Learning Recursive Prolog Programs with Local Variables from Examples

M. R. K. Krishna Rao KRISHNA@CCSE.KFUPM.EDU.SA
Information and Computer Science Department King Fahd University of Petroleum and Minerals, Dhahran 31261, Saudi Arabia.

Abstract

Logic programs with elegant and simple declarative semantics have become very common in many areas of artificial intelligence such as knowledge acquisition, knowledge representation and common sense and legal reasoning. For example, in Human GENOME project, logic programs are used in the analysis of amino acid sequences, protein structure and drug design etc. In this paper, we investigate the problem of learning logic (Prolog) programs from examples and present an inference algorithm for a class of programs. This class of programs (called one-recursive programs) is based on the divide-and-conquer approach and mode/type annotations. Our class is very rich and includes many programs from Sterling and Shapiro's book [33] including `append, merge, split, delete, insert, insertion-sort, preorder` and `inorder` traversal of binary trees, polynomial recognition, derivatives, sum of a list of natural numbers etc., whereas earlier results can only deal with very simple programs without local variables and at most two clauses and one predicate [4].

1. Introduction

The theory of inductive inference attempts to understand the all pervasive phenomena of learning from examples and counterexamples. Starting from the influential works of Gold [12] and Blum and Blum [5], a lot of effort has gone into the development of a rich theory about inductive inference and the classes of concepts which can be learned from both positive (examples) and negative data (counterexamples) and the classes of concepts which can be learned from positive data alone. The study of inferability from positive data alone is important because negative examples are hard to obtain in practice.

Logic programs with simple and elegant declarative semantics can be used as representations of the concepts to be learned. In fact, the problem of learning logic programs from examples has attracted a lot of attention (a.o. [3,4,7,8,10,11,13-16,18-20,22-25,28,29,35]) starting with the seminal work of Shapiro [30, 31] and many techniques and systems for learning logic programs are developed and used in many applications. See [24] for a recent survey.

The existing literature mainly concerns with either nonrecursive programs or recursive programs without local variables, usually with a further restriction that programs contain a unit clause and at most one recursive clause with just one atom in the body. It is a well-known fact that local variables in logic programs play an important role in *sideways information passage*. However, presence of local variables pose a few difficulties in analyzing and learning programs. Moding annotations and linear inequalities have been successfully applied in the literature (cf. [34, 27, 17, 2]) to tame these difficulties in analyzing logic programs with local variables (in particular, termination and occur-check aspects). In this paper, we demonstrate that moding/typing annotations and linear inequalities are useful in learning logic programs as well.

As established by many authors in the literature, learning recursive logic programs, even with the above restrictions, is a very difficult problem. We approach this problem from a programming methodology angle and propose an algorithm to learn a class of Prolog programs, that use divide-and-conquer methodology. Our endeavour is to develop an inference algorithm that learns a very natural class of programs so that it will be quite useful in practice. We measure the naturality of a class of programs in terms of the number of programs it covers from a standard Prolog book such as [33]. We use the inference criterion proposed by An-

gluin [1]: consistent and conservative identification in the limit from positive data with polynomial time in updating conjectures. That is, the program guessed by the algorithm is always consistent with the examples read so far and changes its guess only when the most recently read example is not consistent with the current guess and it updates its guess in polynomial time in the size of the current sample of examples read so far.

2. Preliminaries

We assume that the reader is familiar with the basic terminology of logic programming and inductive inference and use the standard terminology from [21, 24, 12]. We are primarily interested in programs operating on the following recursive types used in Sterling and Shapiro [33].[1]

```
Nat ::= 0 | s(Nat)
List ::= [ ] | [item | List]
ListNat ::= [ ] | [Nat | ListNat]
Btree ::= void | tree(Btree, item, Btree)
```

Definition 1 A term t is a *generic expression* for type T if for every $s \in T$ disjoint with t the following property holds: *if s unifies with t then s is an instance of t.*

For example, a variable is a generic expression for every type T, and [ ], [X], [H|T], [X, Y|Z], $\cdots$ are generic expressions for the type List. Note that a generic expression for type T need not be a member of T — e.g., term $f(X)$ is a generic expression for the type List.

Notation:

1. We call the terms 0, [ ] and void the constants of their respective types, and call the subterms T_1 and T_2 *recursive subterms of* term (or generic-expression) of the form $\texttt{tree}(T_1, X, T_2)$ of type Btree. Similarly, L is the recursive subterm of term (or generic-expression) of the form [H|L] of type List.

