Language Learning by Inverse Resolution on Elementary Formal Systems

Zeng, Chao
Interdisciplinary Graduate School of Engineering Sciences, Kyushu University

Arikawa, Setsuo
Research Institute of Fundamental Information Science Kyushu University

出版情報：RIFIS Technical Report. 79, 1994-01-10. Research Institute of Fundamental Information Science, Kyushu University
バージョン：
権利関係：

RIFIS-TR-CS-79

RIFIS T echnical Report

Language Learning by Inverse Resolution on Elementary Formal Systems

Chao Zeng and Setsuo Arikawa

January 10, 1994

Research Institute of Fundamental Information Science
Kyushu University 33
Fukuoka 812, Japan
E-mail: zeng@rifis.sci.kyushu-u.ac.jp Phone: 092-641-1101 ex.2329

Language Learning by Inverse Resolution on Elementary Formal Systems

Chao Zeng* **and Setsuo Arikawa†**

Abstract

The inverse resolution is a procedure to produce new clauses by applying the resolution principle in the opposite direction. It has mainly been studied in terms of logic. The elementary formal systems(EFS for short) invented by Smullyan proved suitable for a unifying framework for language learning.

In this paper we consider the problem of how to construct an EFS system from some given examples in the desired language. We first discuss a realization of the inverse resolution in the framework of EFS for language learning from positive examples. We also consider an efficient learning procedure for an EFS language class which is a subclass of regular languages.

1 Introduction

Inductive learning is mainly studied in the context of logics and formal languages. A lot of learning methods have been discovered. One of them is proposed by Muggleton[12][13], which uses the mechanism of inverting the resolution procedure in logic programming to construct clauses from given examples and some background knowledge. Many practical methods have been studied based on the inverse resolution(Muggleton[12][13], Ling[10][11], Wirth[19][20]), and they have received much attention as more practical approaches than those methods in the criterion of identification in the limit by Gold[7]. The inverse resolution is expected to be a promising mechanism in inductive learning as is the resolution principle in logic programming.

A new framework for inductive inference of languages proposed by Arikawa et al. [5][6] named as EFS has been proved useful and powerful in language learning from positive data by Shinohara[18] in the criterion of Gold's identification in the limit.

In this paper we attempt to give another practical methods to EFS language learning based on the idea of Muggleton's inverting resolution. The main difference between logic programs and EFSs is in the definition of term; in logic programs,

*Interdisciplinary Graduate School of Engineering Sciences, Kyushu University 39, Kasuga 816, Japan E-mail: zeng@rifis.sci.kyushu-u.ac.jp

†Research Institute of Fundamental Information Sciences, Kyushu University 33, Fukuoka 812, Japan E-mail: arikawa@rifis.sci.kyushu-u.ac.jp

the terms are defined as the combinations of function symbols with variables and constants with some other auxiliary symbols such as parentheses, while in EFSs, terms are just defined as patterns, that is, elements in $(\Sigma \cup X)^+$. From this difference, a lot of difficults and intractabilities make their appearance. As pointed out by Shinohara[18], in EFS framework the given examples are just for a special predicate and any other information should not be given for intermediate predicates that are also needed for constructing the EFS. Therefore, the construction of EFS is more difficult than that of logic programs.

In this paper we also discuss the problem which may arise in introducing inverse resolution to EFS framework, and attempt to give its solutions. By solving the problem, we give the definition of EFS counterpart of inverse substitution and a non-deterministic algorithm to compute EFS languages. Lastly, we consider an efficient learning method for a special class of EFS languages. We give an algorithm to learn languages in the class.

2 Preliminaries

2.1 Elementary Formal Systems

Suppose Σ, X and Π be mutually disjoint sets. Let Σ be a finite set, X a set of variables, and Π a set of predicate symbols. We call Σ as alphabet, denote its elements by a, b, c, ..., and X as variable, denote its elements by x, y, z, x_1, y_1, ... and each element of Π as predicate symbol, denote by p, q, p_1, p_2, ..., where each of them has an arity. Let Σ^* be the set of all words over Σ, and Σ^+ be the set of all nonempty words. Let S be an EFS that is defined below.

We suppose all of the predicate symbols with arity one and get the following definitions. For general case see Arikawa[6].

Definition 1 *A term of S is an element of* $(\Sigma \cup X)^+$. *Each term is denoted by* π, τ, π_1, π_2, ..., τ_1, τ_2, ... *A ground term of S is an element of* Σ^+. *Terms are also called patterns or strings.*

Definition 2 *An atomic formula(or atom for short) of S is an expression of the form* $p(\pi)$, *where p is a predicate symbol in* Π *with arity one and* π *is a term of S. The atom is ground if* π *is ground.*

A definite clause is a clause of the form

$$p \leftarrow p_1, \ldots, p_n \ (n \geq 0).$$

Definition 3 *An elementary formal system(EFS for short) S is a triplet* (Σ, Π, Γ), *where* Γ *is a finite set of definite clauses. The definite clauses in* Γ *are called axioms of S.*

We denote a substitution θ by $\{x_1 := \tau_1, \ldots, x_n := \tau_n\}$, where x_i are mutually distinct variables, and $q(\tau)\theta = q(\tau\theta)$ and

$$(p \leftarrow p_1, \ldots, p_m)\theta = p\theta \leftarrow p_1\theta, \ldots, p_m\theta$$

for an atom $q(\tau)$ and a clause $p \leftarrow p_1, \ldots, p_m$.

Definition 4 *Let $S = (\Sigma, \Pi, \Gamma)$ be an EFS. We define the relation $\Gamma \vdash C$ for a clause C of S inductively as follows:*
(1) *If $\Gamma \ni C$, then $\Gamma \vdash C$.*
(2) *If $\Gamma \vdash C$, then $\Gamma \vdash C\theta$ for any substitution θ.*
(3) *If $\Gamma \vdash p \leftarrow p_1, \ldots, p_{n+1}$ and $\Gamma \vdash p_{n+1}$, then $\Gamma \vdash p \leftarrow p_1, \ldots, p_n$.*
C is provable from Γ if $\Gamma \vdash C$.

Definition 5 *For an EFS S=(Σ, Π, Γ) and $p \in \Pi$, we define*

$$L(S, p) = \{\alpha \in \Sigma^+ \mid \Gamma \vdash p(\alpha)\}.$$

$L(S, p)$ is a language over Σ. A language $L \subseteq \Sigma^+$ is definable by EFS or an EFS language if such an S and a p exist.

Let $\mid \pi \mid$ denotes the length of a term π, and $o(x, \pi)$ denotes the number of all occurrences of a variable x in term π. For an atom $p(\pi)$, we define $\mid p(\pi) \mid = \mid \pi \mid$, and $o(x, p(\pi)) = o(x, \pi)$. A definite clause $p \leftarrow p_1, \ldots, p_n$ is lengh-bounded if

$$\mid p\theta \mid \geq \mid p_1\theta \mid + \ldots + \mid p_n\theta \mid$$

for any substitution θ. An EFS Γ is length-bounded if all definite clauses in Γ are length-bounded.

Let $\nu(p)$ be the set of all variables in an atom p. A definite clause $p \leftarrow p_1, \ldots, p_n$ is variable-bounded if $\nu(p) \supseteq \nu(p_i)(i = 1, \ldots, n)$, and an EFS is variable-bounded if its axioms are all variable-bounded.

Definition 6 *A length-bounded EFS S=(Σ, Π, Γ) is simple if each axiom in Γ is of the form*

$$p(\pi) \leftarrow p_1(x_1), \ldots, p_n(x_n)$$

where $x_1, \ldots, x_n$ are mutually distinct variables.

