a program $P\in\ \mathtt{Hier}(\mathscr{B},\ \mathtt{k},\ \mathtt{l},\ \mathtt{m})$ consistent with a given sequence S of both positive and negative examples of a concept in $\mathtt{Hier}(\mathscr{B},\mathtt{k},\mathtt{l},\mathtt{m})$ in polynomial time. The idea behind the proof is essentially same as that in [21,22,32]. Let $\Pi=\{p_1,\ldots,p_{\mathtt{k}.(\mathtt{l}+1)}\}$ be the set of predicate symbols with $arity(p_i)=i \bmod (\mathtt{l}+1)$ and $c=p_j$ be the target predicate for some $j\in[1,\mathtt{k}.(\mathtt{l}+1)]$.

Given a sequence S of examples, we can produce a program P consistent with S as follows. Let S^+ be the set of positive examples in S. If $S^+=\phi$, the algorithm returns $P=\phi$. Otherwise, let $\mathscr{S}=\{t\in\mathscr{T}(\Gamma);t=t_i$ for some $c(t_1,\ldots,t_h)\in S^+\}$. For each $i\in[1,\mathtt{k}.(\mathtt{l}+1)]$, let H_i denote the set of atoms $\{p_i(t_1,\ldots,t_h);\ h=i \bmod (\mathtt{l}+1)$ and $t_1,\ldots,t_h\in\bigcup_{t\in\mathscr{S}}D(t)\}$. By the polynomial dependency property of $\mathscr{B}$, $|H_i|$ is polynomial in $\sum_{s\in\mathscr{S}}|s|$ for every $i\in[1,\mathtt{k}.(\mathtt{l}+1)]$.

We first generate a set $\mathscr{P}$ of all programs $P\in\mathtt{Hier}(\mathscr{B},\mathtt{k},\mathtt{l},\mathtt{m})$ satisfying: *if there is a clause $H\leftarrow B_1,\ldots,B_n$ in P and the predicate symbol of H is p_i then there exists a $A\in H_i$ such that A and H are unifiable w.r.t. $\mathscr{B}$*. This set $\mathscr{P}$ can be generated in polynomial time as $|\mathscr{P}|$ is polynomial in $\sum_{s\in\mathscr{S}}|s|$ by the above lemma.

We then check for each $P\in\mathscr{P}$ and $s\in S$ whether $s\in M(P)$ or not, as required for the consistency, by constructing a bottom-up proof for s. By Lemma 4, all the atoms appearing in the proof are in the set $\bigcup_{i\in[1,\mathtt{k}.(\mathtt{l}+1)]}H_i$. Hence the proof goes in polynomial in $\sum_{s\in\mathscr{S}}|s|$. This completes the proof. □

Now, we give a few examples illustrating our main result.

Example 10. The programs for `reverse` and `multiplication` given in the introduction are polynomial-time learnable as they are in $\mathtt{Hier}(\mathscr{B},2,6,4)$ and $\mathtt{Hier}(\mathscr{B},2,8,4)$ respectively and the corresponding background knowledges are regular.

Example 11. The `greatest common divisor` program given in Example 9 is polynomial-time learnable as it is in $\mathtt{Hier}(\mathscr{B},2,8,3)$ and the background knowledge $\mathscr{B}$ presented in Example 4 is regular.

Example 12. The following program for merge is hierarchical w.r.t. an empty background knowledge $\mathscr{B}$.

$$\texttt{merge}([\,],[\,],[\,]) \leftarrow$$
$$\texttt{merge}([X|Xs],[Y|Ys],[X|Zs]) \leftarrow X < Y,\ \texttt{merge}(Xs,[Y|Ys],Zs)$$
$$\texttt{merge}([X|Xs],[Y|Ys],[Y|Zs]) \leftarrow X \geqslant Y,\ \texttt{merge}([X|Xs],Ys,Zs)$$

This program is polynomial-time learnable as it is in $\texttt{Hier}(\mathscr{B},3,8,6)$.