2. The generic-expression 0 (resp. [ ] and void) is called the first generic-expression of type Nat (resp. List and Btree). The generic-expression s(X) (resp. [H|L] and $\texttt{tree}(T_1, X, T_2)$), which generalizes all the other terms of type Nat (resp. List and Btree) is called the second generic-expression of type Nat (resp. List and Btree).

[1] Though we only consider Nat, List, ListNat and Btree in the following, any other recursive type can be handled appropriately.

Remark: Note that the first and second generic-expressions of a given recursive type are unique upto variable renaming.

Definition 2 For a term t, the *parametric size* $[t]$ of t is defined recursively as follows:

- if t is a variable X then $[t]$ is a linear expression X,
- if t is the empty list [ ] or the natural number 0 or the empty tree void then $[t]$ is zero,
- if $t = f(t_1, \ldots, t_n)$ and $f \in \Sigma - \{0, [\,], \texttt{void}\}$ then $[t]$ is a linear expression $1 + [t_1] + \cdots + [t_n]$.

The parametric size of a sequence $\mathbf{t}$ of terms $t_1, \cdots, t_n$ is the sum $[t_1] + \cdots + [t_n]$.

The *size* of a term t, denoted by $|t|$, is defined as $[t]\theta$, where θ substitutes 1 for each variable. The *size* of an atom $p(t_1, \cdots, t_n)$ is the sum of the sizes of terms $t_1, \cdots, t_n$.

Example 1 *The parametric sizes of terms* [ ], [X], [a] *and* [a, b, c] *are* 0, $X + 1$, 2, *and* 6 *respectively. Their sizes are* 0, 2, 2, *and* 6 *respectively.*

Remark: In general, the size of a list (or binary tree) with n elements is $2n$. This is similar to the measures used in the termination analysis of logic programs by Plümer [27] in the sense that size of a term is proportional to its contents.

3. Linearly-moded programs

Using moding annotations and linear predicate inequalities, Krishna Rao [18] introduced the following class of programs and proved a theoretical result that this class is inferable from positive examples alone.

Definition 3 A *mode* m of an n-ary predicate p is a function from $\{1, \cdots, n\}$ to the set $\{in, out\}$. The sets $in(p) = \{j \mid m(j) = in\}$ and $out(p) = \{j \mid m(j) = out\}$ are the sets of input and output positions of p respectively.

A moded program is a logic program with each predicate having a unique mode associated with it. In the following, $p(\mathbf{s}; \mathbf{t})$ denotes an atom with input terms $\mathbf{s}$ and output terms $\mathbf{t}$.

Definition 4 Let P be a moded program and I be a mapping from the set of predicates occurring in P to sets of input positions satisfying $I(p) \subseteq in(p)$ for each predicate p in P. For an atom $A = p(\mathbf{s}; \mathbf{t})$, the linear inequality

$$\sum_{i \in I(p)} [s_i] \geq \sum_{j \in out(p)} [t_j] \qquad (1)$$

is denoted by $LI(A, I)$.

Definition 5 A moded program P is *linearly-moded w.r.t. a mapping* I such that $I(p) \subseteq in(p)$ for each predicate p in P, if each clause

$$p_0(\mathbf{s_0};\mathbf{t_0}) \leftarrow p_1(\mathbf{s_1};\mathbf{t_1}),\cdots,p_k(\mathbf{s_k};\mathbf{t_k})$$

$k \geq 0$, in P satisfies the following:

1. $LI(A_1,I),\ldots,LI(A_{j-1},I)$ together imply $[\mathbf{s_0}] \geq [\mathbf{s_j}]$ for each $j \geq 1$, and
2. $LI(A_1,I),\ldots,LI(A_k,I)$ together imply $LI(A_0,I)$,

where A_j is the atom $p_j(\mathbf{s_j};\mathbf{t_j})$ for each $j \geq 0$.
A program P is *linearly-moded* if it is linearly-moded w.r.t. some mapping I.