Definition 7 *A term π is regular if $o(x, \pi) \leq 1$ for any variable x. A simple EFS S=(Σ, Π, Γ) is regular if the term in the head of each definite clause in Γ is regular.*

Lemma 1 [6] *A clause $p \leftarrow p_1, \ldots, p_n$ is length-bounded if and only if*

$$\mid p\theta \mid \geq \mid p_1\theta \mid + \ldots + \mid p_n\theta \mid,$$
$$o(x, p) \geq o(x, p_1) + \ldots + o(x, p_n)$$

for any variable x.

Theorem 1 [18] *For any $n \geq 1$, the class $L^n = \{l(\Gamma, p) \mid \Gamma$ is length-bounded , p is unary, $\sharp\Gamma \leq n$, $L(\Gamma, p) \neq \phi\}$ is inferable from positive data.*

2.2 Some Classes of EFS languages

Definition 8 *A regular EFS* $S=(\Sigma, \Pi, \Gamma)$ *is right-linear (left-linear) if each axiom in* Γ *is one of the following forms:*

$p(\pi) \leftarrow$,

$p(ux) \leftarrow q(x) \qquad (p(xu) \leftarrow q(x))$,

where π *is a regular term and* $u \in \Sigma^+$.

A Regular EFS is called one-sided linear if it is right- or left-linear. Next we give a restricted form of one-sided linear EFS as follows.

Definition 9 *An one-sided linear EFS* $S=(\Sigma, \Pi, \Gamma)$ *is called* Γ*-stratified if for any predicate symbol* p *in* Π *there are no clauses in* Γ *which make a circle about* p *as following.*

$$p(ux) \leftarrow q_1(x), q_1(u_1x_1) \leftarrow q_2(x_1), \ldots, q_{n-1}(u_{n-1}x_{n-1}) \leftarrow q_n(x_{n-1}),$$
$$q_n(u_nx_n) \leftarrow p(x_n).$$

Definition 10 *Let EFS* $S=(\Sigma, \Pi, \Gamma)$ *be a* Γ*-stratified right-linear (left-linear) EFS. Then* S *is called restricted if the following conditions are satisfied.*
(1) *For any two unit clauses*

$$C_1 = p(u) \leftarrow, C_2 = p(v) \leftarrow ,$$

if both C_1 *and* C_2 *are in* Γ *then* u*=*v*, that is, term* u *and* v *are identical.*
(2) *For any two clauses*

$$C_1 = p_1(ux) \leftarrow q_1(x), C_2 = p_1(vy) \leftarrow q_1(y)$$
$$(C_1 = p_1(xu) \leftarrow q_1(x), C_2 = p_1(xy) \leftarrow q_1(y)),$$

if both C_1 *and* C_2 *are in* Γ *then* u*=*v*, that is, term* u *and* v *are identical.*

Definition 11 *An one-side linear EFS* $S=(\Sigma, \Pi, \Gamma)$ *is one-loop definable if there just only one, or no clause with same predicate symbol in head and body exists in* Γ.

We denote the classes of all languages defined by Γ-stratified, restricted, and one-loop definable one-side linear EFSs as Γ-stratified-OSL-EFS, R-OSL-EFS and OLD-OSL-EFS, respectively. We give some examples of these language classes.

Example 1 $L(S,p) = \{a^{3n} \mid n \geq 1\} \in$ *R-OSL-EFS.*

$$\Gamma = \left\{ \begin{array}{l} p(aaax) \leftarrow p(x), \\ p(aaa) \leftarrow \end{array} \right\}$$

Example 2 $L(S,p) = ab\{aba\}^+bba \in$ *R-OSL-EFS.*

$$\Gamma = \left\{ \begin{array}{l} p(abx) \leftarrow q(x), \\ q(abax) \leftarrow q(x), \\ q(abax) \leftarrow r(x), \\ r(bba) \leftarrow \end{array} \right\}$$

Example 3 $L(S,p) = ab(aa+bb)aba^+ \in$ *OLD-OSL-EFS* .

$$\Gamma = \left\{ \begin{array}{l} p(abx) \leftarrow q(x), \\ q(aax) \leftarrow r(x), \\ q(bbx) \leftarrow r(x), \\ r(abx) \leftarrow s(x), \\ s(ax) \leftarrow s(x), \\ s(a) \leftarrow \end{array} \right\}$$

3 Inverse Substitution in EFS

Substitution is essential in Prolog programming. Substitution makes a general clause special. So we can use it to specialize our knowledge to solve more specific problems. Muggleton[13][14] has introduced the opposite concepts that are the inverse substitution and the inverse resolution. Here we attempt to give an EFS version of inverse substitution which is considerably different from the original one in logic programming due to the difference of terms. First we need some preparations for pattern matching.

3.1 Common Strings of a Set of Patterns

A term in EFS is defined as an element in $(\Sigma \cup X)^+$. So when we consider the problem of inference in EFS, we need some methods to get more general patterns from a single pattern or a set of patterns. Many such algorithms have been discovered for getting common patterns in the area of pattern matching.

The problem we are concerned with is described as follows.

PROBLEM 1: Given a set of strings $S = \{p_1, p_2, \ldots, p_m\}$, and suppose $p_i = a_i^1 a_i^2 \ldots a_i^{n_m} (a_i \in \Sigma)$, return the set S_{common} of all their common substrings.

For answering this problem we suppose, without loss of generality, that $p_1 = a_1^1 a_1^2 \ldots a_1^{n_1}$ is the shortest and $p_m = a_m^1 a_m^2 \ldots a_m^{n_1}$ the longest string in S, and SS_{p_1} is the subset of all the substrings of p_1. Then the set S_{common} should be a subset of SS_{p_1}. We know that $\mid SS_{p_1} \mid \leq n_1^2$.

The following Knuth-Morris-Pratt algorithm[1][8] solves the problem that given a pattern p and a text t, answer 'yes' if p occurs as a substring in t, 'no' otherwise, in time O(m+n), where m is the length of p and n is the length of t.

Algorithm **Knuth-Morris-Pratt**
input $p = a_1 a_2 \ldots a_m$, $t = t_1 t_2 \ldots t_n$
output ANSWER
begin
  i:=1; j:=1
  while $i \leq m$ and $j \leq n$ **do**
  begin
    while i>0 and $a_i \neq t_j$ **do** i:=h_i
    i:=i+1; j:=j+1
  end
  if i>m **then return** ANSWER='yes' **else return** ANSWER='no'
end

Using the former Knuth-Morris-Pratt algorithm we can get the following naive algorithm for solving the PROBLEM 1.

Algorithm **Find-Common-String**
input $S = \{p_1, p_2, \ldots, p_m\}$, $p_i = a_i^1 a_i^2 \ldots a_i^{n_i} (i = 1, \ldots, m)$
output S_{common}
begin
  $S_{common} := \phi$
  for i=1 **to** n_1 **do**
  begin
    for h=i to n_1 **do**
    begin
      j:=1
      while $j \leq m$ **do**
        begin
        p:=$a_1^i \ldots a_1^h$; s:=p_j
        call Knuth-Morris-Pratt(p, s)
        if ANSWER='no' **then goto** next:
        j:=j+1
        end
      $S_{common} := S_{common} \cup \{p\}$
next: **end**
  end
end

Analyzing the time complexity of the Find-Common-String algorithm, we have the following theorem.

Theorem 2 *The Find-Common-String algorithm takes $O((s+l)ms^2)$ to solve the* PROBLEM 1, *where m is the number of patterns in S, and s, l are the lengths of the shortest and longest pattern in S, respectively.*

We see that the shortest and longest patterns in the given pattern set control the complexity of the problem. We point out that the the shortest pattern also take an important role in learning EFS languages.

3.2 Inverse Substitutions for Single Ground Term

In inductive learning, generalization mechanism is essential for getting general rules by generalizing examples. As EFS is a framework for studying formal language, the examples given in the learning should be some patterns from the desired language. In order to establish the generalization mechanism in EFS, first we apply the Find-Common-String algorithm to finding inverse substitutions of a single term which is to make the term more general.