Example 13. The following program for quick-sort over lists of distinct elements (need not be natural numbers but on any set with a total order) is hierarchical w.r.t. the background knowledge $\mathscr{B}$ about append. This program is non-conventional and needs an explanation. The lack of local variables (which occur in the body but not in the head) in hierarchical programs contributes to the non-simplicity of the program. The second clause of qs says that app(A, [H|B]) is the result of (quick) sorting [H|L] if A is the sorted list of all the elements in L smaller than H (implied by the atoms less(H, L, A) and qs(A, A) in the body) and B is the sorted list of all the elements in L bigger than H (implied by the atoms great(H, L, B) and qs(B, B) in the body). The atom ls1(H, L, A) stands for the fact that A contains all the elements in L smaller than H and the atom ls2(H, A) stands for the fact that all the elements in A are smaller than H. The meanings of predicates gr1 and gr2 are similar.

```
qs([ ],[ ]) ←
qs([H|L], app(A,[H|B])) ← less(H,L,A), great(H,L,B), qs(A,A), qs(B,B)

less(H,L,A) ← subset(A,L), ls1(H,L,A), ls2(H,A)
ls1(H,[ ],A) ←
ls1(H,[Y|Ys],A) ← H>Y, member(Y,A), ls1(H,Ys,A)
ls1(H,[Y|Ys],A) ← H<Y, ls1(H,Ys,A)
ls2(H,[ ]) ←
ls2(H,[Y|Ys]) ← H>Y, ls2(H,Ys)

great(H,L,A) ← subset(A,L), gr1(H,L,A), gr2(H,A)
gr1(H,[ ],A) ←
gr1(H,[Y|Ys],A) ← H<Y, member(Y,A), gr1(H,Ys,A)
gr1(H,[Y|Ys],A) ← H>Y, gr1(H,Ys,A)
gr2(H,[ ]) ←
gr2(H,[Y|Ys]) ← H<Y, gr2(H,Ys)

subset([ ],L) ←
subset([X|Xs],L) ← member(X,L), subset(Xs,L)

member(H,[H|L]) ←
member(H,[Y|Ys]) ← member(H,Ys)
```

This program is polynomial-time learnable as it is in $\texttt{Hier}(\mathscr{B},18,14,5)$ and $\mathscr{B}$ is regular.

6. Comparison with related works

The present paper pursues the line of research presented in [21, 22, 2, 32, 16]. We first compare our results with the results of [21, 22, 2, 32, 16] and then compare with the other related works by Cohen, Dzeroski, Kietz, Frazier, Page, Lapointe, Matwin and Idestam-almquist.

Our results are generalizations of the results in Yamamoto [32] in the following respects:

(i) The class of innermost-simple programs is a proper subclass of the class of hierarchical programs, as innermost-simple programs allow only variables in the body and these variables should occur in the head of the clause. For this reason, the class of innermost-simple programs [32] does not contain the class of hereditary programs of [21, 22, 2]. Hierarchical programs allow non-variable constructor terms in the body if these terms are subterms of the terms in the head of the clause. The class of hierarchical programs contains both the class of hereditary programs and the class of innermost-simple programs.

(ii) In Yamamoto [32], it is essential that each defined symbol is *completely defined* (over $\mathcal{T}(\Gamma)$). For the `quick-sort` program (with constructors: [ ], $[\cdot|\cdot]$ and natural numbers), it is not clear how to make the defined function `append` a completely defined one since `append` is defined only on lists (but not on natural numbers). We do not need this requirement. Our requirement that the rewrite system presenting the background knowledge is confluent ensures that for any tuple $\langle t_1, \ldots, t_n \rangle$ of ground constructor terms there is at most one ground constructor term t such that $\mathscr{B} \models f(t_1, \ldots, t_n) = t$ for any defined symbol $f \in \Delta$.

(iii) Yamamoto [32] needs that each defined symbol is *injective* in the following sense that for any ground constructor term t there is at most one tuple $\langle t_1, \ldots, t_n \rangle$ of ground constructor terms such that $\mathscr{B} \models f(t_1, \ldots, t_n) = t$ for any defined symbol $f \in \Delta$. We do not impose such a requirement and find *injectivity* very restrictive as many defined functions such as `app` and `add` are not injective. We notice that in the revised version of [32], the requirement of injectivity is omitted.