Example 2 *Consider the following* `reverse` *program.*

```
moding:  app(in,in, out) and rev(in, out).

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

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

This program is linearly-moded *w.r.t. the mapping* $I(\texttt{app}) = in(\texttt{app})$; $I(\texttt{rev}) = in(\texttt{rev})$. *For lack of space, we only prove this for the last clause.* $LI(\texttt{rev(Xs, Ys)}, I)$ *is*

$$Xs \geq Ys, \tag{2}$$

$LI(\texttt{app(Ys, [X], Zs)}, I)$ *is*

$$Ys + 1 + X \geq Zs \tag{3}$$

and $LI(\texttt{rev([X|Xs], Zs)}, I)$ *is*

$$1 + X + Xs \geq Zs. \tag{4}$$

It is easy to see that inequalities 2 and 3 together imply inequality 4 satisfying the requirement 2 of Definition 5. The requirement 1 of Definition 5 holds for atoms `rev(Xs, Ys)` *and* `app(Ys, [X], Zs)` *as follows:* $1 + X + Xs \geq Xs$ *trivially holds for atom* `rev(Xs, Ys)`. *For atom* `app(Ys, [X], Zs)`, *inequality 2 implies* $1 + X + Xs \geq Ys + 1 + X$.

The class of linearly-moded programs is very rich and contains many standard programs such as `split, merge, quick-sort, merge-sort, insertion-sort` and various tree traversal programs.

4. One-Recursive Programs

To facilitate efficient learning of programs, we restrict our attention to a subclass of the class of linearly-moded programs. In particular, we consider well-typed programs [6]. The divide-and-conquer approach and recursive subterms are the two central themes of our class of programs. The predicates defined by these programs are recursive on the leftmost argument. The leftmost argument of each recursive call invoked by a caller is a recursive subterm of the arguments of the caller. In the following, *builtins* is a (possibly empty) sequence of atoms with built-in predicates having no output positions.

Definition 6 (**One-recursive programs**)
A linearly-moded well-typed Prolog program without mutual recursion is *one-recursive* if each clause in it is of the form

$$p(\mathbf{s}_0;\mathbf{t}_0) \leftarrow \mathit{builtins},\ p(\mathbf{s}_1;\mathbf{t}_1),\cdots,p(\mathbf{s}_k;\mathbf{t}_k)$$

or

$$p(\mathbf{s}_0;\mathbf{t}_0) \leftarrow \mathit{builtins}, p(\mathbf{s}_1;\mathbf{t}_1),\cdots,p(\mathbf{s}_k;\mathbf{t}_k), q(\mathbf{s};\mathbf{t})$$

such that (a) $\mathbf{s}_i$ is same as $\mathbf{s}_0$ except that the leftmost term in $\mathbf{s}_i$ is a recursive subterm of the leftmost term in $\mathbf{s}_0$ for each $1 \leq i \leq k$, (b) the terms in $\mathbf{s}_0$ are variables or one of the first two generic-expressions of the asserted types and $|\mathbf{s}_0| \geq |\mathbf{t}_0|$ and (c) the terms in $\mathbf{t}_i,\ i \geq 1$ are distinct variables not occuring in $\mathbf{s}_0$.

It is easy to see that all the above conditions can be checked in linear time by scanning the program once.

Theorem 1 *Whether a well-typed program P is one-recursive or not can be checked in polynomial (over the size of the program) time.*

The following example illustrates the divide-and-conquer nature of one-recursive programs.

Example 3 Consider the following program for preorder traversal of binary trees.

```
mode/type:  preorder(in:Btree,out:List) and
      app (in:List, in:List, out:List)

