Generalization and Predicate Invention in Learning Logic Programs

石坂, 裕毅

https://doi.org/10.11501/3065651

出版情報：九州大学, 1992, 博士（理学）, 論文博士
バージョン：
権利関係：

Chapter 5

Model Inference with Predicate Invention

It seems that the theory of model inference is applicable to more practical problems such as automated program synthesis. However, several problems in applying the theory to the practical problems are also pointed out [AI87]. The most serious problem is that we need to give in advance a first order language $\mathcal{L}$ with finitely many predicate symbols over which a target model is described. Furthermore, the oracle which gives information about a target model to an inference machine is assumed to be able to answer the truth of all elements in $\mathcal{B}_{\mathcal{L}}$. In more practical setting such as in automated program synthesis, it seems difficult to assume such a power on the oracle. The assumptions require a user of the synthesis system to have too much knowledge about the target program.

Now, we consider, via a simple simulation, what will happen when the assumptions are removed.

Suppose that an inexperienced programmer is going to make a program for the predicate $reverse(list1, list2)$, where $list2$ is a reversed list of $list1$, using a system which is an implementation of the model inference algorithm, e.g. Model Inference System [Sha82]. Since he/she is inexperienced, he/she does not have enough knowledge to understand of an algorithm for reversing a list or he/she understands an algorithm but he/she does not see what kind of auxiliary predicates (e.g. $concat(\_, \_, \_)$ or $append(\_, \_, \_)$ etc.) are necessary for the algorithm. So, he/she gives the first order language with only one predicate symbol

reverse(_, _) to the system, then he/she starts the task. He/She patiently continues to input facts about *reverse*(_, _). However, if there is no program for reversing a list which consists of only the predicate symbol, then his/her efforts will result in failure. On the other hand, suppose that the system realizes the difficulty of the programming with only one predicate *reverse*(_, _) and it introduces a new predicate as an auxiliary one. Since the programmer has no interpretation for such a new predicate, he/she cannot answer the questions from the contradiction backtracing algorithm which plays a central role in the model inference algorithm. Hence he/she cannot continue the inference process anymore.

Thus, it seems that the purpose of the inexperienced programmer is not accomplished. However, from the above simulation, the following two questions arise.

(1) Is there no list reversal program with only one predicate *reverse*(_, _).

(2) Is there no method for giving an interpretation, which reflects the programmer's intended model, to the new predicates introduced by the system.

For applying the theory of model inference to more practical problems such as automated program synthesis, it seems very important to answer the above two questions. We leave the first question open for researchers of the theory of logic programs. Here, we reconsider the model inference from the point of view of the second question.

The predicates, which may be necessary to define a target predicate but are not directly observable from the examples of the target predicate, are called *theoretical terms* [Sha81]. Shapiro assumed that the theoretical terms and their intended interpretations necessary for defining the target predicate are given to the inference algorithm. Recently, however, several researchers are trying to develop a method for automatically inventing theoretical terms [MB88, Mug90, Ban88, Lin89b, Lin89a]. In this chapter, we also consider the challenging problem of inventing theoretical terms.

In Section 5.1, we extend the model inference problem so that it may include more practical problems. In Section 5.2, we consider the problems caused in solving the extended model inference problem, especially, in introducing new predicates. In Section 5.3, we propose an easy approach to the problem. Several classes of programs for which the easy approach might go well are introduced in Section 5.4.

This chapter is based on the paper [Ish88b].

5.1 An Extended Model Inference Problem

In this section, we redefine a model inference problem so that it can include more practical problems. We assume that a set of function symbols and a set of variable symbols are common to all first order languages considered in this chapter. The set of predicate symbols of a first order language $\mathcal{L}$ is denoted by $\Pi(\mathcal{L})$. For any first order language $\mathcal{L}$, an *extended language* $\mathcal{L}'$ of $\mathcal{L}$ is a first order language such that $\Pi(\mathcal{L}) \subseteq \Pi(\mathcal{L}')$. If $\mathcal{L}'$ is an extended language of $\mathcal{L}$, then $\mathcal{L}$ is called an *restricted language* of $\mathcal{L}'$. Let P be a program over $\mathcal{L}'$ that is an extended language of $\mathcal{L}$. Then $M(P)_{\mathcal{L}}$ denotes the set $M(P) \cap \mathcal{B}_{\mathcal{L}}$.

Let $\mathcal{L}$ be a first order language and M be the least Herbrand model of an unknown program over an extended language of $\mathcal{L}$. An *oracle for M over $\mathcal{L}$* is a device that can answer the membership of $\alpha \in \mathcal{B}_{\mathcal{L}}$ in M. The answer is *true* if $\alpha \in M$, *false* otherwise. Note that the oracle over $\mathcal{L}$ answers the membership for only the elements of $\mathcal{B}_{\mathcal{L}}$. If a ground atom $\beta \notin \mathcal{B}_{\mathcal{L}}$ is given, then the oracle is assumed to returns a special symbol $\perp$ that means "unknown".

An *extended model inference problem* is defined as follows:

> *Suppose that $\mathcal{L}$ and an oracle for M over $\mathcal{L}$ are given. Then infer a program P over an extended language $\mathcal{L}'$ of $\mathcal{L}$ such that $M(P)_{\mathcal{L}} = M$ from the information given by the oracle.*

Example 5.1 Let $\mathcal{L}$ be a first order language such that $\Pi(\mathcal{L}) = \{reverse(\_,\_)\}$ and $M = \{reverse(l_1, l_2) \in \mathcal{B}_{\mathcal{L}} \mid l_2$ is the reversed list of a list $l_1\}$. An extended model inference problem is to find a program such as

$$P = \left\{ \begin{array}{l} reverse([X|Y], Z) \leftarrow reverse(Y, Y_1), concat(X, Y_1, Z). \\ reverse([], []). \\ concat(X, [Y|Z], [Y|W]) \leftarrow concat(X, Z, W). \\ concat(X, [], [X]). \end{array} \right\}$$

only by using an oracle for M, that is, using information about $reverse(\_,\_)$. In this case, $\Pi(\mathcal{L}') = \Pi(\mathcal{L}) \cup \{concat(\_,\_,\_)\}$.

In what follows, we distinguish the use of terms, *predicate names*, *predicate symbols*, and *predicates* according to the objects indicated by them as follows. A *predicate name*

indicates just a string such as *reverse*, *concat*, etc. A *predicate symbol* indicates a predicate name with arity such as $reverse(\_,\_)$, $concat(\_,\_,\_)$, etc. A *predicate* indicates a predicate symbol concerned with its model, e.g. $reverse(\_,\_)$ concerned with the model $\{reverse([],[]), reverse([a],[a]), reverse([a,b],[a,b])\}$.

5.2 Problems in Extended Model Inference

The most difficult problem for an algorithm to solve an extended model inference problem is how to introduce an unknown predicate symbol in $\Pi(\mathcal{L}') - \Pi(\mathcal{L})$ when it is necessary. In this section, we review recent two approaches to such an extended model inference problem. One was given by Muggleton and Buntine [MB88] and another was given by Banerji [Ban88]. Through their approaches, we consider the essential problems in solving an extended model inference problem.

Muggleton and Buntine proposed a method called *inverse resolution* [MB88]. They implemented their method as a inductive logic programming system called CIGOL. Inverse resolution is executed by three operators, truncation, absorption, and intra-construction. The truncation constructs unit clauses from given positive facts using the least generalization algorithm. The absorption constructs non-unit clauses from unit clauses obtained by the truncation. A new predicate symbol is introduced by the intra-construction. CIGOL requires a more powerful oracle than that we introduced in the previous section. For example, the oracle can answer correctness of a general clause such as

$$reverse([X|Y],Z) \leftarrow reverse(Y,Y_1), concat(X,Y_1,Z).$$

in a target model. Furthermore, although CIGOL invents predicate symbols, it does not invent predicates. That is, assigning a model to a newly invented predicate symbol is left to the oracle.

The following two problems should be considered for introducing new predicates in the extended model inference.

1. When (or why) is it necessary to introduce a new predicate symbol?
2. How to define the model assigned to the new predicate symbol.

Although CIGOL avoids the second problem by oracle power, the problem is apparently most essential in extended model inference.

The first problem is concerned with the convergency of an inference algorithm. For example, CIGOL introduces a new predicate symbol in inferring a model

$$M = \{arch(t) \mid t \text{ is a triplet } (t_1, beam, t_1) \text{ and } t_1 \text{ is a list (possibly empty) which consists of } block \text{ and } brick\}.$$

The new predicate symbol corresponds to a predicate $column(\_)$ whose intended model is $\{column(t) \mid t \text{ is a list consists of } block \text{ and } brick\}$. Of course, the predicate $arch(\_)$ can be defined using $column(\_)$ as $arch((X, beam, X)) \leftarrow column(X)$. However, there exists a program P such that $M(P) = M$ and P:

$$P = \left\{ \begin{array}{l} arch(([block|X], beam, [block|X])) \leftarrow arch((X, beam, X)). \\ arch(([brick|X], beam, [brick|X])) \leftarrow arch((X, beam, X)). \\ arch(([], beam, [])). \end{array} \right\}$$

which consists of only the predicate symbol $arch(\_)$. In order for the inference algorithm to converge to some program, it should be very careful to introduce new predicates. If an inference algorithm introduces new predicates immoderately, then it might diverge. In a setting in which an inference algorithm infers not only a target model but also a language for describing the model, it becomes difficult to ensure the convergency of the inference process.

On the other hand, Banerji proposed a procedure called DREAM that produces a new predicate [Ban88]. DREAM works as follows. Suppose that there exist two clauses $p \leftarrow A, D$ and $p \leftarrow B, D$ in the current hypothesis, where D is a set of atoms appearing in both bodies in common, and A and B are the sets of left atoms of the two bodies after removing the common atoms. Then, DREAM replaces the two clauses by the following three clauses:

$$\begin{array}{l} p \leftarrow new(t_1, \ldots, t_n), D. \\ new(X_1, \ldots, X_n) \leftarrow A'. \\ new(X_1, \ldots, X_n) \leftarrow B'. \ , \end{array}$$

where $t_1, \ldots, t_n$ are the terms in A and B, A' and B' are atoms obtained from A and B by replacing t_i with X_i at every occurrence, and $new(\_, \ldots, \_)$ is a new predicate symbol.

This method answers very clearly for the second problem we mentioned above. The model of $new(\_,\ldots,\_)$ is strictly defined according to the models of A' and B' that are already known. However, for the first problem, the method seems to be problematic. Actually, as Banerji noticed in his paper [Ban88], this kind of new predicates may be useful for simplifying programs. However, it is not necessary to introduce the new predicate. The predicate p had been already defined by the clauses $p \leftarrow A, D$ and $p \leftarrow B, D$.

Ling classified predicates to be invented into two categories called *necessary intermediate terms* and *useful intermediate terms* [Lin89a]. A necessary intermediate term is a predicate which is necessary to describe the target program. For example, the predicate such as $concat(\_,\_,\_)$ in *reverse* program is a necessary intermediate term. On the other hand, a useful intermediate term is a predicate which is useful for simplifying conjectures, but not necessary for describing the target program. For example, both predicates, $column(\_)$ introduced by CIGOL and $new(\_,\ldots,\_)$ introduced by DREAM, are useful intermediate terms. As mentioned in the introduction of this chapter, our interest is in invention of necessary intermediate terms.