(iv) Our notion of *increasing* function is a generalization of the corresponding notion in [32]. Our notion only needs that $|t| \geqslant |t_i|$ for each $i \in [1, n]$ when $\mathscr{B} \models f(t_1, \ldots, t_n) = t$, while the notion in [32] needs that $|t| \geqslant |t_1| + \cdots + |t_n|$ when $\mathscr{B} \models f(t_1, \ldots, t_n) = t$. For this reason, defined symbols `add` and `append` given in the previous sections are not increasing in the sense of [32].

None of the above programs `reverse`, `multiplication`, `quick-sort` and `merge` can be certified as polynomial-time learnable by the results of [32]. Program `merge` cannot be certified as there are nonvariable terms `[X|Xs]` and `[Y|Ys]` in the bodies of the two non-unit clauses. Programs `reverse`, `multiplication` and `quick-sort` cannot be certified as the defined symbols `append` and `add` are not increasing in the sense of [32]. However, a slightly different program (given in the following Example) for `reverse` can be proved as polynomial-time learnable by the results of [32].

Example 14. Consider the following program and background knowledge.

```
reverse(reverse[ ], [ ]) ←
reverse([X|Xs], addlast(X, Ys)) ← reverse(Xs, Ys)

ℬ : addlast(X, [ ]) → [X]
    addlast(X, [Y|Ys]) → [Y|addlast(X, Ys)]
```

This background knowledge is regular in the sense of [32] (also our sense) and the program is innermost-simple. Hence it is polynomial-time learnable.

The function `addlast` is a specialized version of `app` and appends an element at the end of a given list. These two are equally efficient in this special case. Apparently, `multiplication`, `quick-sort` and `merge` cannot be certified as polynomial-time learnable by the results of [32] through such modifications as in the case of `reverse`.

6.1. *The class of generalized-hereditary programs*

In Krishna Rao [16], the class of generalized-hereditary programs is introduced. This class properly contains both the class of hereditary programs and the class of innermost-simple programs. The class of generalized-hereditary programs allow defined functions only in the heads and assume that all the defined functions are increasing. In contrast, the class of hierarchical programs allow defined functions both in the head as well as in the body. The class of generalized-hereditary programs is a proper subclass of hierarchical programs.

The occurrences of defined functions in a clause of a functional logic program has the following relation with the atoms having predicate symbols corresponding to the defined symbols in the clauses of a logic program obtained through the flattening operation. Let P be a functional logic program and P' be the logic programming obtained from P by eliminating the defined symbols through the flattening operation. The atoms with predicate symbols corresponding to the defined symbols in the head of a clause in P occur only at the end of a clause in P'. The atoms with predicate symbols corresponding to the defined symbols in the body of a clause in P occur anywhere in the body of a clause in P'. This is illustrated by the following two examples.

Example 15. Consider the following functional logic program

```
app([ ], Ys) = Ys
app([X|Xs], Ys) = [X|app(Xs, Ys)]

reverse([ ], [ ]) ←
reverse([X|Xs], app(Ys, [X])) ← reverse(Xs, Ys)
```

and the logic program obtained through the flattening operation of [8]

```
reverse([ ], [ ]) ←
reverse([X|Xs], Z) ← reverse(Xs, Ys), append(Ys, [X], Z)
```

```
append([ ],Y,Y) ←
append([X|Xs],Y,[X|Zs]) ← append(Xs,Y,Zs)
```

Note that the append atoms only occur at the end of clauses in the derived logic program as the defined function app occurs only in the heads in the functional logic program.

Example 16. Consider the following functional logic program

```
mod(X,Y) = X                          if X<Y and Y>0
mod(X,Y) = mod(subtract(X,Y),Y)       if X≥Y and Y>0

subtract(X,0) = X
subtract(s(X),s(Y)) = subtract(X,Y)

gcd(X,0,X) ← X>0
gcd(X,Y,Z) ← gcd(Y,mod(X,Y),Z)
```

and the logic program obtained through the flattening operation.

```
gcd(X,0,X) ← X>0
gcd(X,Y,Z) ← mod(X,Y,Y'),gcd(Y,Y',Z)

mod(X,Y,X) ← X<Y,Y>0
mod(X,Y,Z) ← X≥Y, subtract(X,Y,Y'), mod(Y',Y,Z)

subtract(X,0,X) ←
subtract(s(X),s(Y),Z) ← subtract(X,Y,Z)
```

Note that the mod atom occurs in front of the gcd atom as the defined function mod occurs in the body in the functional logic program.