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

preorder(void, [ ]) ←
preorder(tree(T1, X, T2), [X|L]) ←
      preorder(T1, L1), preorder(T2, L2),
      app(L1, L2, L)
```

It is easy to see that this program is well-typed, linearly-moded and one-recursive.

A typical one-recursive clause
$p(\mathbf{s}_0;\mathbf{t}_0) \leftarrow \mathit{builtins}, p(\mathbf{s}_1;\mathbf{t}_1),\cdots,p(\mathbf{s}_k;\mathbf{t}_k), q(\mathbf{s};\mathbf{t})$
satisfies (1) $|\mathbf{s}\sigma| \geq |\mathbf{t}\sigma|$ for every substitution σ such that $p(\mathbf{s}_0;\mathbf{t}_0)\sigma, p(\mathbf{s}_1;\mathbf{t}_1)\sigma,\cdots,p(\mathbf{s}_k;\mathbf{t}_k)\sigma,\ q(\mathbf{s};\mathbf{t})\sigma$ are atoms in the minimal Herbrand model and (2) $[\mathbf{t}_0] \leq [\mathbf{s}_0,\mathbf{t}_1,\cdots,\mathbf{t}_n,\mathbf{t}]$. These properties form the basis for **Step Aux** in the inference algorithm given below.

Remark: The class of one-recursive programs is different from the class of linear-recursive programs studied in Cohen [7, 8]. Linear-recursive programs allow at most one recursive atom in the body of a clause, whereas one-recursive programs allow more than one recursive atoms in the body of a clause.

5. Algorithm for generating one-recursive programs

In this section, we give an inference algorithm to derive one-recursive programs from positive presentations. We only consider programs satisfying the following conditions: (1) programs are deterministic such that the least Herbrand model of a program do not contain two different atoms $p(\mathbf{s};\mathbf{t_1})$ and $p(\mathbf{s};\mathbf{t_2})$ with the same input terms, (2) heads of no two clauses are same (even after renaming) and (3) non-recursive clauses have only builtin atoms in the body. These conditions are obeyed by almost all the programs given in Sterling and Shapiro [33].

We need the following concepts in describing our algorithm. An atom A is a most specific generalization (or msg) of a set S of atoms if (a) each atom is in S is an instance of A and (b) A is an instance of any other atom B satisfying condition (a). It is well known that msg of S can be computed in polynomial time in the total size of atoms in S [26]. In view of the restrictions placed on the atoms in one-recursive programs, it is some times desirable to have more than one atoms (in a particular form) to cover a set S of atoms.

In the following, we assume that the type of the leftmost argument of the target predicate p has n recursive subterms and our recursive clauses are of the form $p(s,\cdots) \leftarrow builtins,\ p(s_1,\cdots),\cdots,p(s_n,\cdots)$ or $p(s,\cdots) \leftarrow builtins,\ p(s_1,\cdots),\cdots,p(s_n,\cdots),\ q(\cdots)$, where $s_1,\cdots,s_n$ are the recursive subterms of s. Two atoms $Pat_1 \equiv p(u_1,\mathbf{u})$ and $Pat_2 \equiv p(u_2,\mathbf{u})$ are called the first two patterns of the target predicate p if (a) u_1 and u_2 are the first two generic-expressions of the asserted type of the leftmost argument of p and (b) $\mathbf{u}$ is a sequence of distinct variables.

Procedure **Infer-one-recursive**;
begin $P := \phi$; $S := \phi$;
Read examples into S until it contains an atom whose leftmost argument has instances of the second generic-expression as recursive subterms, and all the atoms with recursive subterms of this argument as leftmost arguments. That is, if the asserted type of the leftmost position of the target predicate is `List`, read the examples into S until S contains an atom with a list L of at least two elements in the first argument and all the atoms which have sublists of L as first argument.
If an example $p(\mathbf{s};\mathbf{t})$ with $|\mathbf{s}| < |\mathbf{t}|$ is encountered, exit with error message *no linearly-moded program.*
repeat
  Read example $A \equiv p(\mathbf{s};\mathbf{t})$ into S;
  if $|\mathbf{s}| < |\mathbf{t}|$ **then** exit with error(*no LM program*);
  if A is inconsistent with P **then** $P :=$ **Generate**(S);
  if $P =$ false **then** exit with error(*no LM program*)
forever
end;

We say a ground atom $p(\mathbf{s};\mathbf{t})$ is incompatible with a clause $p(\mathbf{u};\mathbf{v}) \leftarrow builtins$ if there is a substitution σ such that (1) *builtins* hold for substitution σ, (2) $\mathbf{s} \equiv \mathbf{u}\sigma$ and (3) $\mathbf{t} \not\equiv \mathbf{v}\sigma$.