In a term there may be many places where one of its subterm appears. For example, consider

$$\pi\text{=aba, } \tau\text{=aababababbbabab.}$$

Obviously, π is a subterm of τ and there are three appearances of term π in τ. Hence when we want to use some other term to replace some particular appearances of π in τ, appointing which of them would be replaced is needed. So giving out concrete address for each appearance becomes to be necessary. We give the definition here. From now on we sometimes denote a term by a string.

Definition 12 *Let π, τ be strings. The address $AD_\tau(\pi)$ of π in τ is defined as the following ordered set*

$$< I_1, I_2, \ldots, I_n >$$

where I_i are positive numbers, and $I_1 < I_2 < \ldots < I_n$. This ordered set means that there are n appearances of π in τ, and I_j is the place of the j-th appearance. If $AD_\tau(\pi) =<>$, it means that there are no appearances of π in τ.

From [8], we know that $I_1, I_2, \ldots, I_n$ can be computed by Knuth-Morris-Pratt algorithm in $O(\mid \pi \mid + \mid \tau \mid)$ time.

Example 4 *For the above example, we have*

$$AD_\tau(\pi) =< 2, 4, 10 >.$$

From the former example, we find out that in some case the address $AD_\tau(\pi)$ is not executable. In another word it is impossible to substitute all π in $AD_\tau(\pi)$ with some other term simultaneously unlike in logic programming. This problem is special to EFS framework. In the above example, the fourth a in τ is also needed in the second and fourth appearances of π . So when we substitute the second and fourth π in τ with some other string, the substitution will fail. We call this kind of non-executable contradiction as intra-contradiction since, it just concerns with a single substring. For dealing with the intra-contradiction we give the following restricted definition of address.

Definition 13 *π, τ are strings. π is a substring of τ. A disjoint address*

$$DAD_\tau(\pi) =< I_{j_1}, I_{j_2}, \ldots, I_{j_m} >$$

of π in τ is a subset of $AD_\tau(\pi)$, which satisfies the following conditions.

(1) $1 \leq j_1 < j_2 < \ldots < j_m \leq n$

(2) $(I_{j_{i+1}} - I_{j_i}) \geq \mid \pi \mid$ *, for every* $i = 2, \ldots, m$,

where $\mid \pi \mid$ is the length of string π.

Obviously, there just only one $AD_\tau(\pi)$ exists for strings π and τ, but many $DAD_\tau(\pi)$s should exist for strings π and τ. An example follows.

Example 5 *Consider π=aba, τ=aababababbbabab. Then we have*

$$AD_\tau(\pi) =< 2, 4, 10 >,$$

and $DAD_\tau(\pi) =< 2 >, < 4 >, < 10 >, < 2, 10 >, < 4, 10 >$ are disjoint addresses. $< 2, 4 >$ is not a disjoint address, since the fourth a in τ is also needed in the second and fourth appearances of π, which is a contradiction.

We have the following proposition which guarantees the executability of substitution based on disjoint addresses.

Proposition 1 *Let π, τ be two strings,and let $DAD_\tau(\pi)$ be a disjoint address. Then any substitution based on $DAD_\tau(\pi)$ is executable without contradiction.*

Proof is trivial.

In the rest of this section, we give the definition of inverse substitution in EFS.

Definition 14 *A substitution θ*

$$\{x_1 := \pi_1, \ldots, x_n := \pi_n\},$$

is a ground substitution if x_i are mutually distinct variables, and $\pi_i \in \Sigma^+$ are ground terms.

Definition 15 *Let t be a term,* vars(t) *be the set of all the variables appeared in t. Then the inverse substitution θ^{-1} of term t is defined as the following finite set*

$$\{(\pi_1, DAD_t(\pi_1)) := x_1, (\pi_2, DAD_t(\pi_2)) := x_2, \ldots, (\pi_n, DAD_t(\pi_n)) := x_n\}(n \geq 0),$$

where π_i are disjoint ground subterm of t, x_i are disjoint variables not in vars(t).

We have succeeded in avoiding the intra-contradiction by introducing the concept of disjoint address. However there still remain another problem. When we consider the executability of inverse substitution, we must deal with more than one substrings. A new contradiction may arise when different substrings use the same characters simultaneously. We call this kind of contradiction an inter-contradiction. We give the following example for explaining the possible inter-contradiction when executing inverse substitution.

Example 6 *Consider*

$$\pi_1 = aba, \pi_2 = abb, t = aababababbbabab,\ DAD_t(\pi_1) =< 4, 10 >, DAD_t(\pi_2) =< 6 >.$$

Then the following inverse substitution is not executable.

$$\theta^{-1} = \{(\pi_1, < 4, 10 >) := x_1, (\pi_2, < 6 >) := x_2\}.$$

The inverse substitution makes an attempt on replacing the fourth aba with x_1 and sixth abb with x_2, but the fourth aba and sixth abb overlap each other with sixth character a in t. So the attempt turns out a failure.

Definition 16 *An inverse substitution*

$$\theta^{-1} = \{(\pi_1, DAD_t(\pi_1)) := x_1, (\pi_2, DAD_t(\pi_2)) := x_2, \ldots, (\pi_n, DAD_t(\pi_n)) := x_n\},$$

is called rational inverse substitution if it does not include any inter-contradictions.

About inter-contradiction, immediately we have the following proposition.

Proposition 2 *The inverse substitution*

$$\theta^{-1} = \{(\pi_1, DAD_t(\pi_1)) := x_1, (\pi_2, DAD_t(\pi_2)) := x_2, \ldots, (\pi_n, DAD_t(\pi_n)) := x_n\},$$

is rational if the substrings $\pi_i (i = 1, ..., n)$ have no common characters.

Here we give a necessary and sufficient condition for an inverse substitution to be executable.

Theorem 3 *Giving inverse substitution*

$$\theta^{-1} = \{(\pi_1, DAD_t(\pi_1)) := x_1, (\pi_2, DAD_t(\pi_2)) := x_2, \ldots, (\pi_n, DAD_t(\pi_n)) := x_n\},$$

here $DAD_t(\pi_i) =< I^i_{j_1}, I^i_{j_2}, \ldots, I^i_{j_{m_i}} > (i = 1, \ldots, n)$. Suppose $DAD_t(\pi_1, \ldots, \pi_n) = < I_1, \ldots, I_{m_1}, I_{m_1+1}, \ldots, I_{m_1+m_2}, \ldots, I_{m_1+m_2+\cdots+m_{n-1}+1}, \ldots, I_{m_1+m_2+\cdots+m_{n-1}+m_n} >$ is an increasingly ordered set of $DAD_t(\pi_1), DAD_t(\pi_2), \ldots, DAD_t(\pi_n)$. Inverse substitution θ^{-1} is rational if and only if $DAD_t(\pi_1, \ldots, \pi_n)$ satisfies the following condition.

$$(I_{j+1} - I_j) \geq \mid \pi\{I_j\} \mid \text{ , for every } j = 2, \ldots, m_1 + \cdots + m_n,$$

where $\pi\{I_j\}$ is the π_i of $DAD_t(\pi_i)$ such that $I_j \in \{I^i_{j_1}, I^i_{j_2}, \ldots, I^i_{j_{m_i}}\}$.

Note that in the above theorem we can assume that

$$1 \leq I_1 < I_2 <, \ldots, < I_{m_1+m_2+\cdots+m_n} \leq \mid t \mid$$

since if there are two I_j, I_{j+1} and $I_j = I_{j+1}$, then it is obviously that we have the inverse substitution θ^{-1} is not rational.

The above theorem gives us a method to avoid the inter-contradiction when using inverse substitution.