To summarize, the results of the present paper allow the already known predicates anywhere in the body, whereas the results of [16, 32] allow the already known predicates only at the end of the new clauses. This is a significant improvement from a programming point of view. The program for gcd given in Example 9 belongs to the class of hierarchical programs but not generalized-hereditary programs.

For the sake of completeness, we discuss an example which is beyond the scope of our results (as well as [21, 22, 2, 32, 16]). A program for merge-sort with a background knowledge about merge cannot be handled by our results, as the defined symbol merge does not have polynomial dependency property (unlike app). For any ground list L of length n there are 2^n pairs of ground lists satisfying $\mathscr{B} \models \mathtt{merge}(L_1, L_2) = L$.

6.2. *Determinate programs*

In the recent years, there has been a significant amount of research (5–7, 15) on the learnability of determinate logic programs. The class of determinate programs is closely

related our class of hierarchical programs. We now define the class of determinate programs.

Definition 21. Consider a clause $H \leftarrow B_1, \ldots, B_n$. A literal B_i is *determinate* if and only if each of its variables that does not occur in preceding literals has only one possible binding given the bindings of its variables that occur in the preceding literals. The clause $H \leftarrow B_1, \ldots, B_n$ is determinate if and only if each of its literals is determinate.

Definition 22. Consider a clause $H \leftarrow B_1, \ldots, B_n$. Variables occurring in H have *depth zero*. The depth of a variable X which occurs first in B_i is $d+1$ if d is the maximum depth of any other variable in B_i that also occurs in $H, B_1, \ldots, B_{i-1}$.

Definition 23. A k-clause predicate definition consists of up to k Horn clauses with the same predicate symbol in the head.

The following results are established in [7]

Theorem 3. (1) k-*clause predicate definitions consisting of non-recursive determinate function-free Horn clauses with variables of bounded depth are polynomially PAC-learnable under simple distributions.*
(2) k-*clause* (*possibly recursive*) *predicate definitions consisting of determinate function-free Horn clauses with variables of bounded depth are polynomially PAC-learnable under simple distributions if we restrict the arity of the target predicate to be less than a fixed bound and allow existential and membership queries about the target predicate.*

The function-free restriction in the above theorem is not a very serious problem as any clause containing function symbols can be rewritten in determinate function-free form with the addition of one background knowledge per function symbol [7].

To relate our results with the above results, note that our assumption that background knowledge is presented by a terminating and confluent rewrite system implies that the program defining the already known predicates is determinate. Since our emphasis is to use the predicates already learned as background knowledge in learning new predicates, we are essentially dealing with determinate programs. Our restriction that defined symbols are not nested in hierarchical programs meant that the variables are of depth at most one. In this sense the class of hierarchical programs is a subclass of the class of determinate programs.

However, the advantages of our result over the above result are that (1) we do not need any existential or membership queries, but learn a consistent program from a given sequence of positive and negative examples and (2) do not require simple distributions in contrast to the above result.

In two survey papers [5, 6], Cohen has presented a list of learnability and non-learnability results. In particular, he proves that one depth bounded determinate recur-

sive clause is learnable from equivalence queries (and hence PAC-learnable), and a program consisting of one such recursive clause and one depth bounded determinate nonrecursive clause is learnable from equivalence queries (and hence PAC-learnable) if an additional 'basecase oracle' is given.

Frazier and Page [9] investigate the learnability of several classes of recursive logic programs from examples only (as in our approach). Their main result is that two-clause two-literal programs built from unary predicates, unary functions and constants are polynomially PAC-learnable. Further they show that removing restriction on predicate arity leads to non-learnability (unless a constant bound is placed on predicate arity). Though these two classes are quite restrictive, they contain some non-determinate programs.