5.3 A Simple Approach to the Problems

It seems very difficult to develop a general strategy, which has plausible answers to the two problems mentioned in the previous section, for introducing necessary intermediate terms. However, if each model in the target class to be inferred can be represented by a fairly restrictive program, it may be possible to develop a plausible strategy for introducing necessary terms in inferring the class. In this section, we consider what kind of restrictions on the program is preferable for such a strategy.

5.3.1 When a new predicate is necessary

First, we consider the situation in which a new predicate should be introduced. Let M be a target model. In such a situation, the current conjecture must be inconsistent with at least one fact known so far, for a new predicate becomes necessary. There are two possible cases. One is when the conjecture P is too strong, that is, $\alpha \in M(P)$ for some known negative

fact $\alpha \notin M$. The other is when P is too weak, that is, $\alpha \notin M(P)$ for some known positive fact $\alpha \in M$. It follows from Corollary 3.3 that, for any too strong P, P has a false clause in M. Hence, when P is too strong then it suffices to remove the false clause. Thus, a new predicate is not necessary in this case.

On the other hand, when P is too weak then there exists some ground atom $\beta \in M$ (possibly, $\beta = \alpha$) such that β is not covered by any clause in P with respect to M. In such a case, a clause which covers β with respect to M should be added to P. A new predicate becomes necessary when there is no clause which consists of known predicates and covers β with respect to M.

Thus, in order to launch into the introduction of a new predicate, the inference system must be able to examine all the candidate clauses and recognize the point when such clauses are exhausted. Hence, it is preferable that, for any positive fact α, the number of all possible clauses which covers α in a target model is finitely bounded. Several types of restrictions for bounding the number of possible clauses can be considered. At least the number of atoms allowed to appear in the body and the depth of terms in the atoms should be bounded in advance. It is preferable that the number or the depth are bounded as small as possible for the efficiency of the inference algorithm.

5.3.2 The model of a new predicate

As mentioned in the previous subsection, a new predicate at first appears as an atom in the body of a clause which covers a positive fact. The model of the new predicate should closely be related to the models of other predicates in the clause. By using this property, we may define the model of the new predicate. For example, suppose that a new predicate $new(\_, \_, \_)$ is introduced with a clause

$$reverse([X|Y], Z) \leftarrow reverse(Y, Y_1), new(X, Y_1, Z).$$

Then, the model of $new(\_, \_, \_)$:

$$\{new(t_1, t_2, t_3) \mid reverse([t_1|t_4], t_3) \in M \text{ and } reverse(t_4, t_2) \in M\}.$$

can be defined in terms of the model M of $reverse(\_, \_, \_)$:

$$\{reverse(t_1, t_2) \mid t_2 \text{ is the reversed list of } t_1\}.$$

Thus, the model is concerned with M through the shared variables in the clause.

In order for this simple method to work well, the following items should have been fixed or bounded as tight as possible.

(a) The arity of a predicate symbol to be introduced.

(b) The possible form of atoms occurring in a clause.

(c) The possible form of terms occurring in the atoms.

It is difficult to fix these items when general programs are targeted. Conversely, if we restrict the target to the class of programs for which the above matter can be fixed, then it may be possible to develop a plausible strategy for introducing new predicates and solving an extended model inference problem for the class of models described by the programs. In the next section, we give some examples of such classes of programs.

5.4 Examples of Restricted Programs

In this section, we give some examples of the class of programs which have considerably restricted syntax, and discuss the possibility of predicate invention in the extended model inference.

5.4.1 DRLP

A *deterministic regular logic program* (DRLP, for short) is a program which is equivalent to a deterministic finite automaton. Thus, the class of models described by DRLP's, called regular models, is equivalent to the class of regular languages. A DRLP consists of clauses of the following two forms:

$$q_i([a|X]) \leftarrow q_j(X).$$
$$q_i([]).$$

Furthermore, for each predicate symbol $q_i(\_)$ and each constant symbol a ($\neq []$), a DRLP has at most one clause whose head is $q_i([a|X])$.

A DRLP is so restrictive as to satisfy the requirement mentioned in the previous section. The model of a new predicate $q_j(\_)$ introduced by a clause $q_i([a|X]) \leftarrow q_j(X)$ can be defined

as $\{q_j(w) \mid q_i([a|w]) \in M\}$, where w denotes a ground list. In Chapter 6, we will discuss in detail the extended model inference of regular models.

5.4.2 LMLP

A *linear monadic logic program* (LMLP, for short) is a program which is equivalent to a deterministic tree automaton. A LMLP consists of clauses of the following two forms:

$$q_i(f(X_1, \ldots, X_n)) \leftarrow q_{j_1}(X_1), \ldots, q_{j_n}(X_n).$$
$$q_i(a).$$

where $x_1, \ldots, x_n$ are mutually distinct variables. That is, the class of LMLP's is a natural extension of that of DRLP's. Skakibara showed that the extended model inference problem for the class of models of LMLP's can efficiently be solvable [Sak90].

Since the syntax of LMLP's is fixed, they also satisfy the requirement mentioned in the previous section. Furthermore, the atoms appearing in the body do not share a variable. Thus, as in the case of DRLP, it seems that the model of a new predicate which is introduced by a clause

$$q_i(f(X_1, \ldots, X_n)) \leftarrow q_{j_1}(X_1), \ldots, q_{j_n}(X_n).$$

can be defined as

$$\{q_{j_k}(t_k) \mid q_i(t_1, \ldots, t_k, \ldots, t_n) \in M\}.$$

However, for a LMLP, the uniqueness of a clause according to its structure is not assumed. Hence, a LMLP is allowed to contain the following clauses:

$$q_1(X \vee Y) \leftarrow q_1(X), q_2(Y).$$
$$q_1(X \vee Y) \leftarrow q_1(X), q_3(Y).$$

In such a case, it is impossible to distinguish the model of $q_2(\_)$ from the model of $q_3(\_)$. Thus, for the programs containing such clauses, that is, for the programs which essentially have OR-parallelism, the simple way of defining the model may not work well. Actually, the efficient extended model inference algorithm by Sakakibara is based on a special strategy.

5.4.3 SDG

A LMLP has no shared variable in the body of each clause but may have clauses with a common head. Conversely, a program introduced here has no other clauses than those which have a common head but may have a shared variable in the body of a clause. This class corresponds to that of *simple deterministic grammars* (SDG, for short). An SDG is a context-free grammar $G = (N, \Sigma, P, S)$ such that

$$\text{if both } A \to a\alpha \text{ and } A \to a\beta \text{ are production rules in } P \text{ then } \alpha = \beta. \tag{5.1}$$

A language generated by an SDG is called a simple deterministic language (SDL, for short).

It is well known that every context-free grammar can be represented by a definite clause grammar which is a kind of logic program. A definite clause grammar corresponding to an SDG, denoted by SDDCG, consists of the following three types of clauses:

$$\begin{aligned} &A([a|X], Y) \leftarrow B(X, Z), C(Z, Y). \\ &A([a|X], Y) \leftarrow B(X, Y). \\ &A([a|X], X). \ , \end{aligned}$$

where each predicate name corresponds to a nonterminal of the SDG and a is a terminal symbol of the SDG. Each predicate $N(x, y)$ is interpreted as a predicate which is true if and only the string represented by the *differential list* $x - y$[1] can be generated from the nonterminal N in the SDG. From the condition (5.1) on the production rules, for each predicate symbol $A(\_, \_)$ and each terminal symbol a, there exists at most one clause whose head is an instance of $A([a|X], Y)$. Thus, the problem as mentioned in the previous subsection does not occur.

For SDDCG's, however, a new problem arises because a clause of the first type shares a variable in its body. For example, the models MB and MC of the predicate symbols $B(\_, \_)$ and $C(\_, \_)$ can be defined respectively as follows:

$$\begin{aligned} MB &= \{B(t_1, t_2) \mid A([a|t_1], t_3) \in M, C(t_3, t_2) \in MC\}, \\ MC &= \{C(t_2, t_2) \mid A([a|t_3], t_2) \in M, B(t_3, t_1) \in MB\}. \end{aligned}$$

Since each model is defined mutually by its alternative's, the definitions are problematic.

[1] For example, a differential list $[a, b, c, d] - [c, d]$ represents the list $[a, b]$.

Thus, for a program which needs a clause with shared variables in the body, when two or more predicates are necessary to be introduced simultaneously by one clause, a problem how to define the models of both predicate arises. For general programs, the problem may be intractable. Fortunately, however, for an SDG $G = (N, \Sigma, P, S)$,it follows from the condition 5.1 that

$$\text{if } A \Rightarrow^* w\alpha \text{ and } A \Rightarrow^* w\beta \ (A \in N,\ w \in \Sigma^+,\ \beta \in N^*) \text{ then } \alpha = \beta.$$

Hence, for a clause of the form $A([a|X], Y) \leftarrow B(X, Z), C(Z, Y)$, if some positive fact $B([w|u], u)$[2] is found, then the model MC of $C(\_, \_)$ can uniquely be defined by M as follows:

$$MC = \{C(t_1, t_2) \mid A([aw|t_1], t_2) \in M\}.$$

A positive fact about $B(\_, \_)$ can be obtained as $B([w|u], u)$ from a positive fact $A([ax|y], y)$ such that $xy = wu$ and w is a prefix of x.

In Chapter 7, we will discuss in detail the extended model inference problem for this class. The algorithm described in there will use the above property and achieve an efficient inference.

5.4.4 Simple recursive programs

Since each program introduced above is a direct representation of an automaton or a grammar, there exists a special pattern on the structure of a clause or the arity of a predicate symbol appearing in the clause. It may be impossible to expect a class of general programs to have such a pattern. However, it may also be a fact that we often make a program by repeating some stereotyped patterns. Here, we see such patterns found in some simple programs called *term-free transformations* introduced by Shapiro [Sha81].

Multiplication:

$$times(0, X, 0).$$
$$times(s(X), Y, Z) \leftarrow times(X, Y, U), plus(Y, U, Z).$$
$$plus(0, X, X).$$
$$plus(s(X), Y, s(Z)) \leftarrow plus(X, Y, Z).$$

[2] As in the next chapter, we abbreviate a list $[a_1, \ldots, a_n|[b_1, \ldots, b_m]]$ as $[w|u]$ where $w = a_1 a_2 \cdots a_n$ and $u = b_1 b_2, \cdots b_m$.

Subset relation:

$$subset([], X).$$
$$subset([X|Y], Z) \leftarrow subset(Y, Z), member(X, Z).$$
$$member(X, [X|Y]).$$
$$member(X, [Y|Z]) \leftarrow member(X, Z).$$

Insertion sort:

$$sort([], []).$$
$$sort([X|Y], Z) \leftarrow sort(Y, Y_1), insert(X, Y_1, Z).$$
$$insert(X, [], [X]).$$
$$insert(X, [Y|Z], [X, Y|Z]) \leftarrow X \leq Y.$$
$$insert(X, [Y|Z], [Y|Z_1]) \leftarrow Y \leq X, insert(X, Z, Z_1).$$
$$X \leq X.$$
$$X \leq s(Y) \leftarrow X \leq Y.$$

The list reversal program described in the introduction of this chapter is also a term-free transformation. There are following properties common to these programs.