3.3 Inverse Substitution for a Set of Ground Terms

In the previous section we have given the concept of inverse substitution for a single term. It is necessary to consider inverse substitution for a set of terms. When we want to learn some EFS rule from a set of examples(i.e. ground patterns in Σ^+), it is also necessary to get some more general patterns from these examples for constructing new EFS rules. In order to get such general patterns, in this section we introduce a notion of inverse substitution for set of ground terms.

Definition 17 *Let* $S = \{p_1, p_2, \ldots, p_m\}$ *be a set of strings, and* $s_1, s_2, \ldots, s_n$ *be common substrings in* S_{common}. *Then the inverse substitution* θ_S^{-1} for S is defined as the following finite set

$$\begin{aligned}\{&[s_1, (DAD_{p_1}(s_1), \ldots, DAD_{p_m}(s_1))] := x_1,\\ &[s_2, (DAD_{p_1}(s_2), \ldots, DAD_{p_m}(s_2))] := x_2,\\ &\vdots\\ &[s_n, (DAD_{p_1}(s_n), \ldots, DAD_{p_m}(s_n))] := x_n\},\end{aligned}$$

where $x_1, x_2, \ldots, x_n$ are disjoint variables.

We define the set $\{s_1, s_2, \ldots, s_n\}$ as the substituted set, $\{x_1, x_2, \ldots, x_n\}$ as the substituting set of a given inverse substitution θ_S^{-1}, and denote them by $SDS(\theta_S^{-1})$, $SGS(\theta_S^{-1})$ respectively. Obviously there should be many inverse substitutions for a set. An example follows.

Example 7 *Let* $S = \{p_1 = aabaabaaabba, p_2 = bbaaaabaabb\}$,*and* $s_1 = ab, s_2 = ba$ *be two common substrings of S. Then*

$$\begin{aligned}\theta_S^{-1}[1] &= \{[s_1, (<2>, <6, 9>)] := x_1, [s_2, (<6>, <2>)] := x_2\},\\ \theta_S^{-1}[2] &= \{[s_1, (<5>, <6>)] := x_1, [s_2, (<3, 11>, <2>)] := x_2\}\end{aligned}$$

are two inverse substitutions for S, and

$$\begin{aligned}S\theta_S^{-1}[1] &= \{ax_1aax_2aabba, bx_2aax_1ax_1b\},\\ S\theta_S^{-1}[2] &= \{aax_2x_1aaabx_2, bx_2aax_1aabb\}.\end{aligned}$$

But

$$\theta_S^{-1} = \{[s_1, (<2>, <6, 9>)] := x_1, [s_2, (<3>, <7>)] := x_2\}$$

is a bad inverse substitution because it causes contradiction and $S\theta_S^{-1}$ is not executable.

Definition 18 *An inverse substitution is called bad if it causes some contradictions that make the execution of substitution impossible .*

From the theorem 3, we can get the following necessary and sufficient condition for an inverse substitution not to be bad.

Theorem 4 *Suppose* $S = \{p_1, p_2, \ldots, p_m\}$ *is a set of strings,* $s_1, s_2, \ldots, s_n$ *are common substrings in* S_{common}. *Then the inverse substitution* θ_S^{-1}

$$\begin{aligned}\{&[s_1, (DAD_{p_1}(s_1), \ldots, DAD_{p_m}(s_1))] := x_1,\\ &[s_2, (DAD_{p_1}(s_2), \ldots, DAD_{p_m}(s_2))] := x_2,\\ &\vdots\\ &[s_n, (DAD_{p_1}(s_n), \ldots, DAD_{p_m}(s_n))] := x_n\},\end{aligned}$$

is not bad if and only if the following inverse substitutions for single term are all rational.

$$\begin{aligned}&\{(s_1, DAD_{p_1}(s_1)) := x_1, (s_2, DAD_{p_1}(s_2)) := x_2, \ldots, (s_n, DAD_{p_1}(s_n)) := x_n\},\\ &\{(s_1, DAD_{p_2}(s_1)) := x_1, (s_2, DAD_{p_2}(s_2)) := x_2, \ldots, (s_n, DAD_{p_2}(s_n)) := x_n\},\\ &\vdots\\ &\{(s_1, DAD_{p_m}(s_1)) := x_1, (s_2, DAD_{p_m}(s_2)) := x_2, \ldots, (s_n, DAD_{p_m}(s_n)) := x_n\}.\end{aligned}$$

3.4 Algorithm for Computing Inverse Substitutions

Now we give a non-deterministic algorithm for computing inverse substitutions of a ground term set. As pointed out in the previous sections, some inverse substitutions would cause contradiction, so that these inverse substitutions are not executable. In order to avoid the creation of this kind of useless inverse substitutions, we suppose that there is a procedure Check(*input* : θ^{-1}; *output* : CONTRA) for checking whether θ^{-1} is a bad inverse substitution.

Algorithm **Compute-IS**
input $S = \{s_1, s_2, \ldots, s_m\}$, $s_i = s_i^1 s_i^2 \ldots s_i^{m_i} (i = 1, \ldots, m)$
output θ_S^{-1}
begin
  call Find-Common-String(S)
  get $T = \{t_1, \ldots, t_k\} \subseteq S_{common}$
ex: **for** i=1 **to** k **step** 1 **do**
    begin
    for j=1 **to** m **step** 1 **do choose** $DAD_{s_j}(t_i)$
    $\theta_S^{-1} := \theta_S^{-1} \cup \{[t_i, (DAD_{s_1}(t_i), \ldots, DAD_{s_m}(t_i)] := x_i\}$
    end
  call Check(θ_S^{-1})
  if CONTRA=**yes then goto** ex:
  output θ_S^{-1}
end

The algorithm Compute-IS(S) give inverse substitution once at a time non-determinately. From the Algorithm, we see that choosing the $DAD_{s_j}(t_i)$ is non-determinate.

Example 8 *Let* $S = \{p_1 = aabaabaaabba, p_2 = bbaaaabaabb\}$. *Then using the Find-Common-String algorithm we get* $S_{\text{COMMON}} = \{a, ab, ba, aba, abb, bba, \ldots\}$. Also let T=ab, ba. Then by executing the algorithm, we get the following inverse substitutions.

$\theta_S^{-1}[1] = \{[ab, (< 2 >, < 6, 9 >)] := x_1, [ba, (< 6 >, < 2 >)] := x_2\}$,
$\theta_S^{-1}[2] = \{[ab, (< 5 >, < 6 >)] := x_1, [ba, (< 3, 11 >, < 2 >)] := x_2\}$,
$\theta_S^{-1}[3] = \{[ab, (<>, < 6 >)] := x_1, [ba, (< 6 >, < 2 >)] := x_2\}$,
$\theta_S^{-1}[4] = \{[ab, (< 5 >, < 6 >)] := x_1, [ba, (< 3 >, <>)] := x_2\}$

4 An Non-deterministic Algorithm for Learning OSL Languages

In this section we will use the inverse substitution defined in the previous section to introduce some operators for generating new rules. Here we introduce an absorption-like operator which learns rules from ground examples of the desired language. Using it we give an algorithm for learning some EFS languages.

4.1 Operator GROUND-ABS

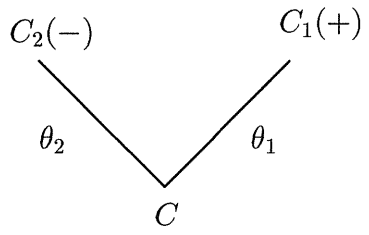

Figure 1: one step resolution

Figure 1 is just one step resolution that C is derived from a ground example C_1 and a clause C_2 with one predicate symbol in its body. We can write it as the following formula.

$$C = (C_1 - \{L_1\})\theta_1 \cup (C_2 - \{L_2\})\theta_2 \tag{1}$$

where L_1 is a positive literal in C_1, L_2 is a negative literal in C_2, and $\overline{L_1}\theta_1 = L_2\theta_2$.