Function **Generate**(S);
begin $P := \phi$;
$S1 := \{B \in S \mid$ B is an instance of $Pat_1\}$;
$S2 := \{B \in S \mid$ B is an instance of $Pat_2\}$;
Step 1: $P := P \cup$ **Non-rec**($S1$);
Step 2: $P := P \cup$ **Non-rec**($S2$);
Step 3: % Recursive clauses. %
Let $S3$ be the set of atoms in $S2$ which are not covered by the clauses added in **Step 2**;
if $S3 \neq \phi$ **then**
begin
Let *builtin*3 be the sequence of builtin-atoms complementing the builtin-atoms of the clauses added in Step 2;
Compute **the** msg $p(\mathbf{s}_0;\mathbf{t}_0)$ of $S3$;
Consider the following one-recursive clause:
  $p(\mathbf{s}_0;\mathbf{t}_0) \leftarrow builtin3, p(\mathbf{s}_1;\mathbf{t}_1),\cdots,p(\mathbf{s}_n;\mathbf{t}_n)$;
Let T be the set of tuples $\{\langle \mathbf{s}_0\sigma,\mathbf{t}_1\sigma,\cdots,\mathbf{t}_n\sigma,\mathbf{t}_0\sigma\rangle$ such that $p(\mathbf{s}_0;\mathbf{t}_0)\sigma \in S3$ and $p(\mathbf{s}_i;\mathbf{t}_i)\sigma \in S$ for each $1 \le i \le n\}$;
Get a set $T2$ of msg's of the form $\langle \mathbf{s}_0,\mathbf{t}_1,\cdots,\mathbf{t}_n,\mathbf{t}_0\theta\rangle$ covering **all** the tuples in T such that
  (a) $[\mathbf{t}_0\theta] \le [\mathbf{s}_0,\mathbf{t}_1,\cdots,\mathbf{t}_n]$ and
  (b) $|\mathbf{s}_0| \ge |\mathbf{t}_0\theta|$;
if $T2$ is a singleton set **then** $P := P \cup \{C\}$ where C is $p(\mathbf{s}_0;\mathbf{t}_0\theta) \leftarrow builtin3, p(\mathbf{s}_1;\mathbf{t}_1),\cdots,p(\mathbf{s}_n;\mathbf{t}_n)\}$
elsif $|T2| = m > 1$ **then** form m clauses with additional built-in atoms and add them to P
elsif $T2 = \phi$ **then**
begin
Step Aux: % Add auxiliary predicate. %
Let $T3$ be the set of atoms of the form $q(\mathbf{u};\mathbf{v})$ such that
  (1) $|\mathbf{u}\sigma| \ge |\mathbf{v}\sigma|$ for each σ such that $p(\mathbf{s}_0;\mathbf{t}_0)\sigma \in S3$ and $p(\mathbf{s}_i;\mathbf{t}_i)\sigma \in S$ for each $1 \le i \le n$,
  (2) $LI(A_1,I),\ldots,LI(A_n,I)$ together imply $[\mathbf{s}_0] \ge [\mathbf{u}]$ where $A_i \equiv p(\mathbf{s_i};\mathbf{t_i})$ and
  (3) there is a θ such that $[\mathbf{t}_0\theta] \le [\mathbf{s}_0,\mathbf{t}_1,\cdots,\mathbf{t}_n,\mathbf{v}]$ and $|\mathbf{s}_0| \ge |\mathbf{t}_0\theta|$;
Flag := false;
while not Flag and $T3 \neq \phi$ **do**
begin
Pick an atom $A \equiv q(\mathbf{u};\mathbf{v}) \in T3$;
$T3 := T3 - \{q(\mathbf{u};\mathbf{v})\}$;
Let $T4$ the set of atoms $\{A\sigma$ such that $p(\mathbf{s}_0;\mathbf{t}_0)\sigma \in S3$ and $p(\mathbf{s}_i;\mathbf{t}_i)\sigma \in S$ for each $1 \le i \le n\}$;
AuxP := **Generate**($T4$);
if AuxP $\neq$ false **then** Flag := true

end;
if Flag = false **then** Return(false)
else $P := P \cup AuxP \cup \{C1\}$ where $C1$ is
$p(\mathbf{s_0}; \mathbf{t_0}\theta) \leftarrow builtin3,\ p(\mathbf{s}_1; \mathbf{t}_1), \cdots, p(\mathbf{s}_n; \mathbf{t}_n), q(\mathbf{u}; \mathbf{v})$
end;
end;
Return(P)
end Generate;

Function **Non-rec**(S);
begin $P := \phi$;
Get a set of msg's of the form $p(\mathbf{s}; \mathbf{t})$ for S such that $[\mathbf{t}] \leq [\mathbf{s}]$.
for each msg $p(\mathbf{s}; \mathbf{t})$ **do**
if no atom in S is incompatible with unit clause $p(\mathbf{s}; \mathbf{t}) \leftarrow$
then $P := P \cup \{p(\mathbf{s}; \mathbf{t}) \leftarrow\}$
else try to get a clause $p(\mathbf{s}; \mathbf{t}) \leftarrow$ *builtin atoms* without any incompatible atom in S (if possible) and add it to P;
Return(P)
end Non-rec;

It may be noted that the clauses returned by **Non-rec** for input $S1$ cover all the examples in $S1$ for the following reasons: (1) as the leftmost argument of Pat_1 has no recursive arguments, no recursive clauses can be considered, (2) all the atoms in $S1$ are covered by the clauses of the form $p(\mathbf{s}; \mathbf{t}) \leftarrow$ *builtins* and (3) since $S1$ is a part of the positive presentation of a linearly-moded program, $[\mathbf{t}] \leq [\mathbf{s}]$ holds.
However, the clauses returned by **Non-rec** for input $S2$ need not cover all the examples in $S2$ as shown by the following example. In fact, this is expected, as **Non-rec** only generates unit clauses or clauses with just built-in atoms in the body, while most of the problems need recursive clauses.