(1) There is at most one auxiliary predicate in the body of each clause.

(2) All arguments of the atoms appearing in the bodies are variables.

(3) No free-variable appears in the body, where a free-variable is a variable which appears at most once only in the body of a clause.

(4) Each head has no common instance with other head.

These properties seem to be preferable for clearing the problems mentioned before. From the property (1), we do not need to worry if several new predicates are introduced simultaneously by one clause. By the propety (4), we can clear the problem mentioned in Section 5.4.2. By the properties (2) and (3), we can restrict (b) and (c) in Section 5.3.2. Furthermore, there is one more typical property common to the above programs, which may be useful for deciding the arity of predicate to be introduced.

(5) Each variable processed recursively does not occur in each auxiliary predicate.

For example, the variable x in $times(s(x), y, z)$ does not occur in $plus(y, u, z)$.

Thus, there may exist a plausible strategy to solve the extended model inference problem such as in Example 1.1. It is a very interesting and challenging theme to develop such a strategy.

In the following two chapters, we shall consider some concrete extended model inference problems. One is concerned with inferring DRLP's and the other with inferring SDG's. In both cases, we shall develop efficient inference algorithms. The existence of an effective method for introducing predicate symbols and assigning appropriate models to the symbols is essential for the efficiency.

Chapter 6

Learning Regular Languages

In this chapter, we consider a problem of inferring a class of restricted logic programs that corresponds to the class of acceptors for regular languages. In our setting, a target logic program is over a first order language $\mathcal{L}$ with countably many unary predicate symbols: $q_0, q_1, q_2, \ldots$. A given oracle is that for a model M_0 over $\mathcal{L}_0$, the restricted language of $\mathcal{L}$ in which only one predicate symbol q_0 is allowed. As mentioned in the previous chapter, in such a setting, the oracle has no interpretation for predicates other than the predicate q_0. This implies that we cannot take advantage of the contradiction backtracing algorithm which is one of the most important part for the efficiency of Shapiro's model inference algorithm.

In order to overcome the disadvantage, we develop a method for giving an interpretation for predicates other than the predicate q_0, which is based on the idea of using the oracle for M_0 and a one to one mapping from a set of predicates to a set of strings. Furthermore, we propose a model inference algorithm for regular languages using the method, then discuss the correctness and the time complexity of the algorithm.

In Section 6.1, we define the problem of learning regular languages as an extended model inference problem. In Section 6.2, we propose a key idea to solve the extended model inference problem. We introduce a simple mapping, called a predicate characterization, by which a model over $\mathcal{L}_0$ can be extended to a model over $\mathcal{L}$. Then, we show some conditions which the mapping should satisfy to perform an appropriate extension of the model. In Chapter 6.3, we give an inference algorithm to solve the extended model inference problem. The correctness of the algorithm is shown in Chapter 6.4, and time analysis of the algorithm

is made in Chapter 6.5.

This chapter is based on the paper [Ish88a].

6.1 Regular Model Inference Problem

We introduce a class of Herbrand models called *regular models* and show that the class is equivalent to that of regular languages. Then a regular model inference problem, which is an extended model inference problem, is defined.

Let $\mathcal{L}$ be a first order language with countably many unary predicate symbols q_0, $q_1, \ldots$, a list constructor $[\_|\_]$, and finitely many constant symbols $[]$, a_1, a_2, $\ldots$, a_m. Since every predicate symbol treated in this chapter is unary, we shall identify each predicate symbol $q_i(\_)$ with its predicate name q_i for notational convenience. $\mathcal{L}_0$ denotes the restricted language of $\mathcal{L}$ in which only one predicate symbol q_0 is allowed. Let P be a logic program over $\mathcal{L}$. $\mathcal{L}_P$ denotes the language with only predicate, function and constant symbols occurring in P, $\Pi(P)$ denotes the set of the predicate symbols.

Let Σ denote the set $\{a_1, a_2, \ldots, a_m\}$. A string $x = a_{i_1} a_{i_2} \cdots a_{i_n} \in \Sigma^*$ is denoted using list notation by $[a_{i_1}, a_{i_2}, \ldots, a_{i_n}]$. Furthermore, the list $[a_{i_1}, a_{i_2}, \ldots, a_{i_n}]$ is abbreviated as $[a_{i_1} a_{i_2} \cdots a_{i_n}]$. Thus the string x is represented as $[x]$ in the following context of logic programs. Note that, for the empty string, $[\varepsilon]$ is denoted by the empty list $[]$. For a program P over $\mathcal{L}$, $M(P)_{q_i}$ denotes the set $\{q_i([x]) \in M(P) \mid x \in \Sigma^*\}$.

Definition 6.1 A *deterministic regular logic program* (DRLP, for short) P is the logic program over $\mathcal{L}$ which satisfies the following conditions.

1) Each clause in P is of one of the following two forms:

$$q_i([a_k|X]) \leftarrow q_j(X),$$
$$q_i([]).$$

2) For any $q_i \in \Pi(P)$ and $a_k \in \Sigma$, there is at most one clause in P whose head is $q_i([a_k|X])$.

Definition 6.2 Let $M_0 \subseteq \mathcal{B}_{\mathcal{L}_0}$. We say M_0 is a *regular model* if there exists a DRLP P such that $M_0 = M(P)_{q_0}$

Let $L(M_0)$ denote the set $\{x \in \Sigma^* \mid q_0([x]) \in M_0\}$. Then we have the following theorem which ensures that the class of regular models is equivalent to that of regular languages.

Theorem 6.1 *For any $M_0 \subseteq \mathcal{B}_{\mathcal{L}_0}$, M_0 is a regular model if and only if $L(M_0)$ is a regular language.*

Proof: First, we prove the *only if* direction. Let P be a DRLP such that $M(P)_{q_0} = M_0$. Without loss of generality, we may assume that $\Pi(P) = \{q_0, q_1, \ldots, q_m\}$. Then, construct a DFA $A = (Q \cup \{q_{m+1}\}, \Sigma, \delta, q_0, F)$, where $Q = \Pi(P)$, q_{m+1} is a predicate symbol which does not appear in $\Pi(P)$, $F = \{q_i \in Q \mid q_i([]) \in P\}$, and

$$\delta(q_i, a_k) = \begin{cases} q_j & \text{if} \quad q_i([a_k|X]) \leftarrow q_j(X) \in P, \\ q_{m+1} & \text{otherwise.} \end{cases}$$

Here, we show that $\delta(q_i, x) \in F$ if and only if $q_i([x]) \in M(P)$ for any $q_i \in \Pi(P)$ and $x \in \Sigma^*$ by induction on the length of x.

If $|x| = 0$, that is, $x = \varepsilon$ then it follows from the definition of A that

$$\delta(q_i, \varepsilon) = q_i \in F \quad \text{iff} \quad q_i([]) \in P \tag{6.1}$$

From Corollary 3.3, if $q_i([]) \in P$ then $q_i([]) \in M(P)$. Also, if $q_i([]) \in M(P)$ then there exists a clause C in P such that $q_i([]) \in C(M(P))$. Hence, it follows from the definition of DRLP's that C is of the form $q_i([])$. Thus it holds that $q_i([]) \in P$ if and only if $q_i([]) \in M(P)$. With (6.1), this implies that $\delta(q_i, \varepsilon) \in F$ if and only if $q_i([\varepsilon]) \in M(P)$.

Suppose that $\delta(q_i, a) = q_j$. Since $\delta(q_i, ax) = \delta(\delta(q_i, a), x)$, it holds that $\delta(q_i, ax) \in F$ if and only if $\delta(q_j, x) \in F$. From the induction assumption, it holds that $\delta(q_j, x) \in F$ if and only if $q_j([x]) \in M(P)$ for any $q_j \in \Pi(P)$ and any string $x \in \Sigma^*$ such that $|x| \leq n$ for some $n \geq 0$. On the other hand, from the definition of A, it holds that, for any $a \in \Sigma$ and $q_i \in \Pi(P)$,

$$\delta(q_i, a) = q_j \quad \text{iff} \quad q_i([a|X]) \leftarrow q_j(X) \in P.$$

Thus, it holds that

$$\delta(q_i, ax) \in F \quad \text{iff} \quad q_i([a|X]) \leftarrow q_j(X) \in P \quad \text{and} \quad q_j(x) \in M(P).$$

From Corollary 3.3, if $q_i([a|X]) \leftarrow q_j(X) \in P$ and $q_j(x) \in M(P)$, then $q_i([ax]) \in M(P)$. Also, if $q_i([ax]) \in M(P)$, then there exists a clause C in P such that $q_i([ax]) \in C(M(P))$. It follows from the definition of DRLP's that such C is unique if it exists. Hence, if $q_i([ax]) \in M(P)$ then there uniquely exists a clause $q_i([a|X]) \leftarrow q_j(X)$ in P such that $q_j(x) \in M(P)$. Thus, it holds that

$$\delta(q_i, ax) \in F \quad \text{iff} \quad q_i([ax]) \in M(P).$$

This completes the induction step of the proof. Hence, it holds that $\delta(q_i, x) \in F$ if and only if $q_i([x]) \in M(P)$ for any $q_i \in \Pi(P)$ and $x \in \Sigma^*$. As a result, we obtain the following relation:

$$\delta(q_0, x) \in F \quad \text{iff} \quad q_0([x]) \in M(P)_{q_0} \quad \text{iff} \quad q_0([x]) \in M_0.$$

Thus $L(M_0)$ is a regular language.

Conversely, from a DFA which accepts $L(M_0)$, we can construct a DRLP and show that the other direction in a similar fashion. □

Here we define the problem of learning regular languages as an extended model inference problem. Let M_0 ($\subseteq \mathcal{B}_{\mathcal{L}_0}$) be a regular model. An *oracle* for M_0 over $\mathcal{L}_0$ is the device which, for any input $\alpha \in \mathcal{B}_{\mathcal{L}_0}$, returns *true* if $\alpha \in M_0$, *false* otherwise. *Facts* about M_0 are pairs of the form $\langle \alpha, V \rangle$, where $\alpha \in \mathcal{B}_{\mathcal{L}_0}$ and $V \in \{true, false\}$ is the output value of an oracle for M_0 on an input α. Ground atoms in M_0 are called *positive facts*, while others *negative facts*. An *enumeration of* M_0 is an infinite sequence: $F_1, F_2, F_3, \ldots$, where each F_i is a fact about M_0 and every $\alpha \in \mathcal{B}_{\mathcal{L}_0}$ occurs in a fact $F_i = \langle \alpha, V \rangle$ for some $i \leq 1$. We assume the oracle for M_0 can give any enumeration of M_0 to an inference algorithm.

The main problem considered in this chapter is as follows.