We construct an operator GROUND-ABS(C, C_1) to perform the inverse of Figure 1. It inversely derives clause C_2 from clauses C and C_1. Formula (1) can be used to compute clause C based on the selection of substitutions θ_1 and θ_2. For computing C_2 from formula (1), we can use the concept of inverse substitution defined in the previous section. Next we consider a special case to construct GROUND-ABS(C, C_1).

Let C and C_1 be ground unit clauses. Then θ_1 is empty set, $C_1 = L_1$. From formula (1), we can get following formula for C_2.

$$C_2 = C\theta_2^{-1} \cup \{L_2\} \tag{2}$$

where θ_2^{-1} is a inverse substitution of θ_2. Using $L_2 = \overline{L_1}\theta_2^{-1}$ to replace L_2 in formula (2), we can get the following formula (3).

$$C_2 = (C \cup \{\overline{L_1}\})\theta_2^{-1} \tag{3}$$

Since there may be a lot of inverse substitutions of θ_2, the computation of θ_2 is non-deterministic. We call the formula (3) as GROUND-ABS$(\{C, C_1\})$. In the next section we consider algorithm to compute it.

Example 9 *Suppose*

$C = p(bbcc) \leftarrow$,
$C_1 = p(bc) \leftarrow$,
$L_1 = p(bc)$,

and the following inverse substitution of θ_2 is chosen.

$$\theta_2^{-1} = \{[bc, (< 2 >, < 1 >)] := x\}$$

Then according to formula (3)*, we get*

$$\begin{aligned} C_2 =& GROUND\text{-}ABS(\{C, C_1\}) = (C \cup \{\overline{L_1}\})\theta_2^{-1} \\ =& (p(bbcc) \leftarrow p(bc))\theta_2^{-1} = p(bxc) \leftarrow p(x). \end{aligned}$$

Hence, using the operator GROUND-ABS(S), we can learn a rule from ground examples.

4.2 Algorithm for Computing GROUND-ABS

In the previous section, we have given the rule-generating operator GROUND-ABS(S) which learns some rules from an example set S. From the formula (3) we know that the main problem for computing GROUND-ABS(S) is how to compute the inverse substitution θ_2^{-1}. In this section, we give a general algorithm for computing GROUND-ABS(S).

Suppose S is a given set of ground terms. We consider algorithm for computing GROUND-ABS(S). For convenience, we assume that there is a procedure GET-STRING(S) to return us a string in S. Since GROUND-ABS(S) is one step inverse resolution, we can assume that the substituted set of θ_S^{-1} computed by Compute-IS(S) has just one term. We have the following non-deterministic algorithm.

Algorithm **General**
input $S = \{p_1, p_2, \ldots, p_n\}$, $p_i = p_i^1 p_i^2 \ldots p_i^{n_i} (i = 1, \ldots, n)$
output CLAUSES
begin
  call Compute-IS(S)
  π=GET-STRING(S)
  get SDS(θ_S^{-1}) = $\{t\}$; SGS(θ_S^{-1}) = $\{x\}$
  create $\tau = \pi\{[t, DAD_\pi(t)] := x\}$
  create clause $p(\tau) \leftarrow p(x)$
  output CLAUSES=$\{p(\tau) \leftarrow p(x),$
    $p(t) \leftarrow\}$
end

Algorithm General(S) computes operator GROUND-ABS(S), and returns us a set of clauses. The execution of algorithm General(S) is non-determinate since it call procedure Compute-IS(S) to compute an inverse substitution for S.

Example 10 *Suppose* S=$\{bc, bbcc, bbbccc, bbbbcccc\}$, *and the inverse substitution*

$$\theta_2^{-1} = \{[bc, (< 1 >, < 2 >, < 3 >, < 4 >)] := x\}$$

is computed by algorithm Compute-IS(S). Getting $\pi = bbcc$, *then*

$$\tau = \pi\{[bc, DAD_\pi(bc)] := x\} = bxc,$$

then we get the output

$$\text{CLAUSES} = \{p(axc) \leftarrow p(x), p(bc) \leftarrow\}.$$

The above algorithm **General**(S, CLAUSES) also gives us a non-deterministic method to learn an one-side linear EFS language from a given example set, though there are so many non-deterministic features and some key examples should be included in the example set. In the next section we will consider how to deal with these intractabilities by introducing some bias to our language classes.

4.3 Repetition-reducing Operators

If a grammar generates infinite language, then it necessarily includes some repetitive structures. It is the case for our EFSs.

When we consider learning of EFS languages, some examples of the desired language will be given.

By observing and analyzing the examples, we should be able to find out the features that the desired language has. This is what we call an intellectual activity, and in general it is a difficult activity even for us human being. If we want to make our learning system able to do such an activity, some mechanisms for discovering this kind of features from the known examples should be given to the system. Before we can teach the method to the system, first we should understand the spirits of the method. In this section we consider such repetitions as a key point for learning EFS languages. We give a repetition-reducing operator which helps our system to analyze the given examples to find out the repetitive part for constructing the desired grammars. First we give definitions of two kinds of repetitions about terms.

Definition 19 *Given term π. The subterm τ with place $< i >$ of π is said to have a global repetition in π if term $\tau\tau$ is also in π and has $< i >$ or $< i- \mid \tau \mid >$ as its place in π. If $\pi = \tau\tau\cdots\tau$ for several τ, we say that π is a complete global repetition of τ.*

Definition 20 *Given term π. The subterm $\tau = \tau_1\tau_2$ with place $< i >$ of π is said to have a partial repetition in π if term $\tau_1\tau_1\tau_2\tau_2$ is also in π and has $< i- \mid \tau_1 \mid >$ as its place in π.*

The above two kinds of repetitions are different inherently. We give examples to describe the above definitions of global and partial repetitions.

Example 11 *Suppose $\pi = ababab$. $\tau = ab$ is a subterm of π with place $< 3 >$. Then τ has a global repetition in π since $\tau\tau = abab$ with place $< 1 >$ is also in π. From the definition we see that each of the three place's ab has a global repetition in π but has no partial repetition.*

Example 12 *Suppose $\pi = aaabbb$. $\tau = ab$, here $\tau_1 = a$ and $\tau_2 = b$, is a subterm of π with place $< 3 >$. Then τ has a partial repetition in π since $\tau_1\tau_1\tau_2\tau_2 = abab$ with place $< 2 >$ is also in π. From the definition we see that $\tau = ab$ has no global repetition in π.*

Note that there may happen a case where two kinds of repetitions exist simultaneously in a term with respect to the same subterm or different subterms.

Now we introduce the following operators, called a repetition-reducing operator, which reduce the repetition of some subterm from the given term. According to the classification of repetitions above we have the following two kinds of repetition-reducing operators.

Definition 21 *Given term π. Suppose subterm τ with place $< i >$ of π has a global repetition in π. We define the G-operator which makes the global repetition $\tau\tau$ in place $< i >$ or $< i- \mid \tau \mid >$ to just a single τ in π.*

Definition 22 *Given term π. Suppose subterm $\tau = \tau_1\tau_2$ with place $< i >$ of π has a partial repetition in π. We define the P-operator which makes the partial repetition $\tau_1\tau_1\tau_2\tau_2$ in place $< i- \mid \tau_1 \mid >$ to $\tau_1\tau_2$ in π.*

Next we will consider the problem of how to find out the repetition subterms in a given particular term. Our purpose is, through analyzing the repetition subterms of the given examples, to get some hints for learning the EFS(grammar) which generates our desired language. That is, the learned EFS(grammar) can explain all the given examples and give expectation to the future results of experimentations.