Example 4 Let us consider inference of a program `del` for deleting all the occurrence of a given element from a list. The relevant mode/type annotation is `del(in:List, in:Item; out:List)`. The patterns to consider are: `del([ ], Y; Zs)` and `del([X|Xs], Y; Zs)`.

Consider the invocation of **Generate** with examples: `del([ ], 1; [ ])`, `del([ ], 2; [ ])`, `del([1], 1; [ ])`, `del([2], 1; [2])`, `del([2,1], 1;, [2])`, `del([1,2], 1; [2])`, `del([1,2,3], 1; [2,3])`, `del([1,2,1], 1; [2])`. From the first pattern and examples `del([ ], 1; [ ])`, `del([ ], 2; [ ])`, we get a unit clause
`del([ ], Y; [ ])` $\leftarrow$.

Consider step 2 now. Only msg's to be considered by **Non-rec**(S2) are `del([X|Xs], X; Xs)`, `del([X|Xs], X; [X|Xs])`, `del([X|Xs], Y; Xs)` and `del([X|Xs], Y; [Y|Xs])`. There are incompatible examples with each of the unit clauses suggested by these msg's and no sequence of built-in atoms help. Hence step 2 does not generate any clause and $S3 = S2$.

Consider step 3 now. Unlike step 2, step 3 considers the unique msg of $S3$ without any restriction. That msg is `del([X|Xs], Y; Zs)` and the considered recursive clause is `del([X|Xs], Y; Zs)` $\leftarrow$ `del(Xs, Y; Z1s)`. The set of tuples T is $\{\langle[1], 1, [\ ], [\ ]\rangle, \langle[2], 1, [\ ], [2]\rangle, \langle[2, 1], 1, [\ ], [2]\rangle, \langle[1, 2], 1, [2], [2]\rangle\rangle, \langle[1, 2, 3], 1, [2, 3], [2, 3]\rangle, \langle[1, 2, 1], 1, [2], [2]\rangle\}$. Now, $T2$ contains 2 msg's $\langle[X|Xs], X, Z1s, Z1s\rangle$ and $\langle[X|Xs], Y, Z1s, [X|Z1s]\rangle$ and we get the following two recursive clauses after adding apprpriate built-in atoms.
`del([X|Xs], X; Z1s)` $\leftarrow$ `del(Xs, X; Z1s)`
`del([X|Xs], Y; [X|Z1s])` $\leftarrow$ `X` $\neq$ `Y, del(Xs, Y; Z1s)`
and inference algorithm does not invoke **Generate** hereafter as this program is consistent with each example in any positive presentation of `del`.

The following example illustrates the addition of an **auxiliary** predicate by **Generate**.

Example 5 Let us consider inference of a program for `reverse` with mode/type annotations `rev(in : List; out : List)`. The two patterns to consider are `rev([ ]; Ys)` and `rev([X|Xs]; Ys)`.

Consider the invocation of **Generate** with examples: `rev([]; [])`, `rev([a]; [a])`, `rev([b]; [b])`, `rev([a, a]; [a, a])`, `rev([a, b]; [b, a])`. Step 1 generates the unit clause `rev([ ]; [ ])` $\leftarrow$ from the first example and step 2 does not add any clauses as in the above Example.

Step 3 computes the msg, `rev([X|Xs]; [Y|Ys])` of S3 and considers the following one-recursive clause: `rev([X|Xs]; [Y|Ys])` $\leftarrow$ `p(Xs; Zs)`. The set of tuples T is $\{\langle[a], [], [a]\rangle, \langle[b], [], [b]\rangle, \langle[a, a], [a], [a, a]\rangle, \langle[a, b], [b], [b, a]\rangle\}$. There is no set $T2$ of msg's covering all the tuples in T to relate the output terms `[Y|Ys]` and `Zs` and hence Step Aux is executed.

Conditions 1, 2 and 3 force us to consider `q(Zs, [X]; [Y|Ys])`. In particular, condition 1 forces us to use `[X]` rather than `X`. Now the examples for Auxiliary predicate `q` are $T4 = \{$`q([], [a]; [a])`, `q([], [b]; [b])`, `q([a], [a]; [a, a])`, `q([b], [a]; [b, a])`$\}$. From these examples, **Generate**(T4) generates the clauses:
`q([], Ys; Ys)` $\leftarrow$
`q([X|Xs], Ys; [X|Zs])` $\leftarrow$ `q(Xs, Ys; Zs)`
which are nothing but the clauses of append. The recursive clause added for `rev` is
`rev([X|Xs]; [Y|Ys])` $\leftarrow$ `rev(Xs; Zs), q(Zs, [X]; [Y|Ys])`.
In the post processing, this clause will be rewritten to
`rev([X|Xs]; Z)` $\leftarrow$ `rev(Xs; Zs), q(Zs, [X]; Z)`
replacing the term `[Y|Ys]` by `Z` in both the head and body.

The following example is to illustrate that the algorithm learns predicates without any output position as well.

Example 6 *The algorithm considers two patterns* `list([ ])` *and* `list([H|L])` *and generates the following two clauses*
`list([ ])` $\leftarrow$
`list([H|L])` $\leftarrow$ `list(L)`
in learning a predicate `list` *which checks whether a given term is a list or not.*

The following theorem establishes correctness of our algorithm.

Theorem 2 *The above procedure* **Infer-one-recursive**