Lapointe and Matwin [19] propose an algorithm to learn recursive programs with just one unit clause and just one recursive clause, using the notion of sub-unification. They use logical implication (rather than θ-subsumption used in all other approaches) as generalization principle and argue that it is the best approach to inductive logic programming. However, there is no consensus on this issue in the literature so far. Their algorithm works in two modes (1) to learn recursive clauses with just two literals and (2) to learn left recursive clauses (possibly with more than one literal in the body). At present, their approach cannot deal with clauses having more than one recursive literal or the clauses with non-left recursion.

Idestam-almquist [13] proposes an approach to learn tail recursive programs with at most two clauses but no more than one recursive clause. He uses structural analysis of saturations of the given positive examples in constructing tail recursive clauses.

In contrast to the approaches of [9,19,13] we allow predicate definitions with more than one recursive clauses and more than one recursive literals in the recursive clauses.

7. Concluding remarks

In this paper, we propose a framework for incremental learning of logic programs. In this framework, logic programs defining the already known predicates are used as background knowledge in learning new predicates. The programs learned in this way have nice modular structure. A class of logic programs which can be learned in polynomial-time using this approach is identified and a comparison with the known results is provided.

The algorithm sketched in the proof of the polynomial-time learnability result has left lot of details to the implementation and the bounds given there are very loose. Lots of improvements can be made during the implementation stage. For example, it does not discuss the predicate invention needed while learning more than one predicate from the examples of a single target predicate (see `quick-sort` example). Our proof shows that more than one predicate (a target predicate and the intermediate predicates needed in defining the target predicate by a hierarchical program) can be learned in principle through an enumeration. However, it will be more efficient to use some known predicate invention algorithm for inventing intermediate predicates. Further, some implementation

decisions to restrict the size of constructor terms in the clauses can drastically improve the efficiency of the algorithm even though the class of programs that can be learned becomes a bit small. For example, restricting size of constructor terms in the clauses to 3 allows all the example programs given in previous sections to be learned, but gives a vast improvement in the efficiency.

Acknowledgements

The author would like to thank the anonymous referees for their constructive comments improving the paper in a significant way. He also thanks Luc De Raedt for his timely help of supplying some papers and a copy of his recent book.

References

[1] G. Aguzzi, U. Modigliani, Proving termination of logic programs by transforming them into equivalent term rewriting systems, in: Proc. FST&TCS'93, Lecture Notes in Computer Science, Vol. 761, Springer, Berlin, 1993, pp. 114–124.

[2] S. Arikawa, S. Miyano, A. Shinohara, T. Shinohara, A. Yamamoto, Algorithmic learning theory and elementary formal systems, IEICE Trans. Inform. System. E75-D (1992) 405–414.

[3] A. Blumer, A. Ehrenfeucht, D. Haussler, M.K. Warmuth, Learnability and Vapnik-Chervonenkis dimension, J. Assoc. Comput. Math. 36 (1989) 929–965.

[4] P.G. Bosco, E. Giovannetti, C. Moiso, Narrowing vs. SLD-resolution, Theoret. Comput. Sci. 59 (1988) 3–23.

[5] W.W. Cohen, Pac-learning recursive logic programs; efficient algorithms, J. Artificial Intell. Res. 2 (1995) 501–539.

[6] W.W. Cohen, Pac-learning recursive logic programs; negative results, J. Artificial Intell. Res. 2 (1995) 541–573.

[7] S. Dzeroski, S. Muggleton, S. Russel, PAC-learnability of determinate logic programs, in: Proc. COLT'92, 1992, 128–135.

[8] R. Farrow, T.J. Marlowe, D.M. Yellin, Composable attribute grammars: support for modularity in translator design and implementation, in: Proc. POPL'92, 1992, 223–237.

[9] M. Frazier, C.D. Page, Learnability in inductive logic programming: some results and techniques, in: Proc. AAAI'93, 1993, 93–98.

[10] H. Ganzinger, U. Waldmann, Termination proofs of well-moded logic programs via conditional rewrite systems, in: Proc. CTRS'92, Lecture Notes in Computer Science, Vol. 656, Springer, Berlin, 1992, pp. 216–222.