Given any term π, and any one of its subterm τ. The address

$$AD_\pi(\tau) =< I_1, I_2, \ldots, I_n >$$

gives the all appearances of τ in π with their places in turn. We are interesting in the appearances which are repetitions in the term π. First we should check out the appearances which have a global or partial repetition in the given term π. We give the following algorithm which checks out the repetition appearances of the subterm τ in π. For convenience we give the following notations.

Definition 23 *Given term $\pi = a_1a_2\cdots a_n$. We use $\pi(i,k)$ to denote the subterm $a_ia_{i+1}\cdots a_{i+k-1}$ of π starting from a_i with length k. If i is less than 1 then $\pi(i,k)$ is defined as $a_1a_2\cdots a_k$, and if $i+k-1$ is greater than n then $\pi(i,k)$ is defined as $a_ia_{i+1}\cdots a_n$.*

Now we give the following procedure to check the repetition.

Procedure **RE-CHECK**
input: term π, one of subterms τ of π and $AD_\pi(\tau) =< I_1, I_2, \ldots, I_n >$
output: a subset of RE-AD$_\pi(\tau)$ of $AD_\pi(\tau)$
begin
for i=1 **to** n **do**
begin
(global repetition check)
if $\pi(i, 2* \mid \tau \mid) = \tau\tau$ or $\pi(i- \mid \tau \mid, 2* \mid \tau \mid) = \tau\tau$ **then**
RE-AD$_\pi(\tau)$ =RE-AD$_\pi(\tau) + \{i\}$
else
(partial repetition check)
for j=1 **to** $\mid \tau \mid$ **do**
if $\tau(1,j)\tau(1,j)\tau(j+1, \mid \tau \mid -j)\tau(j+1, \mid \tau \mid -j)$
$= \pi(i- \mid \tau(1,j) \mid, 2* \mid \tau \mid)$ **then**
RE-AD$_\pi(\tau)$ =RE-AD$_\pi(\tau) + \{i\}$
end
output RE-AD$_\pi(\tau)$
end

RE-AD$_\pi(\tau)$ gives all the possible repetition subterms and their places that should be considered as the candidates of the repetitive parts of the desired rules. For fixing which are the real repetition, some other additional information is needed. We will reduce our language classes to make the fixing easy.

5 Learning Restricted OSL Languages

In this section, we consider learning of the restricted one-side linear(R-OSL) languages.

Learning the axioms Γ of an EFS to define the desired language just from giving examples should be the most primitive and natural setting of learning language. The learnt EFS should be able to explain all the given examples. There are a lot of learning systems in use of background knowledge. In this paper we do not use such background knowledge but just use the given examples in the desired language from the restricted one-side linear language class. We may just consider the case of right-linear languages, because the left-linear languages are equivalent to the right-linear one.

From the definition of R-OSL EFS languages, the repetition feature can easily extracted from the given examples. Since the restricted one-side linear EFSs have at most one clause with the same predicate symbol in both head and body, the essential parts of repetition in the given examples of R-OSL language should appear in just one place.

We need some notations before we introduce the algorithm Suppose L is the R-OSL EFS language that we want to learn. Given alphabet Σ, and a set of examples EX(L) is a subset of L. The algorithm produces a set of clauses Γ. Π is the predicate symbols used in Γ. Predicate symbol p is a special one in Π. We give the following definition.

Definition 24 *Let $S = (\Sigma, \Pi, \Gamma)$ Then language $L(S, p)$ is called a proper EFS language for L if the following holds.*

$$EX(L) \subset L(S,p) = \{\alpha \in \Sigma^+ | \Gamma \vdash p(\alpha)\}$$

There should be many proper EFS languages for a given EX(L). The following algorithm gives us an output which is a proper EFS language for the desired language L. The algorithm learns the desired language using some bottom-up like method to avoid unnecessary over-generalization to lead to trivial EFS languages. For giving EX(L), we suppose s is the shortest example in EX(L). Since the desired language is in restricted OSL EFS class, the real repetition part should be a substring of all elements in EX(L). We define a series of string sets based on s and EX(L).

Definition 25 *Given example set $EX(L)$ and the shortest string s in it. We define $EX_1(L)$ as following from s and $EX(L)$.*

$EX_1(L) = \{t \mid r \in EX(L)$, *t is the tail of r deleting s(1, 1) from the leftmost of r*$\}$

Generally we have the definition of $EX_k(L)(1 \leq k \leq | s |)$ *as*

$EX_k(L) = \{t \mid r \in EX(L)$, *t is the tail of r deleting* $s(1,k)$ *from the leftmost of r*$\}$.

Similarly we have the definition of $EX^k(L)(1 \leq k \leq | s |)$ *as follows.*

$EX^k(L) = \{t \mid r \in EX(L)$, *t is the head of r deleting* $s(| s | -k+1, k)$ *from the rightmost of r*$\}$

The main task that the algorithm should do is how to find out the repetition part from the given EX(L). We will utilize the set series of $EX_k(L)$ and $EX^k(L)$ to help finding out the repetition place. We define $EX_0(L) = EX^0(L) = EX(L)$.

Definition 26 *Suppose EX is a set of strings and s is a single string. If every element in EX is a complete global repetition of s, we say that set EX is a complete global repetition set of string s. We denote the complete global repetition as CGR.*

Algorithm **MAIN**
input: example set EX(L) of desired language L
output: clause set Γ s.t. L(S, p) is proper for L, where $S = (\Sigma, \Pi, \Gamma)$ and
p is a predicate symbol in Γ
begin
set $\Gamma = \Phi$; select string s as the shortest one in $EX(L)$
for k=0 **to** $\mid s \mid$ **do**
begin
create $EX_k(L)$;
for i=0 **to** $\mid s \mid$ **do**
begin
create $EX_k^i(L)$;
check $s(k+1, \mid s \mid -k-i)$ and $EX_k^i(L)$
if $EX_k^i(L)$ is a complete global repetition set of $s(k+1, \mid s \mid -k)$
then do
begin
create clauses

$$C_1 = p_1(s(\mid s \mid -i+1, i) \leftarrow$$
$$C_2 = p_2(s(k+1, \mid s \mid -k-i)x) \leftarrow p_1(x)$$
$$C_3 = p_2(s(k+1, \mid s \mid -k-i)x) \leftarrow p_2(x)$$
$$C_4 = p(s(1,k)x) \leftarrow p_2(x)$$

add C_1, C_2, C_3, C_4 to Γ
output Γ **goto** exit
end
end
end
exit :
end

We have the following lemma to guarantee that the learnt EFS is a proper EFS language of the desired language.

Lemma 2 *The MAIN algorithm computes a proper EFS language for any restricted one-side linear EFS language by giving an example set including the shortest example of the desired language .*

Our criterion of successful learning, i.e. the properness, is very weak, and hence there may allow undesired ones. But the EFS language learnt by MAIN is considerably a good EFS for the desired language.

We give some concrete executing examples of algorithm MAIN.