1. *only generates one-recursive programs which are consistent with the examples read so far (consistent),*

2. *changes its guess only when the most recently read example is not consistent with the current guess (conservative) and*

3. *updates its guess in polynomial time in the size of the current sample of examples read so far (polynomial time updates).*

In view of the notorious difficulty in learning recursive clauses mentioned often in the literature, we explain the main reasons for polynomial time complexity of our algorithm. After reading each example, the algorithm checks whether this new example is consistent with the current program. This consistency check can be done in polynomial time as (1) the leftmost argument of a recursive call is a **proper subterm** of the leftmost argument of the caller, (2) the sum of the sizes of the leftmost arguments of all the recursive calls (in the body of the clause) is at most the size of the leftmost argument of the caller (head of the clause) and (3) the sum of the sizes of input terms of the auxiliary predicate is bounded by the sum of the sizes of input terms of the head. In fact, the sum of the sizes of input terms of atoms in any SLD-derivation of a linear-moded program-query pair is bounded by the sum of the sizes of input terms of the query. Further, by enforcing the discipline that the leftmost arguments of all the recursive atoms in the body are recursive subterms of the leftmost argument of the body and the terms in the clauses are either variables, constants or the first two generic-expressions of the annotated types, we drastically reduce the search space for recursive clauses. This is in sharp contrast to the fact that most of the learning algorithms in the literature spend a lot of time in searching for a suitable recursive clause. The above discipline is encouraged in the programming methodologies advocated by Deville [9] and Sterling and Shapiro [33]. Only notable exception is the `even` program for checking whether a given natural number is even or not, which has a clause `even(s(s(X))` $\leftarrow$ `even(X)` with a term `s(s(X))` that is not among the first two generic-expressions of the type `Nat`. We can relax our restriction to cover this program by allowing terms of depth more than 2, but then the algorithm will become a bit inefficient. These decisions should be postponed to the implementation time.

6. Conclusion

In this paper, we approach the problem of learning logic programs from a programming methodology point of view and propose an algorithm to learn a class of Prolog programs, that use divide-and-conquer methodology. This class of programs is very natural and rich and contains many programs from chapter 3 (on recursive programs) of Sterling and Shapiro's standard book on Prolog [33]. This indicates that our algorithm will be successful in practical situations as the underlying class of programs is very natural.

We believe that our results can be extended in the following two directions: (1) to consider predicates that have more than one recursive arguments (we call such programs, k-recursive programs) and (2) to cover the programs which uses divide-and-conquer approach but splits the input using a specific (to that data type) splitting algorithm rather than the splitting suggested by the recursive structure of the data type. For example, splitting a list into two lists of almost equal length. This can be done when we are looking for learning algorithms that work on a particular (fixed) data type. Further investigations are needed in these directions.

References

[1] D. Angluin (1980), *Inductive inference of formal languages from positive data*, Information and Control **45**, pp. 117-135.

[2] K.R. Apt and A. Pellegrini (1992), *Why the occur-check is not a problem*, Proc. of PLILP'92, LNCS **681**, pp. 69-86.

[3] H. Arimura and T. Shinohara (1994), *Inductive inference of Prolog programs with linear data dependency from positive data*, Proc. Information Modelling and Knowledge Bases V, pp. 365-375, IOS press.

[4] H. Arimura, H. Ishizaka and T. Shinohara (1992), *Polynomial time inference of a subclass of context-free transformations*, Proc. Computational Learning Theory, COLT'92, pp. 136-143.