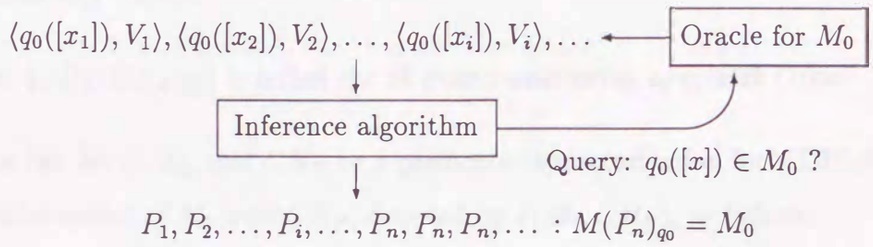
> Suppose an oracle for some unknown regular model M_0 over $\mathcal{L}_0$ is given. Find a DRLP P such that $M(P)_{q_0} = M_0$.

In this chapter, the inference algorithm is allowed to use membership queries about a target regular model M_0. A membership query about M_0 is to propose a ground atom $\alpha \in \mathcal{B}_{\mathcal{L}_0}$ and ask the oracle if $\alpha \in M_0$. An inference process of the algorithm is as follows. At each time, the algorithm reads one fact $\langle \alpha_i, V_i \rangle$ from a given enumeration of the target model

M_0. Then it makes finitely many membership queries about M_0. According to answers from the oracle, the algorithm produces a DRLP P_i as a conjecture.

Although, the almost all definitions concerned with the behavior of an inference algorithm are same as those defined in Chapter 2, we need to redefine the notion of identifiability. An inference algorithm is said to *identify a regular model M_0 in the limit* if it converges on every enumeration of M_0 given by the oracle to a conjecture P such that $M(P)_{q_0} = M_0$. Figure 6.1 illustrates the framework of the regular model inference.

Figure 6.1: The framework of the regular model inference



In the framework of Shapiro's model inference [Sha81, Sha82], a first order language $\mathcal{L}$ with finitely many predicate symbols is given in advance. The Model Inference System (MIS, for short) based on the model inference algorithm requires an oracle for a model M over $\mathcal{L}$. Then MIS efficiently synthesizes a logic program P over $\mathcal{L}$ such that $M(P) = M$. The model inference algorithm is equipped with the contradiction backtracing algorithm as a sub-algorithm. The sub-algorithm plays the most important part for the efficiency of the model inference algorithm. With the help of the given oracle, the contradiction backtracing algorithm finds out a wrong clause in a hypothesis unsuitable for a conjecture by examining falsity of clauses in the hypothesis. The examination is executed by making membership queries about M, that is, by proposing a ground atom $\alpha \in \mathcal{B}_\mathcal{L}$ and asking if $\alpha \in M$. Note that, in the setting, hypotheses are logic programs over $\mathcal{L}$ and a target model M is a Herbrand model over the same language $\mathcal{L}$.

In our setting, however, the given oracle cannot answer membership queries for $\alpha \in \mathcal{B}_\mathcal{L} - \mathcal{B}_{\mathcal{L}_0}$. Thus, the oracle cannot be used directly to examine the falsity of clauses which contain a predicate symbol except q_0. In order to construct an efficient regular model inference algorithm, we should develop some method for examining falsity of clauses over $\mathcal{L}_P$ for any

hypotheses P. In the next section, we shall argue such a method.

6.2 An Extended Model of a Regular Model

We introduce a mapping called a *predicate characterization*. With the mapping satisfying some conditions, we can appropriately extend a regular model over $\mathcal{L}_0$ to a model over $\mathcal{L}_P$ for any DRLP P.

Definition 6.3 A *predicate characterization for a DRLP* P, denoted by CH_P, is a one to one mapping from $\Pi(P)$ to Σ^*.

For any $q_i \in \Pi(P)$, $CH_P(q_i)$ is called the *characteristic string of* q_i *with* CH_P.

Definition 6.4 Let $M_0 \subseteq \mathcal{B}_{\mathcal{L}_0}$ and CH_P be a predicate characterization for a DRLP P. We define an *extended model of* M_0 *with* CH_P, denoted by $I(M_0, CH_P)$, as follows:

$$I(M_0, CH_P) = \{q_i([x]) \in \mathcal{B}_{\mathcal{L}_P} \mid q_0([CH_P(q_i) \cdot x]) \in M_0\},$$

With the above extension of a model over $\mathcal{L}_0$, for the present, we can get a model over $\mathcal{L}_P$. However, it is nonsense that a model over $\mathcal{L}_0$ is arbitrarily extended with a haphazard predicate characterization. The extension should satisfy the following condition:

$$M(P)_{q_0} = M_0 \quad \text{iff} \quad M(P) = I(M_0, CH_P).$$

We show that some restrictions on CH_P lead to such an extension.

Definition 6.5 Let CH_P be a predicate characterization for a DRLP P. The CH_P is said to be *consistent* if, for any $q_i \in \Pi(P)$, there exists a derivation tree of $q_0([CH_P(q_i)])$ on P in which $q_i([])$ appears.

Note that, for any DRLP P and any ground atom $q_i([x]) \in \mathcal{B}_{\mathcal{L}_P}$, any possible derivation tree of α on P is unique and is of the very simple form as in Figure 6.2. The above definition is translated in terms of the theory of finite-state automata as follows. Let P be a DFA with a transition function δ, CH_P is consistent if and only if $\delta(q_0, CH_P(q_i)) = q_i$ for any state q_i in P.

Figure 6.2: A derivation tree on a DRLP

$$
\begin{array}{c}
q_i([a_1a_2a_3\cdots]) \\
| \\
q_{i_1}([a_2a_3\cdots]) \\
| \\
q_{i_2}([a_3\cdots]) \\
| \\
\vdots
\end{array}
$$

Lemma 6.2 *For any DRLP P, if a predicate characterization CH_P for P is consistent, then it holds that $M(P) = I(M(P)_{q_0}, CH_P)$.*

Proof: From the uniqueness of the derivation tree on a DRLP and the consistency of CH_P, for any $q_i \in \Pi(P)$ and $x \in \Sigma^*$, it holds that

$$P \vdash q_i([x]) \quad \text{iff} \quad P \vdash q_0([CH_P(q_i)\cdot x]).$$

Since $P \vdash \alpha$ if and only if $\alpha \in M(P)$, it holds that

$$\begin{aligned}
q_i([x]) \in M(P) \quad &\text{iff} \quad q_0([CH_P(q_i)\cdot x]) \in M(P) \\
&\text{iff} \quad q_0([CH_P(q_i)\cdot x]) \in M(P)_{q_0} \\
&\text{iff} \quad q_i([x]) \in I(M(P)_{q_0}, CH_P).
\end{aligned}$$

□

Theorem 6.3 *Suppose CH_P for a DRLP P is consistent and $CH_P(q_0) = \varepsilon$. Then, for any $M_0 \subseteq \mathcal{B}_{\mathcal{L}_0}$, $M(P)_{q_0} = M_0$ if and only if $M(P) = I(M_0, CH_P)$.*

Proof: Since the *only if* direction immediately follows from Lemma 6.2, it is sufficient to prove the *if* direction. For any $x \in \Sigma^*$, it holds that

$$\begin{aligned}
q_0([x]) \in M_0 \quad &\text{iff} \quad q_0([CH_P(q_0)\cdot x]) \in M_0 \qquad (\text{from } CH_P(q_0) = \varepsilon) \\
&\text{iff} \quad q_0([x]) \in I(M_0, CH_P) \\
&\text{iff} \quad q_0([x]) \in M(P) \qquad (\text{from the assumption}) \\
&\text{iff} \quad q_0([x]) \in M(P)_{q_0}.
\end{aligned}$$

□

By the above theorem, the regular model inference problem can be restated as follows.

Suppose an oracle for some unknown regular model M_0 is given. Find a DRLP P such that $M(P) = I(M_0, CH_P)$, where CH_P is consistent and $CH_P(q_0) = \varepsilon$.

With such a predicate characterization, it is possible to take advantage of the contradiction backtracing algorithm. Whenever the algorithm needs information about membership of some ground atom $q_i([x]) \in \mathcal{B}_{\mathcal{L}_P}$ in an extended model $I(M_0, CH_P)$, it can get the information by making a query "$q_0([CH_P(q_i) \cdot x]) \in M_0$?" to the given oracle.

6.3 A Regular Model Inference Algorithm

In this section, first we show that every clause in a DRLP P has a property called completeness with respect to $M(P)$. Then we give a regular model inference algorithm in which an extended model of a target regular model is used for detecting incomplete clause in unsuitable hypotheses with respect to the extended model.

Definition 6.6 Let $M \subseteq \mathcal{B}_{\mathcal{L}}$ and C be a clause. Then C is said to be *sufficient in* M if, for every $\alpha \in M$ such that $head(C) \succeq \alpha$, it holds that $\alpha \in C(M)$.

Definition 6.7 A clause C is said to be *complete in* M if C is both true and sufficient in M.

The following proposition directly follows from the above definitions.

Proposition 6.4 *The following two statements are equivalent.*

1. *A clause* $q_i([a|X]) \leftarrow q_j(X)$ *is complete in a model* M.
2. *For any* $x \in \Sigma^*$, $q_i([ax]) \in M$ *if and only if* $q_j([x]) \in M$.

For any DRLP P and $\alpha \in M(P)$, since there exists at most one clause which covers α in $M(P)$, each clause in P is sufficient in M. Thus we have the following proposition.

Proposition 6.5 *For any DRLP* P, *every clause in* P *is complete in* $M(P)$.

With Lemma 6.2, this implies the following proposition.

Proposition 6.6 *Let* CH_P *be a consistent predicate characterization for a DRLP* P. *If* $M(P)_{q_0} = M_0$ *then every clause in* P *is complete in* $I(M_0, CH_P)$.

By the above proposition, if there is a clause in P which is not complete in $I(M_0, CH_P)$ for some CH_P, then $M(P)_{q_0} \neq M_0$. Hence, whenever a clause which is not complete in $I(M_0, CH_P)$ exists in a hypothesis P, then the clause must be eliminated from P. Thus in our setting, the clause that are not sufficient in an extended model are removed from an unsuitable hypothesis, while, in Shapiro's model inference algorithm, only the clauses that are not true in a target model are removed.

Algorithm 6.1: A regular model inference algorithm

Given: An oracle for a regular model M_0 over $\mathcal{L}_0$.
Input: An enumeration about M_0.
Output: A sequence of DRLP's.
Procedure:
$P := \phi$; $CH_P := \{(q_0, \varepsilon)\}$; $S_{true} := \phi$; $S_{false} := \phi$; $State := 0$;
repeat
read the next fact $\langle \alpha, V \rangle$; $S_V := S_V \cup \{\alpha\}$;
repeat
while there exists $\alpha \in S_{false}$ such that $\alpha \in M(P)$ **do**
let PT_α be the proof tree of α on P;
$C :=$ **contradiction_backtracing**(PT_α);
$P := P - \{C\}$;
$C' :=$ **next_clause**(C);
$P := P \cup \{C'\}$;
while there exists $\beta \in S_{true}$ such that $\beta \notin M(P)$ **do**
$\beta' :=$ **uncovered_atom**(β);
$C :=$ **search_clause**(β');
$P := P \cup \{C\}$;
until neither of the **while** loop is entered;
output P;
forever.