Example 13 *Suppose the desired language L is $ab\{aba\}^+bba \in$ R-OSL-EFS, and the given example set is*

$$EX(L) = \{abababba, ababaabaababba, ababaabaabaababba, ababaabaabaabaababba\}.$$

We trace the execution of MAIN to learn the EFS language for L.

step 1: *set $\Gamma = \Phi$, select s=abababba. $k =\mid s \mid = 8$.*
step 2: *k=0: create $EX_0(L) = EX(L) =$*
$\{abababba, ababaabaababba, ababaabaabaababba, ababaabaabaabaababba\}$.
i=0: create $EX_0^0(L) = EX(L) =$
$\{abababba, ababaabaababba, ababaabaabaababba, ababaabaabaabaababba\}$.
check EX_0^0, a CGR of $s(1,8) = abababba$? **no**
i=1: create $EX_0^1(L) =$
$\{abababb, ababaabaababb, ababaabaabaababb, ababaabaabaabaababb\}$.
check EX_0^1, a CGR of $s(1,7) = abababb$? **no**
check as above, case of i=2, 3, 4, 5, 6, 7, 8 return **no**
step 3: *k=1: create $EX_1(L) =$*
$\{bababba, babaabaababba, babaabaabaababba, babaabaabaabaababba\}$.
do the same for i=0 to 8, the answers are all **no**
step 4: *k=2: create $EX_2(L) =$*
$\{ababba, abaabaababba, abaabaabaababba, abaabaabaabaababba\}$.
for i=0, 1, 2, the answers are all **no**
i=3: create $EX_2^3(L) =$
$\{aba, abaabaaba, abaabaabaaba, abaabaabaabaaba\}$.
check $EX_2^3(L)$, a CGR of $s(3,5) = aba$? **yes**
step 5: *create clauses*
$C_1 = p_1(s(\mid s \mid -i+1, i) \leftarrow\!= p_1(bba) \leftarrow$
$C_2 = p_2(s(k+1, \mid s \mid -k-i)x) \leftarrow p_1(x) = p_2(abax) \leftarrow p_1(x)$
$C_3 = p_2(s(k+1, \mid s \mid -k-i)x) \leftarrow p_2(x) = p_2(abax) \leftarrow p_2(x)$
$C_4 = p(s(1,k)x) \leftarrow p_2(x) = p(abx) \leftarrow p_2(x)$
step 6: *add C_1, C_2, C_3, C_4 to Γ*
step 7: *output Γ and terminate the execution*

We get the following EFS $S = (\Sigma, \Pi, \Gamma)$ with

$$\Gamma = \left\{ \begin{array}{l} p(abx) \leftarrow p_2(x), \\ p_2(abax) \leftarrow p_2(x), \\ p_2(abax) \leftarrow p_1(x), \\ p_1(bba) \leftarrow \end{array} \right\}.$$

We have $EX(L) \subset L(S,p) = \{\alpha \in \Sigma^+ | \Gamma \vdash p(\alpha)\}$.

Example 14 *Suppose the desired language L is $\{a^{3n} \mid n \geq 1\} \in$ R-OSL-EFS. Obviously for an EFS $S = (\Sigma, \Pi, \Gamma)$ with*

$$\Gamma = \left\{ \begin{array}{l} p(ax) \leftarrow p(x), \\ p(a) \leftarrow \end{array} \right\},$$

$L(S,p)$ is a trivial proper EFS language for L.

Suppose the given example set is

$$EX(L) = \{aaa, aaaaaaaaa, aaaaaaaaaaaaaaa, aaaaaaaaaaaaaaaaaaaaa\}.$$

We trace the execution of MAIN to learn the EFS language for L.

step 1: *set* $\Gamma = \Phi$, *select s=aaa.* $k = \mid s \mid = 3$.
step 2: *k=0: create* $EX_0(L) = EX(L) =$
$\{aaa, aaaaaaaaa, aaaaaaaaaaaaaaa, aaaaaaaaaaaaaaaaaaaaa\}$.
i=0: create $EX_0^0(L) = EX(L) =$
$\{aaa, aaaaaaaaa, aaaaaaaaaaaaaaa, aaaaaaaaaaaaaaaaaaaaa\}$.
check EX_0^0, *a CGR of* $s(1,3) = aaa$? **yes**
step 3: *create clauses*
$C_1 = p_1(s(\mid s \mid -i+1, i) \leftarrow = p_1(\epsilon) \leftarrow$
$C_2 = p_2(s(k+1, \mid s \mid -k-i)x) \leftarrow p_1(x) = p_2(aaax) \leftarrow p_1(x)$
$C_3 = p_2(s(k+1, \mid s \mid -k-i)x) \leftarrow p_2(x) = p_2(aaax) \leftarrow p_2(x)$
$C_4 = p(s(1,k)x) \leftarrow p_2(x) = p(x) \leftarrow p_2(x)$
step 4: *add* C_1, C_2, C_3, C_4 *to* Γ
step 5: *output* Γ *and terminate the execution*

We get the following EFS $S = (\Sigma, \Pi, \Gamma)$ *with*

$$\Gamma = \left\{ \begin{array}{l} p(x) \leftarrow p_2(x), \\ p_2(aaax) \leftarrow p_2(x), \\ p_2(aaax) \leftarrow p_1(x), \\ p_1(\epsilon) \leftarrow \end{array} \right\} = \left\{ \begin{array}{l} p(x) \leftarrow p_2(x), \\ p_2(aaax) \leftarrow p_2(x), \\ p_2(aaa) \leftarrow \end{array} \right\}.$$

We have $EX(L) \subset L(S,p) = \{\alpha \in \Sigma^+ | \Gamma \vdash p(\alpha)\}$.

We give one more example.

Example 15 *Suppose the desired language L is* $\{a^{2n+1} \mid n \geq 1\} \in$ *R-OSL-EFS. The desired EFS for L is* $S = (\Sigma, \Pi, \Gamma)$ *with*

$$\Gamma = \left\{ \begin{array}{l} p(aax) \leftarrow p_1(x), \\ p_1(aax) \leftarrow p_1(x), \\ p_1(a) \leftarrow \end{array} \right\},$$

Suppose the given example set is

$$EX(L) = \{aaa, aaaaaaaaa, aaaaaaaaaaaaaaa, aaaaaaaaaaaaaaaaaaaaa\}.$$

We trace the execution of MAIN to learn the EFS language for L.

step 1: *set* $\Gamma = \Phi$, *select s=aaa.* $k = \mid s \mid = 3$.
step 2: *k=0: create* $EX_0(L) = EX(L) =$
$\{aaa, aaaaaaaaa, aaaaaaaaaaaaaaa, aaaaaaaaaaaaaaaaaaaaa\}$.
i=0: create $EX_0^0(L) = EX(L) =$
$\{aaa, aaaaaaaaa, aaaaaaaaaaaaaaa, aaaaaaaaaaaaaaaaaaaaa\}$.
check EX_0^0, *a CGR of* $s(1,3) = aaa$? **yes**
step 5: *create clauses*
$C_1 = p_1(s(\mid s \mid -i+1, i) \leftarrow = p_1(\epsilon) \leftarrow$
$C_2 = p_2(s(k+1, \mid s \mid -k-i)x) \leftarrow p_1(x) = p_2(aaax) \leftarrow p_1(x)$
$C_3 = p_2(s(k+1, \mid s \mid -k-i)x) \leftarrow p_2(x) = p_2(aaax) \leftarrow p_2(x)$
$C_4 = p(s(1,k)x) \leftarrow p_2(x) = p(x) \leftarrow p_2(x)$
step 6: *add* C_1, C_2, C_3, C_4 *to* Γ
step 7: *output* Γ *and terminate the execution*

We get the following EFS $S = (\Sigma, \Pi, \Gamma)$ *with*

$$\Gamma = \left\{ \begin{array}{l} p(x) \leftarrow p_2(x), \\ p_2(aaax) \leftarrow p_2(x), \\ p_2(aaax) \leftarrow p_1(x), \\ p_1(\epsilon) \leftarrow \end{array} \right\} = \left\{ \begin{array}{l} p(x) \leftarrow p_2(x), \\ p_2(aaax) \leftarrow p_2(x), \\ p_2(aaa) \leftarrow \end{array} \right\}.$$

Although it is not the desired one we want, we have $EX(L) \subset L(S,p) = \{\alpha \in \Sigma^+ | \Gamma \vdash p(\alpha)\}$. *That is, MAIN learns an EFS language that can explain all the known examples. The reason that MAIN did not compute the desired one is in the lack of examples for the learning. So we add some new examples to MAIN and run MAIN again. Then we get the desired EFS language for L in the following way.*