[11] M. Hanus, The integration of functions into logic programming: a survey, J. Logic Programming 19/20 (1994) 583–628.

[12] J.-M. Hullot, Canonical forms and unification, in: Proc. CADE'80, Lecture Notes in Computer Science, Vol. 87, Springer, Berlin, 1980, pp. 318–334.

[13] P. Idestam-almquist, Efficient induction of recursive definitions by structural analysis of saturations, in: L. De Raedt (Ed.), Advances in Inductive Logic Programming, IOS Press, 1996, pp. 192–205.

[14] K. Ito, A. Yamamoto, Polynomial-time MAT learning of multilinear logic programs, in: Proc. ALT'92, Lecture Notes in Computer Science, Vol. 743, Springer, Berlin, 1993, pp. 63–74.

[15] J.-U. Kietz, S. Dzeroski, Inductive logic programming and learnability, SIGART Bull. 5 (1994) 22–32.

[16] M.R.K. Krishna Rao, Incremental Learning of Logic Programs, in: Proc. ALT'95, Lecture Notes in Computer Science, Vol. 997, Springer, Berlin, 1995, pp. 95–109.

[17] M.R.K. Krishna Rao, D. Kapur, R.K. Shyamasundar, A Transformational methodology for proving termination of logic programs, in: Proc. CSL'91, Lecture Notes in Computer Science, Vol. 626, Springer, Berlin, 1992, pp. 213–226.

[18] M.R.K. Krishna Rao, D. Kapur, R.K. Shyamasundar, Proving termination of GHC programs, in: Proc. ICLP'93, 1993, pp. 720–736.
[19] S. Lapointe, S. Matwin, Sub-unification: a tool for efficient induction of recursive programs, in: Proc. ML'92, 1992, pp. 273–281.
[20] J.W. Lloyd, Foundations of Logic Programming, Springer, Berlin, 1987.
[21] S. Miyano, A. Shinohara, T. Shinohara, Which classes of elementary formal systems are polynomial-time learnable? in: Proc. ALT'91, 1991, pp. 139–150.
[22] S. Miyano, A. Shinohara, T. Shinohara, Learning elementary formal systems and an application to discovering motifs in proteins, Tech. Report RIFIS-TR-CS-37, RIFIS, Kyushu University, 1993.
[23] S. Muggleton, L. De Raedt, Inductive logic programming: theory and methods, J. Logic Programming. 19/20 (1994) pp. 629–679.
[24] B.K. Natarajan, Machine Learning: A Theoretical Approach, Morgan-Kaufmann, Los Altos, MA, 1991.
[25] L. Plümer, Termination proofs for logic programs, Ph. D. Thesis, University of Dortmund, Also appears as Lecture Notes in Computer Science, Vol. 446, Springer, Berlin, 1990.
[26] Y. Sakakibara, Inductive inference of logic programs based on algebraic semantics, New Gen. Comp. 7 (1990) 365–380.
[27] E. Shapiro, Inductive inference of theories from facts, Tech. Report, Yale Univ., 1981.
[28] E. Shapiro, Algorithmic Program Debugging, MIT Press, Cambridge, MA, 1983.
[29] R.K. Shyamasundar, M.R.K. Krishna Rao, D. Kapur, Rewriting concepts in the study of termination of logic Programs, in: K. Broda (Ed.) Proc. ALPUK'92 Conf. Workshops in Computing series, Springer, Berlin, 1992.
[30] J.D. Ullman, A. van Gelder, Efficient tests for top-Down termination of logical rules, JACM 35 (1988) 345–373.
[31] L.G. Valiant, A theory of learnable, Comm. CACM 27 (1984) 1134–1142.
[32] A. Yamamoto, Generalized unification as background knowledge in learning logic programs, in: Proc. ALT'93, Lecture Notes in Computer Science, Vol. 744, Springer, Berlin, 1993, pp. 111–122. Revised version as Learning logic programs using definite equality theories as background knowledge, IEICE Trans. Inform. System. E78-D (May 1995) 539–544.