Now we state an outline of the algorithm mainly concentrating on the following two:

- how to modify hypotheses,
- how to construct a predicate characterization.

For the present, we assume that, at any time, the predicate characterization CH_P constructed by the algorithm is consistent and $CH_P(q_0) = \varepsilon$.

Algorithm 6.2: Sub-procedures for modifying a too strong hypothesis

contradiction_backtracing:
Given: An oracle for a regular model M_0 over $\mathcal{L}_0$.
Input: A proof tree of an atom $q_i([ax])$ on P such that
$q_i([ax]) \in M(P)$ but $q_i([ax]) \notin I(M_0, CH_P)$.
Output: A clause $C \in P$ which is false in $I(M_0, CH_P)$.
Procedure:
let $q_j([x])$ be the child of $q_i([ax])$ in the proof tree;
if $q_j([x]) \in I(M_0, CH_P)$ **then**
return $q_i([a|X]) \leftarrow q_j(X)$;
else
let PT be the proof tree of $q_j([x])$ on P;
/* Such a proof tree can be obtained from the input
proof tree by removing the root node */
return **contradiction_backtracing**(PT).

next_clause:
Input: A clause $q_i([a|X]) \leftarrow q_j(X)$.
Output: A clause $q_i([a|X]) \leftarrow q_{j+1}(X)$.
Procedure:
if $j = State$ **then**
$State := State + 1$;
let x be the string such that $(q_i, x) \in CH_P$;
$CH_P := CH_P \cup \{(q_{j+1}, xa)\}$;
return $q_i([a|X]) \leftarrow q_j(X)$.

There are following two cases in which a hypothesis P should be modified.

(1) The hypothesis is too strong, that is, $M(P)$ contains some negative fact (the first **while** loop in Algorithm 6.1).

(2) The hypothesis is too weak, that is, $M(P)$ does not contain some positive fact (the second **while** loop in Algorithm 6.1).

In the first case, there exists at least one clause in P which is not true in $I(M_0, CH_P)$ (this will be clear in the proof of Lemma 6.8). The algorithm finds such a clause using the contradiction backtracing algorithm (procedure **contradiction_backtracing**). The clause

is removed from the hypothesis. Then an alternate clause constructed by a clause generator (procedure **next_clause**) is added to the hypothesis.

In the procedures **contradiction_backtracing** and **uncovered_atom**, the examination of **if** statement "$q_j([x]) \in I(M_0, CH_P)$" is done by making a membership query proposing $q_0([CH_P(q_j)\cdot x])$. We do not consider the atom of the form $q_i([])$ as an input for **contradiction_backtracing**. The reason be described after Proposition 6.7.

Algorithm 6.3: Sub-procedures for modifying a too weak hypothesis

undovered_atom:
Input: An atom β such that $\beta \in I(M_0, CH_P)$ but $\beta \notin M(P)$.
Output: An atom in $I(M_0, CH_P)$ which is not covered by any clause in P with respect to $I(M_0, CH_P)$
Procedure:
if there exists a clause $q_i([a|X]) \leftarrow q_j(X) \in P$ such that $q_i([a|X])\theta = \beta$ for some substitution θ **then**
if $q_j(X)\theta \in I(M_0, CH_P)$ **then**
return **uncovered_atom**$(q_j(X)\theta)$;
else return β;
else return β.

search_clause:
Input: An uncovered atom $q_i([x])$ which is returned by the above procedure.
Output: A new clause C whose head is unifiable with $q_i([x])$.
Procedure:
if $x = \varepsilon$ **then**
return $q_i([])$;
else let $x = ax'$;
if there exists $C \in P$ such that $head(C) = q_i([a|X])$ **then**
$P := P - \{C\}$;
return **next_clause**(C);
else return $q_i([a|X]) \leftarrow q_0(X)$.

In the second case, there exists a ground atom $q_i([x]) \in I(M_0, CH_P)$ which is not covered by any clause in P with respect to $I(M_0, CH_P)$ (this will be clear in the proof of Lemma 6.9). The procedure **uncovered_atom** finds out such an atom. Note that when the procedure is

called with an input $q_i([])$, it always return the input. Because, if $q_i([]) \notin M(P)$, then it is immediately found that $q_i([])$ is not covered by any clause in P with respect to any model. For such an uncovered atom, the following two cases are possible.

(i) There is no clause in P whose head is unifiable with $q_i([x])$.

(ii) Although there is a clause C whose head is unifiable with $q_i([x])$, C does not cover $q_i([x])$ in $I(M_0, CH_P)$.

In the case (i), if $x = \varepsilon$, then the unit clause $q_i([])$ is added to the hypothesis, and if $x = aw$ ($a \in \Sigma$, $w \in \Sigma^*$), then the clause $q_i([a|X]) \leftarrow q_0(X)$ is added to the hypothesis. In the case (ii), according to Proposition 6.6, C is removed from the hypothesis, then an alternate clause is added to the hypothesis similarly in the case (1). The procedure **search_clause** works according to each case.

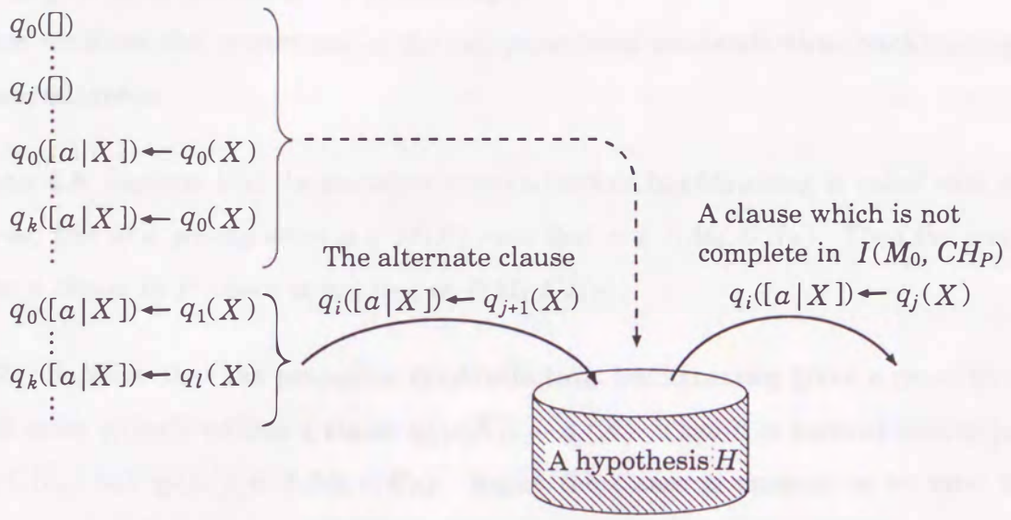
In the algorithm, the predicate characterization is represented as a set of pairs of a predicate symbol and a string. The set is constructed as follows. Let P be the current hypothesis and $CH_P = \{(q_0, \varepsilon), (q_1, x_1), (q_2, x_2), \ldots, (q_k, x_k)\}$ be the current predicate characterization. Now we assume that the algorithm finds out a clause $C = q_i([a|X]) \leftarrow q_k(X)$ in P which is not complete in $I(M_0, CH_P)$. Then P is modified in the way mentioned above. Hence, C is removed from P and an alternative constructed by the clause generator is added to P. Since the clause generator increases the index of q_k, the alternative has a new predicate symbol q_{k+1} which has never appeared in P before. When such a new predicate symbol is introduced, the algorithm adds a pair $(q_{k+1}, x_i a)$ to CH_P, that is, a characteristic string of the new predicate symbol is determined by the head of the clause which caused the introduction of the new predicate symbol.

Figure 6.3 illustrates the outline of modifying hypotheses mentioned above. The dotted line denotes the operation which is made in the case (2)-(i). The two solid lines denote the operations that are made simultaneously in the case (1) and (2)-(ii).

6.4 Correctness of the Algorithm

First, we give a simple proposition concerned with the property of predicate characterization. From the definition of an extended model, the following holds.

Figure 6.3: The outline of modifying hypotheses



Proposition 6.7 *Let P_1 and P_2 be any DRLP's such that $\Pi(P_1) \subseteq \Pi(P_2)$. Suppose that $CH_{P_1}(q) = CH_{P_2}(q)$ for any $q \in \Pi(P_1)$. Then, for any $M_0 \subseteq \mathcal{B}_{\mathcal{L}_0}$ and $q([x]) \in \mathcal{B}_{\mathcal{L}_{P_1}}$, $q([x]) \in I(M_0, CH_{P_1})$ if and only if $q([x]) \in I(M_0, CH_{P_2})$.*

The predicate characterization constructed by the algorithm changes with increase of its domain, the set of predicate symbols in hypotheses. This leads to the change of the extended model with the predicate characterization. However, from the way of constructing a predicate characterization, it is clear that the change of the predicate characterization caused by modifying hypotheses satisfies the premises of Proposition 6.7. Hence the change of the extended model with the predicate characterization satisfies the conclusion of Proposition 6.7. Thus, once a ground atom is true (false) in an extended model, the atom is still true (false) in subsequent models.

In Algorithm 6.2, we do not consider the atom of the form $q_i([])$ as an input for **contradiction_backtracing**. Because, at any time on the inference process, there is no case in which $q_i([])$ is in $M(P)$ but not in $I(M_0, CH_P)$. The ground atom $q_i([])$ is in $M(P)$ if and only if there exists a unit clause $q_i([])$ in P. The clause $q_i([])$ is added to P after the procedure **search_clause** is called on the input $q_i([])$. Let P' be the hypothesis for which the procedure call is occurred. Then $q_i([])$ is in $I(M_0, CH_{P'})$. By Proposition 6.7, $q_i([])$ is

ensured to be in $I(M_0, CH_P)$ for any subsequent hypothesis P. Hence, there is no case in which $q_i([])$ is in $M(P)$ but not in $I(M_0, CH_P)$.

Here we show the correctness of the sub-procedures **contradiction_backtracing** and **uncovered_atom**.

Lemma 6.8 *Suppose that the procedure* **contradiction_backtracing** *is called with the input proof tree of a ground atom $\alpha \in M(P)$ such that $\alpha \notin I(M_0, CH_P)$. Then the procedure returns a clause in P which is not true in $I(M_0, CH_P)$.*

Proof: Suppose that the procedure **contradiction_backtracing** given a proof tree of a ground atom $q_i([ax])$ returns a clause $q_i([a|X]) \leftarrow q_j(X)$. Then it is ensured that $q_i([ax]) \notin I(M_0, CH_P)$ but $q_j([x]) \in I(M_0, CH_P)$. Hence the clause is ensured to be false in the extended model $I(M_0, CH_P)$. Furthermore, since $q_j([x])$ is the child of $q_i([ax])$ in the input proof tree on P, the clause $q_i([a|X]) \leftarrow q_j(X)$ exists in P.