[5] L. Blum and M. Blum (1975), *Towards a mathematical theory of inductive inference*, Information and Control **28**, pp. 125-155.

[6] F. Bronsard, T.K. Lakshman and U.S. Reddy (1992), *A framework of directionality for proving termination of logic programs*, Proc. Joint Intl. Conf. and Symp. on Logic Prog., JICSLP'92, pp. 321-335

[7] W.W. Cohen (1995a), *Pac-learning recursive logic programs: efficient algorithms*, Journal of Artificial Intelligence Research **2**, pp. 501-539.

[8] W.W. Cohen (1995b), *Pac-learning recursive logic programs: negative results*, Journal of Artificial Intelligence Research **2**, pp. 541-573.

[9] Y. Deville (1990), *Logic Programming: Systematic Program Development*, Addison Wesley.

[10] S. Dzeroski, S. Muggleton and S. Russel (1992), *PAC-learnability of determinate logic programs*, Proc. of COLT'92, pp. 128-135.

[11] M. Frazier and C.D. Page (1993), *Learnability in inductive logic programming: some results and techniques*, Proc. of AAAI'93, pp. 93-98.

[12] E.M. Gold (1967), *Language identification in the limit*, Information and Control **10**, pp. 447-474.

[13] P. Idestam-Almquist (1993), *Generalization under Implication by Recursive Anti-unification*, Proc. of ICML'93.

[14] P. Idestam-Almquist (1996), *Efficient induction of recursive definitions by structural analysis of saturations*, pp. 192-205 in L. De Raedt (ed.), *Advances in inductive logic programming*, IOS Press.

[15] J.-U. Kietz (1993), *A Comparative Study of Structural Most Specific Generalizations Used in Machine Learning*, Proc. Workshop on Inductive Logic Programming, ILP'93, pp. 149-164.

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

[17] M.R.K. Krishna Rao, D. Kapur and R.K. Shyamasundar (1997), *A Transformational methodology for proving termination of logic programs*, The Journal of Logic Programming **34**, pp. 1-41.

[18] M.R.K. Krishna Rao (2001), *Some classes of Prolog programs inferable from positive data*, *Theoretical Computer Science* **241**, pp. 211-234.

[19] S. Lapointe and S. Matwin (1992), *Sub-unification: a tool for efficient induction of recursive programs*, Proc. of ICML'92, pp. 273-281.

[20] N. Lavrac, S. Dzeroski and M. Grobelnik (1991), *Learning nonrecursive definitions of relations with LINUS*, Proc. European working session on learning, pp. 265-81, Springer-Verlag.

[21] J. W. Lloyd (1987), *Foundations of Logic Programming*, Springer-Verlag.

[22] S. Miyano, A. Shinohara and T. Shinohara (1991), *Which classes of elementary formal systems are polynomial-time learnable?*, Proc. of ALT'91, pp. 139-150.

[23] S. Miyano, A. Shinohara and T. Shinohara (1993), *Learning elementary formal systems and an application to discovering motifs in proteins*, Tech. Rep. RIFIS-TR-CS-37, Kyushu University.

[24] S. Muggleton and L. De Raedt (1994), *Inductive logic programming: theory and methods*, J. Logic Prog. **19/20**, pp. 629-679.

[25] S. Muggleton (1995), *Inverting entailment and Progol*, in *Machine Intelligence 14*, pp. 133-188.

[26] G. Plotkin (1970), *A note on inductive generalization*, in Meltzer and Mitchie, *Machine Intelligence 5*, pp. 153-163.

[27] L. Plümer (1990), *Termination proofs for logic programs*, Ph. D. thesis, University of Dortmund, Also appears as Lecture Notes in Computer Science **446**, Springer-Verlag.

[28] J.R. Quinlan and R.M. Cameron-Jones (1995), *Induction of logic programs: foil and related systems*, New Generation Computing **13**, pp. 287-312.

[29] Y. Sakakibara (1990), *Inductive inference of logic programs based on algebraic semantics*, New Generation Computing **7**, pp. 365-380.

[30] E. Shapiro (1981), *Inductive inference of theories from facts*, Tech. Rep., Yale Univ.

[31] E. Shapiro (1983), *Algorithmic Program Debugging*, MIT Press.

[32] T. Shinohara (1991), *Inductive inference of monotonic formal systems from positive data*, New Generation Computing **8**, pp. 371-384.

[33] L. Sterling and E. Shapiro (1994), *The Art of Prolog*, MIT Press.

[34] J.D. Ullman and A. van Gelder (1988), *Efficient tests for top-Down termination of logical rules*, JACM **35**, pp. 345-373.

[35] A. Yamamoto (1993), *Generalized unification as background knowledge in learning logic programs*, Proc. of ALT'93, LNCS **744**, pp. 111-122.