Suppose the given example set is

$$EX(L) =$$
$$\{aaa, \mathbf{aaaaa}, \mathbf{aaaaaa}, aaaaaaaaa, aaaaaaaaaaaaaa, aaaaaaaaaaaaaaaaaa\}.$$

We trace the execution of MAIN to learn the EFS language for L.

step 1: *set* $\Gamma = \Phi$, *select s=aaa.* $k =| s |= 3$.
step 2: *k=0: create* $EX_0(L) = EX(L) =$
$\{aaa, \mathbf{aaaaa}, \mathbf{aaaaaa}, aaaaaaaaa, aaaaaaaaaaaaaa, aaaaaaaaaaaaaaaaaa\}$.
i=0: create $EX_0^0(L) = EX(L) =$
$\{aaa, \mathbf{aaaaa}, \mathbf{aaaaaa}, aaaaaaaaa, aaaaaaaaaaaaaa, aaaaaaaaaaaaaaaaaa\}$.
check EX_0^0, *a CGR of* $s(1,3) = aaa$? **no**
i=1: create $EX_0^1(L) =$
$\{aa, \mathbf{aaaa}, \mathbf{aaaaa}, aaaaaaaa, aaaaaaaaaaaaaa, aaaaaaaaaaaaaaaaaa\}$.
check EX_0^1, *a CGR of* $s(1,2) = abababb$? **yes**
step 3: *create clauses*
$C_1 = p_1(s(| s | -i+1, i) \leftarrow = p_1(a) \leftarrow$
$C_2 = p_2(s(k+1, | s | -k-i)x) \leftarrow p_1(x) = p_2(aax) \leftarrow p_1(x)$
$C_3 = p_2(s(k+1, | s | -k-i)x) \leftarrow p_2(x) = p_2(aax) \leftarrow p_2(x)$
$C_4 = p(s(1,k)x) \leftarrow p_2(x) = p(x) \leftarrow p_2(x)$
step 4: *add* C_1, C_2, C_3, C_4 *to* Γ
step 5: *output* Γ *and terminate the execution*

We get the following EFS $S = (\Sigma, \Pi, \Gamma)$ *with*

$$\Gamma = \left\{ \begin{array}{l} p(x) \leftarrow p_2(x), \\ p_2(aax) \leftarrow p_2(x), \\ p_2(aax) \leftarrow p_1(x), \\ p_1(a) \leftarrow \end{array} \right\}.$$

6 Conclusion and Discussion

We discussed the inverse resolution in the framework of EFSs(elementary formal systems). In EFS, there are no auxiliary symbols like a pair of parentheses in the ordinary logic programming languages, or more exactly the terms in EFS framework are just defined as patterns of symbols and variables. This gives rise to a lot of difficult and intractable problems when we discuss inverse resolution on EFS. In this paper, we have pointed out that these problems can be captured by the concepts of

the intra- and inter-contradictions in the inverse substitutions, and given a method to avoid these contradictions. Based on the method we have given a non-deterministic algorithm to compute some classes of EFS languages.

We also considered an efficient learning algorithm for a small class of EFS languages called a restricted EFS language class. The algorithm can compute an EFS for any desired language in the class just by using examples in the language but not using any background knowledge or oracle's help. It uses a kind of bottom-up mechanism to avoid over-generalization.

This paper considered the learning of a relatively small EFS language class. Using it as a basis, the larger language classes will be considered using a hierarchical learning concept as we discussed in [23].

References

[1] A.V. Aho (1990) : *Algorithms for Finding Pattern in Strings.* in J. van Leeuwen(managing editor) : *Handbook of Theoretical Computer Science, Volume A, Algorithms and Complexity,* Elsevier Science Publishers B.V., 1990, pp. 256-300.

[2] A.V. Aho, J.E. Hopcroft and J.D. Ullman (1974) : *The Design and Analysis of Computer Algorithms.* Addison-Wesley, Reading, Mass.

[3] D. Angluin (1980) : *Finding Pattern Common to a Set of Strings.* Journal of Computer and System Sciences, 21: pp. 46-62, 1980.

[4] D. Angluin and C.H. Smith (1983) : *Inductive Inference : Theory and Methods.* in ACM computing Surveys 15, 237-269.

[5] S. Arikawa (1970) : *Elementary Formal Systems and Formal Languages-Simple Formal Systems.* Memories of Fac.Sci., Kyushu Univ. Ser. A., Math. 24: pp. 47-75, 1970.

[6] S. Arikawa, T. Shinohara and A. Yamamoto (1989) : *Elementary Formal System as a Unifying Framework for Language Learning.* In Proceedings of the Second Annual Workshop on Computational Learning Theory(COLT'89), pp. 312-327.

[7] E. Gold (1967) : *Language Identification in the Limit.* Inform. and Control 10, pp. 47-474.

[8] D.E. Knuth, J.H. Morris and V.R. Pratt (1977) : *Fast Pattern Matching in Strings.* SIAM J. Comput. 6(2), 1977, pp. 323-350.

[9] P.D. Laird (1988) : *Learning from Good and Bad Data.* Kluwer Academic Publishers, 1988.

[10] X. Ling (1989) : *Inventing Theoretical Terms in Inductive Learning of Functions - Search and Constructive Method.* In Z.W. Ras, editor, Methodologies for Intelligent Systems, 4, pp. 332-341. North-Holland, October, 1989.

[11] X. Ling (1989) : *Learning and Invention of Horn Clause Theories - a Constructive Method.* In Z.W. Ras, editor, Methodologies for Intelligent Systems, 4, pp. 323-331. North-Holland, October, 1989.

[12] S. Muggleton (1987) : *Duce, an Oracle Based Approach to Constructive Induction.* In IJCAI-87, Kaufmann, pp. 287-292.

[13] S. Muggleton and W. Buntine (1988) : *Towards Constructive Induction in first-order predicate calculus.* Turing Institute Working Paper.

[14] S. Muggleton and W. Buntine (1988) : *Machine Invention of First-Order Predicates by Inverting Resolution.* In *Machine Learning 5* , Kaufmann, pp. 339-352, 1988.

[15] S. Muggleton and C. Feng (1990) : *Efficient Induction of Logic Programs.* In Proceedings of the First Conference on Algorithm Learning Theory, Ohmsha, Tokyo, 1990.

[16] J.R. Quinlan (1991) : *Determinate Literals in Inductive Logic Programming.* In proceedings of the Eighth International Workshop on Machine Learning, Ithaca, New york, 1991, Morgan Kaufmann.

[17] Y. Sakakibara (1989) : *On Learning Elementary Formal Systems: Towards an Efficient Learning for Context-Sensitive Languages.* Research Report 97, IIAS-SIS, FUJITSU LIMITED, 1989.

[18] T. Shinohara (1990) : *Inductive Inference from Positive Data is Powerful.* In Proceedings of the Third Annual Workshop on Computational Learning Theory(COLT'90), pp. 97-110.

[19] R. Wirth (1988) : *Learning by Failure to Prove.* in Proceedings of the Third European Working Session on Learning, Biddles Ltd, Guildford and King's Lynn, 1988.

[20] R. Wirth (1989) : *Completing Logic Programs by Inverse Resolution.* in Proceedings of the Fourth European Working Session on Learning, Pitman, Morgan Kaufmann, 1989.

[21] A. Yamamoto (1989) : *Elementery Formal System as a Logic Programming Language.* Technical Report RIFIS-TR-CS-12, Research Institute of Fundamental Information Science, Kyushu University.

[22] C. Zeng and S. Arikawa (1991) : *Sufficiency of Operators Identification and Inter-construction in Inverting Resolution.* Bulletin of Informatics and Cybernetics, Vol. 24, No. 3~4, 1991.

[23] C. Zeng (1991) : Inductive Learning with Inverse Resolution. master thesis, Kyushu University (in Japanese).