On the other hand, every input proof tree has the leaf $q_k([])$ for some $q_k([]) \in P$. From the discussion above, it is ensured that $q_k([]) \in I(M_0, CH_P)$. Since the input proof tree of each recursive call clear the input condition, a clause which is false in $I(M_0, CH_P)$ must be found eventually. □

Lemma 6.9 *Suppose that the procedure* **uncovered_atom** *is called with an input $\beta \in M_0$ such that $\beta \notin M(P)$. Then the procedure returns some ground atom $\beta' \in I(M_0, CH_P)$ such that β' is not covered by any clause in P with respect to $I(M_0, CH_P)$.*

Proof: Since the procedure examines if $q_j(X)\theta \in I(M_0, CH_P)$ before calling itself recursively, every input $q_j(X)\theta$ for its recursive call is ensured to be in $I(M_0, CH_P)$. On the other hand, if an input for its recursive call is in $M(P)$, then the input has a proof tree on P. This implies that all ancestors of the input have also proof trees on P. This contradicts that $\beta \notin M(P)$, that is, the initial input of the procedure has no proof tree on P. Thus every input for its recursive call is not in $M(P)$.

Since $q_i([])$ is not unifiable with any $q_i([a|X])$, the procedure called with an input of the form $q_i([])$ returns the input directly. Since $q_i([]) \notin M(P)$, it holds that $q_i([]) \notin P$. For any clause in a DRLP, a ground atom $q_i([])$ is covered only by the clause $q_i([])$. Thus, if the

procedure is called with the input $q_i([])$, then it immediately follows that $q_i([])$ is not covered by any clause in P with respect to $I(M_0, CH_P)$.

For an input of the form $q_i([ax])$, if there is no clause whose head is unifiable with $q_i([ax])$, it is clear that $q_i([ax])$ is not covered by any clause in P. Since a clause of the form $q_i([a|X]) \leftarrow q_{j+1}(X)$ is introduced into a hypothesis after the clause $q_i([a|X]) \leftarrow q_j(X)$ is removed from the hypothesis, there is at most one clause whose head is unifiable with $q_i([ax])$. Thus if $q_i([ax])$ is not covered such a clause, there is no other clause which can cover $q_i([ax])$ with respect to $I(M_0, CH_P)$. Thus, if the procedure make an output, then the output is ensured to satisfy the claim of the lemma.

On the other hand, the size of each input $q_j([X])\theta$ for the recursive call is decreasing one by one. Thus, even if in the worst case, the procedure will encounter an input of the form $q_i([])$ and terminate. This completes the proof of the lemma. □

Next, we show the justification of the way of constructing a predicate characterization. We can restate Proposition 6.7 as follows.

Proposition 6.10 *Let P_1 and P_2 be any DRLP's such that $\Pi(P_1) \subseteq \Pi(P_2)$. Suppose that $CH_{P_1}(q) = CH_{P_2}(q)$ for any $q \in \Pi(P_1)$. Then, for any $M_0 \subseteq \mathcal{B}_{\mathcal{L}_0}$ and $C \in P_1$, C is complete in $I(M_0, CH_{P_1})$ if and only if C is complete in $I(M_0, CH_{P_2})$.*

Lemma 6.11 *Let P be a DRLP. Suppose $CH_P(q_j) = CH_P(q_i)\cdot a$, where $a \in \Sigma$ and $q_i, q_j \in \Pi(P)$. Then, for any $M_0 \subseteq \mathcal{B}_{\mathcal{L}_0}$, the clause $q_i([a|X]) \leftarrow q_j(X)$ is complete in $I(M_0, CH_P)$.*

Proof: By the definition of the extended model, for any $x \in \Sigma^*$, it follows that

$$\begin{aligned} q_i([ax]) \in I(M_0, CH_P) \quad &\text{iff} \quad q_0([CH_P(q_i)\cdot ax]) \in M_0 \\ &\text{iff} \quad q_0([CH_P(q_j)\cdot x]) \in M_0 \\ &\text{iff} \quad q_j([x]) \in I(M_0, CH_P). \end{aligned}$$

Hence, from Proposition 6.4, $q_i([a|X]) \leftarrow q_j(X)$ is complete in $I(M_0, CH_P)$. □

Theorem 6.12 *The predicate characterization CH_P constructed by the algorithm is, at any time, consistent and $CH_P(q_0) = \varepsilon$.*

Proof: It is clear that $CH_P(q_0) = \varepsilon$.

Let P be a hypothesis constructed by the algorithm. From the way of constructing the predicate characterization, for any $q_j \in \Pi(P)$ $(j \geq 1)$, there exists a predicate symbol $q_i \in \Pi(P)$ such that $CH_P(q_j) = CH_P(q_i) \cdot a$ for some $a \in \Sigma$. On the other hand, in defining the characteristic string $CH_P(q_j) = CH_P(q_i) \cdot a$, the clause $q_i([a|X]) \leftarrow q_j(X)$ is added to the hypothesis simultaneously. Since Lemma 6.11 ensures the clause being complete in $I(M_0, CH_P)$, it is never removed from the hypothesis. Hence, for any $q_j \in \Pi(P)$ $(j \geq 1)$, there exist clauses in P that are necessary for constructing such a derivation tree as in Definition 6.5.

Since $CH_P(q_0) = \varepsilon$, $q_0([CH_P(q_0)]) = q_0([\varepsilon]) = q_0([])$ itself gives a derivation tree as in Definition 6.5. This holds even if P is empty. Thus, the theorem holds. □

Now we show the correctness of the algorithm. That is, for any regular model M_0, the algorithm identifies M_0 in the limit. Since the conjectures of Algorithm 6.1 are consistent with known facts, it is sufficient to show the followings:

- The algorithm produces an infinite sequence of conjectures.
- The infinite sequence of the conjectures converges a DRLP.

For the former, we must show that the inner **repeat** loop of Algorithm 6.1 terminates finitely. For the latter, we must show that there are at most finitely many occasions in which hypotheses are modified. Finally, it is sufficient to show that the bodies of the two **while** loops are executed at most finitely many times in total.

When the target model is empty, the bodies of the **while** loops are never executed, because the initial hypothesis is empty. Thus, in the following, we assume that the target regular model M_0 is not empty.

Lemma 6.13 *Let P be any hypothesis and CH_P be the predicate characterization for P constructed by Algorithm 6.1. For any $q_i \in \Pi(P)$, there exists a string $x \in \Sigma^*$ such that $q_0([CH_P(q_i) \cdot x]) \in M_0$.*

Proof: First, we show that, for any non-unit clause $q_k([a|X]) \leftarrow q_j(X) \in P$, there exists a string $x \in \Sigma^*$ such that $q_k([ax]) \in I(M_0, CH_P)$.

The clause whose head is $q_k([a|X])$ first appears in a hypothesis after executing the last **else** statement in the procedure call of **search_clause** on the input $q_k([ax])$. Let P' be the algorithm's hypothesis at that time. Then $q_k([ax]) \in I(M_0, CH_{P'})$ and $q_k([ax]) \notin M(P')$. By Proposition 6.7, for any subsequent CH_P, it holds that $q_k([ax]) \in I(M_0, CH_P)$. Thus, for any non-unit clause $q_k([a|X]) \leftarrow q_j(X) \in P$, there exists a string $x \in \Sigma^*$ such that $q_k([ax]) \in I(M_0, CH_P)$.

On the other hand, for any $q_i \in \Pi(P)$ $(i \geq 1)$, there exists a predicate symbol $q_k \in \Pi(P)$ such that $CH_P(q_i) = CH_P(q_k) \cdot a$ for some $a \in \Sigma$. By the argument in the proof of Theorem 6.12, there exists a clause $q_k([a|X]) \leftarrow q_i(X) \in P$. By the above discussion, there exists a string $x \in \Sigma^*$ such that $q_k([ax]) \in I(M_0, CH_P)$, that is, $q_0([CH_P(q_k) \cdot ax]) \in M_0$. Hence, for any $q_i \in \Pi(P)$ $(i \geq 1)$, there exists a string $x \in \Sigma^*$ such that $q_0([CH_P(q_i) \cdot x]) \in M_0$.

For the predicate q_0, since M_0 is not empty, there exists a string $x \in \Sigma^*$ such that $q_0([CH_P(q_i) \cdot x]) = q_0([x]) \in M_0$. □

In the following lemma, we consider the DRLP $\hat{P}$ with the minimum number of predicate symbols satisfying $M_0 = M(\hat{P})_{q_0}$. Such a DRLP can be constructed, by a similar method in the proof of Theorem 6.1, from the minimum size DFA which accepts $L(M_0)$.

Lemma 6.14 *Let $\hat{P}$ be a DRLP with the minimum number of predicate symbols such that $M_0 = M(\hat{P})_{q_0}$. Let P be an arbitrary hypothesis constructed by the algorithm. Then it follows that $|\Pi(P)| \leq |\Pi(\hat{P})|$.*

Proof: Let CH_P be the predicate characterization for P constructed by the algorithm.

From Lemma 6.13, for any $q_i \in \Pi(P)$, there exists a string $x \in \Sigma^*$ such that $q_0([CH_P(q_i) \cdot x]) \in M_0$. Since $M_0 = M(\hat{P})_{q_0}$, there uniquely exists a proof tree of $q_0([CH_P(q_i) \cdot x])$ on P. Hence, for any $q_i \in \Pi(P)$, there uniquely exists a predicate symbol $\hat{q}_i \in \Pi(\hat{P})$ such that $\hat{q}_i([])$ appears in the derivation tree of $q_0([CH_P(q_i)])$ on $\hat{P}$. For such $\hat{q}_i$, it holds that, for any string $x \in \Sigma^*$,

$$\hat{q}_i([x]) \in M(\hat{P}) \quad \text{iff} \quad q_0([CH_P(q_i) \cdot x]) \in M(\hat{P}) \qquad (*)$$

Now we consider the mapping τ from $\Pi(P)$ to $\Pi(\hat{P})$ such that $\tau(q_i) = \hat{q}_i$. For the proof of the lemma, it is sufficient to show that τ is injective.

Suppose that $\tau(q_i) = \tau(q_j)$ for some $i < j$. Then, for any string $x \in \Sigma^*$, it holds that

$$\begin{aligned} q_0([CH_P(q_i)\cdot x]) \in M(\hat{P}) \quad &\text{iff} \quad \hat{q}_i([x]) \in M(\hat{P}) && \text{(from } (*)\text{)} \\ &\text{iff} \quad q'_j([x]) \in M(\hat{P}) && \text{(from the assumption)} \\ &\text{iff} \quad q_0([CH_P(q_j)\cdot x]) \in M(\hat{P}). && \text{(from } (*)\text{)} \end{aligned}$$

Hence, we obtain the following relation:

$$q_0([CH_P(q_i)\cdot x]) \in M_0 \quad \text{iff} \quad q_0([CH_P(q_j)\cdot x]) \in M_0.$$

Since $0 \leq i < j$, there exists a predicate symbol $q_k \in \Pi(P)$ $(k < j)$ such that $CH_P(q_j) = CH_P(q_k)\cdot a$ for some $a \in \Sigma$. Hence, for any $x \in \Sigma^*$, the following relation holds.

$$q_0([CH_P(q_i)\cdot x]) \in M_0 \quad \text{iff} \quad q_0([CH_P(q_j)\cdot x]) \in M_0 \quad \text{iff} \quad q_0([CH_P(q_k)\cdot ax]) \in M_0.$$

As a result, it holds that

$$q_i([x]) \in I(M_0, CH_P) \quad \text{iff} \quad q_k([ax]) \in I(M_0, CH_P).$$

Hence, it follows from Proposition 6.4 that the clause $C = q_k([a|X]) \leftarrow q_i(X)$ is complete in $I(M_0, CH_P)$. Since $i < j$, C is generated by the procedure **next_clause** and added to the hypothesis before the clause $q_k([a|X]) \leftarrow q_j(X)$. From Proposition 6.10, C is complete in any extended model subsequently. Thus C is never removed from subsequent hypotheses. This contradicts that $CH_P(q_j) = CH_P(q_k)\cdot a$. □

Theorem 6.15 *For any regular model M_0, Algorithm 6.1 identifies M_0 in the limit.*

Proof: It is clear that the procedures **next_clause** and **search_clause** terminate finitely and return the desired output. It follows from Lemma 6.8 and Lemma 6.9 that the procedures **contradiction_back-tracing** and **uncovered_atom** terminate finitely and return the desired output. Hence, each computation in the bodies of the two **while** loops terminates finitely and a operation corresponding to either the two solid lines or the dotted line in Figure 6.3 is executed. The operation is executed at most once for each clause enumerated in the left hand of Figure 6.3. Hence, if only finitely many clauses are generated, then the bodies of the two **while** loops are executed at most finitely many times in total.

On the other hand, by Proposition 6.10 and Lemma 6.11, once a predicate symbol is introduced into a hypothesis, the symbol never disappears from the subsequent hypotheses. Hence, by Lemma 6.14, only finitely many predicate symbols are introduced into the hypotheses. Thus the number of clauses generated is at most finite.

From the discussion preceding Lemma 6.13, this completes the proof of the theorem. □

6.5 Time Complexity of the Algorithm

We assume that the given oracle answers each membership query immediately. Thus, the examination if $q_i([x]) \in I(M_0, CH_P)$ is done in one step.

Let M_0 be a target regular model and $\hat{P}$ be a DRLP with a minimum number of predicate symbols such that $M_0 = M(\hat{P})_{q_0}$.

Theorem 6.16 *At any stage in inferring M, after Algorithm 6.1 reads a fact, it outputs a conjecture in time polynomial in $size(S)$, $|\Sigma|$, and $|\Pi(\hat{P})|$, where S is the set of all positive and negative facts given so far, that is, $S = S_{true} \cup S_{false}$.*

Proof: For notational convenience, we denote $|S|$ by ℓ, $|\Sigma|$ by k, $|\Pi(\hat{P})|$ by n, and the maximum size of any element in S by m. Note that $size(S) \leq m\ell$. Let P and CH_P be the hypothesis and the predicate characterization in the algorithm. Then, it follows from the argument in the previous section that $|P| \leq n(k+1)$ and $|CH_P| \leq n$.

The both procedure **next_clause** and **search_clause** just make a simple search in CH_P and P respectively. Thus the time required in each procedure call is bounded by a linear in $n(k+1)$.

From the structure of a derivation tree on a DRLP, the input proof tree of some negative fact α for the procedure **contradiction_backtracing** can be treated as a sequence with length at most $size(\alpha)$. The procedure just traces the sequence from the root to the leaf. Thus the procedure terminates and find a false clause in time linear in $size(\alpha) \leq m$.

For an input ground atom β, the procedure **uncovered_atom** searches a clause in P whose head is unifiable with β. If such a clause found then it calls itself recursively with input β' such that $size(\beta') = size(\beta) - 1$. Otherwise, it returns the input directly. Since the

main operation executed in the procedure is to search the clause, the time required in the procedure call is bounded by a linear in $size(\beta) \times |P| \leq mn(k+1)$.

As a result, the time required in an execution of the body of each **while** loop is bounded by a linear in $mn(k+1)$.

Each examination of if $\alpha \in M(P)$ in the condition of two **while** loops of Algorithm 6.1 can be done by a similar procedure to **uncovered_atom**. By replacing the inner **if-then-else** block by the recursive call **uncovered_atom**$(q_j(X)\theta)$, we can obtain a simple resolution prover for DRLP's. Thus the examination also terminates in time linear in $size(\alpha) \times |P| \leq mn(k+1)$. In the worst case, the examination is done for every element in S_{true} or S_{false}. Hence, the time required in each iteration of the **while** loops is bounded by a linear in $\ell mn(k+1)$. On the other hand, from the argument in the previous section, each **while** loop is entered at most kn^2+n times. Where kn^2+n is the number of possible clauses of a DRLP constructed from at most n predicates.

Consequently, the amount of the time required in each iteration of the outer **repeat** loop is at most $O((kn^2+n)(\ell mn(k+1)) = O(k^2\ell mn^3)$. □

Chapter 7

Learning Simple Deterministic Languages

In this chapter, we consider the problem of learning simple deterministic languages using membership queries and extended equivalence queries. The discussion in this chapter will be made in terms of formal language theory, because a logic program corresponding to a context-free grammar uses the technique of differential lists which are too complicated to make the discussion clear.

A simple deterministic language (SDL, for short) is a language that is accepted by a 1-state deterministic push-down automaton by empty store. The class of SDL's is a proper sub-class of deterministic languages. The SDL's may also be characterized as the languages that are generated by context-free grammars in a special form of Greibach normal form, called simple deterministic grammars (SDG's, for short).

Angluin [Ang87a] shows that the class of k-bounded context-free grammars is learnable in polynomial time using membership queries, nonterminal membership queries and equivalence queries. The algorithm described in this chapter is based on her algorithm. Both algorithms are essentially based on Shapiro's model inference algorithm [Sha82]. Our setting, however, differs from Angluin's and Shapiro's in the types of queries that are available to the learning algorithm. That is, the algorithm is allowed to use membership queries but not nonterminal membership queries. This difference leads to the problem of introducing new nonterminals that are not observed in interactions between the oracle and the inference algorithm. As

discussed in Chapter 5, this relates to the problem of inventing necessary predicates in learning logic programs.

Another feature of our setting is that the algorithm is allowed to use extended equivalence queries. The equivalence query defined in [Ang88] is allowed to conjecture only elements of the original hypothesis space. For example, if the target class[1] of learning is a set of concept representations $R = \{r_1, r_2, \ldots\}$, then any equivalence query made by the learning algorithm must be with some r_i from R. We lift this restriction in this chapter. In particular, the learning algorithm described in this chapter is allowed to make an equivalence query conjecturing any grammar in 2-standard form; not necessarily simple deterministic. Hence, each intermediate hypothesis conjectured by an extended equivalence query might define a general context-free language.

Yokomori [Yok88] gives another algorithm for learning SDL's in polynomial time. Our setting also differs from his, as will be described at the end of Section 7.6. Berman and Roos [BR87] show that the class of deterministic one-counter languages is learnable in polynomial time using membership queries and equivalence queries. The class of one-counter languages is incomparable with the class of SDL's. For example, the language $\{\{a^n b^n \mid n \geq 1\}c\}^+$ is not simple deterministic, but deterministic one-counter. On the other hand, the language $\{a^m b^n c a^n b^m \mid m, n \geq 1\}$ is not deterministic one-counter, but simple deterministic. However, there is an interesting similarity that both are classes with decidable equivalence problems.

In the next section, we give the definition of simple deterministic grammars and languages. We also give several properties of the grammars which will be necessary for our discussions. In Section 7.2, we define the two types of queries by which a learning algorithm obtains the information about a target language. In Section 7.3, we give the learning algorithm and the main result of this chapter. In Section 7.4, we describe how to diagnose an incorrect hypothesis. The diagnosis method is essentially same as that of the procedure **contradiction_backtracing** given in the previous chapter. In Section 7.5, we describe how to generate nonterminals with appropriate intended models. The correctness and the time complexity of the algorithm will be discussed in Section 7.6.

[1] *The target class* is a class of representations that define the class of concepts to be learned by the algorithm (see, e.g., [Pit89]).

This chapter is based on the paper [Ish89b, Ish90].

7.1 SDG and SDL

First, we define simple deterministic grammars and languages.

Definition 7.1 A context-free grammar in Greibach normal form G is *simple deterministic* if the following condition holds: for any $A \in N, a \in \Sigma, \alpha, \beta \in N^*$, if there exist productions $A \to a\alpha$ and $A \to a\beta$ in P, then $\alpha = \beta$.

Definition 7.2 A language L is *simple deterministic* if there exists an SDG G such that $L(G) = L$.

Example 7.1 The grammar $G = (\{S, A, B, C\}, \{a, b\}, P, S)$, where

$$P = \{S \to aA, A \to b, A \to aB, B \to aBC, B \to bC, C \to b\},$$

is one of the SDG's that generates an SDL $\{a^m b^m | 1 \leq m\}$.

The following propositions (see, e.g., [Har79]) provide properties of SDG's and SDL's that are useful for our purpose.

Proposition 7.1 *For any SDG $G = (N, \Sigma, P, S)$, G is unambiguous, that is, for any $w \in L(G)$, there is a unique left-most derivation of w from S.*

Proposition 7.2 *Let $G = (N, \Sigma, P, S)$ be an SDG. For any $A \in N, x \in \Sigma^+$ and $\alpha \in N^*$, if there exists a derivation $A \Rightarrow^* x\alpha$, then $L(\alpha) = \overline{x}L(A)$.*

Proposition 7.3 *Let $G = (N, \Sigma, P, S)$ be an SDG. For any $A \in N$, $L(A)$ is prefix-free, that is, if $x \in L(A)$, then, for any $y \in \Sigma^+$, $xy \notin L(A)$.*

Proposition 7.4 *For any SDG G, there exists an equivalent SDG G' that is in 2-standard form, i.e., there exists an SDG $G' = (N', \Sigma, P', S)$ such that*

(1) $L(G) = L(G')$;

(2) *Each production in P' is of one of the following forms: $A \to a$, $A \to aB$, $A \to aBC$, where $A, B, C \in N', a \in \Sigma$.*

Proposition 7.4 allows us to consider only (ε-free) context-free grammars in 2-standard form as the hypotheses of the learning algorithm.

We will analyze the complexity of the learning algorithm given in this chapter on two types of complexity measures: one is the length of the given example strings and the other is the number of nonterminals of a *minimal SDG* for the target language. Let L be an SDL. A *minimal SDG for* L is an SDG $G = (N, \Sigma, P, S)$ in 2-standard form satisfying the following conditions:

1. $L(G) = L$;
2. For any SDG $G' = (N', \Sigma, P', S')$ in 2-standard form such that $L(G') = L$, $|N| \leq |N'|$.

7.2 Learning via Queries

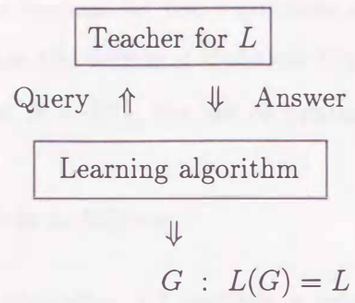
In this chapter, we consider the problem of learning SDL's under the framework called *learning via queries* which is rather different form that of identification in the limit. In the framework, the learning algorithm can actively get information about a target SDL L by making queries to the oracle. The criterion for success of learning is to terminate finitely and output a grammar G such that $L(G) = L$. According to Angulin's fashion [Ang87b], in this chapter, we call an oracle for L a *teacher* for L. Figure 7.2 illustrates the framework of learning via query.

Let L be the target SDL to be learned by our learning algorithm. We assume that the teacher for L can answer the following two types of queries.

Definition 7.3 A *membership query* proposes a string $x \in \Sigma^+$ and asks whether $x \in L$. The reply is either *yes* or *no*.

Definition 7.4 An *extended equivalence query* conjectures a grammar G in 2-standard form and asks whether $L = L(G)$. The reply is either *yes* or *no*. If it is *no*, then a *counterexample* is also provided. A counterexample is a string x in the symmetric difference of L and $L(G)$.

Figure 7.1: The framework of learning via query



If $x \in L - L(G)$, x is called a *positive* counterexample, and if $x \in L(G) - L$, x is called a *negative* counterexample. The choice of a counterexample is assumed to be arbitrary.

Note the difference between the extended equivalence query and the equivalence query defined in [Ang88]. The equivalence query is only allowed to conjecture members of the target class. Thus, in learning SDL's, any hypothesis conjectured by the algorithm would have to be a grammar generating an SDL. In contrast, the hypothesis conjectured by an extended equivalence query does not have to generate an SDL.

A teacher who answers equivalence queries and membership queries was called a *minimally adequate teacher* [Ang87b]. We call a teacher who answers extended equivalence queries and membership queries an *extended minimally adequate teacher.*

The notion of the extended equivalence query corresponds to the notion, in the context of the PAC-learning model, of learning the target class R *in terms of* the class of representations R', not necessarily identical to R (see, e.g., [PW88]). Informally, R is said to be PAC-learnable in terms of R' if there exists a polynomial time algorithm A such that for any target concept (description) $r \in R$, if A is given randomly chosen examples of r, A outputs, with high probability, a concept (description) $r' \in R'$ that approximates the target concept r. In our setting, R corresponds to the class of SDG's (or SDL's) and R' corresponds to the class of CFG's in 2-standard form. In general, such a relaxation of the learnability criterion enriches the learnable classes of concepts. For example, the class of k-term DNF's is not PAC-learnable in terms of itself unless $\mathcal{RP} = \mathcal{NP}$, but the class is learnable in terms of the class of k-CNF's. For the result given in this chapter, however, the learnability of SDG's in terms of itself (the learnability of SDG's from a minimally adequate teacher) is still open.

7.3 A Learning Algorithm for SDL

Let L be the unknown SDL to be learned by the algorithm and $G_0 = (N_0, \Sigma, P_0, S)$ be a minimal SDG for L. We assume that the terminal alphabet Σ and start symbol S are known to the learning algorithm, but that $N - \{S\}$, the set of nonterminals except S, and P, the set of productions, are unknown.

The main result of this chapter is as follows.

Theorem 7.5 *For any SDL L, Algorithm 7.1 outputs a grammar G in 2-standard form such that $L(G) = L$ using extended equivalence queries and membership queries. Moreover, at any point during the run, the time used by the algorithm to that point is bounded by some polynomial in $|N_0|$, the number of nonterminals of a minimal SDG for L, and the length of the longest counterexample returned by any equivalence query seen to that point.*

Note that the grammar learned by the algorithm may not be SDG. The grammar is simply in 2-standard form.

Here, we describe the outline of Algorithm 7.1. The details concerning how to diagnose hypotheses and how to introduce new nonterminals will be described in the succeeding two sections.

First, the algorithm initializes N to $\{S\}$, and P to the set of all productions containing S as the only nonterminal. As a model M for G, we initially consider $\{M(S) = L\}$. Models for any other nonterminals introduced by the algorithm will be defined in Section 7.5. Then the algorithm iterates the following loop: An extended equivalence query is made, conjecturing G. If the reply is *yes*, then the algorithm outputs G and halts. Otherwise, a counterexample w is returned. The algorithm tries to find a derivation tree T of G such that $rt(T) = S$ and $fr(T) = w$. If it exists, that is, when w is a negative counterexample, the algorithm diagnoses G on T and finds an incorrect production for M. The incorrect production is removed from P. Otherwise, that is, when w is a positive counterexample, new nonterminals are introduced and all new productions constructed from them are added to P.

In this chapter, we assume a parsing sub-procedure that runs in time polynomial in the size of a grammar G and $|w|$, e.g., Angluin's [Ang87a] parsing procedure[2]. In the

[2]Since G is in 2-standard form, Lemma 3 and Lemma 4 in [Ang87a] hold. In fact, the procedure returns

Algorithm 7.1: An algorithm for learning SDL's

Given: An extended minimally adequate teacher for L and a terminal alphabet Σ.
Output: A grammar $G = (N, \Sigma, P, S)$ in 2-standard form such that $L(G) = L$.
Procedure:

$N := \{S\}$; $P := \{S \to aSS, S \to aS, S \to a | a \in \Sigma\}$; $G := (N, \Sigma, P, S)$;
repeat
  make an extended equivalence query with G;
  if the reply is a positive counterexample **then**
    introduce new nonterminals with their models;
    put all candidate productions into P;
  else if the reply is a negative counterexample, **then**
    diagnose G;
    remove the incorrect production returned
      by the diagnosis routine from P;
until the reply is *yes.*
Output G.

following two subsections, we describe the diagnosis routine and how new nonterminals and productions are generated. Then, in the third subsection, we show the correctness and characterize the complexity of the entire algorithm.

7.4 Diagnosing an Incorrect Hypothesis

The diagnosis routine finds an incorrect production for M on an input derivation tree T of G such that $fr(T) \notin M(rt(T))$. It is essentially same as the procedure **contradiction_backtracing** given in the previous chapter and each of them is a special case of the *contradiction backtracing algorithm* given by Shapiro [Sha81].

For a given input derivation tree T, the diagnosis routine considers, in turn, each child of the root of T. If the child is labeled with a nonterminal and T' is the sub-tree rooted at the child, then the diagnosis routine inquires whether $fr(T') \in M(rt(T'))$. If $fr(T') \notin M(rt(T'))$, then it calls itself recursively with T'. Otherwise, it goes on to the next child of

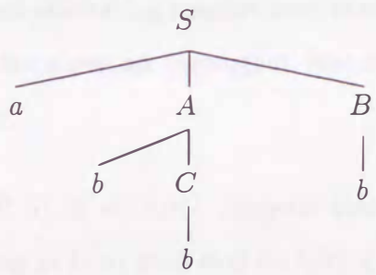
a parse-DAG (directed acyclic graph) instead of a derivation tree. Our discussion, however, is not affected by the difference.

the root of T. If there is no nonterminal child such that $fr(T') \notin M(rt(T'))$, for the sub-tree T' rooted at the child, then the diagnosis routine returns the production $rt(T) \rightarrow \alpha \in P$, where α is the concatenation of the labels of the children of the root of T in left-to-right order.

For example, consider the derivation tree in Figure 7.4 for a negative counterexample *abbb*.

Initially, $abbb \notin M(S) = L$. First, the child labeled with A generating the string bb is considered. The diagnosis routine inquires whether $bb \in M(A)$. If $bb \notin M(A)$, then it calls itself recursively with the sub-tree rooted at the child. If $bb \in M(A)$, then it goes to the next child labeled with B and makes a similar inquiry. If $b \notin M(B)$, then it returns the production $B \rightarrow b$. Otherwise, it returns the production $S \rightarrow aAB$.

Figure 7.2: An example of an input for the diagnosis routine



In [Ang87a], such a diagnosis is made through nonterminal membership queries of the type "$bb \in L(A)$?". In our approach, it is performed through membership queries only. The next section shows how to introduce new nonterminals and replace nonterminal membership queries by membership queries.

Lemma 7.6 *Suppose that the diagnosis routine is given as its input a derivation tree T of G such that $fr(T) \notin M(rt(T))$. Then it returns a production in P that is incorrect for M.*

Proof: Since each recursive call is with a proper sub-tree of its input derivation tree, the diagnosis routine must eventually terminate and output some production in P (since each sub-tree is also a derivation tree of G, the output production is a member of P).

Let $A \rightarrow \alpha$ be the returned production. For each nonterminal occurrence X in the production, let T_X be the sub-tree of T that is rooted at the corresponding node labeled with X

in T. From the input condition, it holds that $fr(T_A) \notin M(A)$. If α contains no nonterminal, then the empty replacement ρ satisfies $\rho[\alpha] = \alpha = fr(T_A) \notin M(A)$. Otherwise, from the termination condition of the procedure, for each B_i appearing in α, $fr(T_{B_i}) \in M(B_i)$. Thus there exists a replacement $\rho = \langle (fr(T_{B_1}), B_1), \ldots, (fr(T_{B_n}), B_n) \rangle$ that is compatible with α such that, for each i, $fr(T_{B_i}) \in M(B_i)$, but $\rho[\alpha] = fr(T_A) \notin M(A)$. So $A \rightarrow \alpha$ is incorrect for M. □

Note that, at the initial call to the diagnosis routine, the input derivation tree T is for a negative counterexample w. Since $fr(T) = w \notin L = M(S) = M(rt(T))$, the input condition is satisfied initially.

7.5 Generating Nonterminals and Productions

The key idea of the nonterminal-generating routine has its roots in an extension of a model described in [Ish89a]. First, we show an important feature of SDG's for describing the nonterminal-generating routine.

Lemma 7.7 *Let $G = (N, \Sigma, P, S)$ be an SDG. Suppose that $A \Rightarrow^* rB\alpha$ for $A, B \in N, \alpha \in N^*, r \in \Sigma^+$, and that t is a string in $L(\alpha)$ such that $Suf_j(t) \notin L(\alpha)$ for any j $(1 \leq j \leq |t|-1)$ (if $\alpha = \varepsilon$ then $t = \varepsilon$). Then, for any $x \in \Sigma^+$, $x \in L(B)$ if and only if* (i) *$rxt \in L(A)$ and* (ii) *$rPre_i(x)t \notin L(A)$ for any i $(1 \leq i \leq |x| - 1)$.*

Proof: Suppose $x \in L(B)$. Then $A \Rightarrow^* rB\alpha \Rightarrow^* rx\alpha \Rightarrow^* rxt$. Thus, $rxt \in L(A)$. Since $L(B)$ is prefix-free, $Pre_i(x) \notin L(B)$ for any i $(1 \leq i \leq |x| - 1)$. Hence, if $rPre_i(x)t \in L(A)$, that is, $Pre_i(x)t \in \overline{r}L(A) = L(B\alpha)$, then there exists j $(1 \leq j \leq |t| - 1)$ such that $Pre_i(x)Pre_j(t) \in L(B)$ and $Suf_j(t) \in L(\alpha)$. This contradicts the fact that $Suf_j(t) \notin L(\alpha)$ for any j $(1 \leq j \leq |t| - 1)$. Thus, $rPre_i(x)t \notin L(A)$ for any i $(1 \leq i \leq |x| - 1)$.

Conversely, assume that (i) and (ii) hold. From (i), it follows that $xt \in \overline{r}L(A) = L(B\alpha)$. Since there is no proper suffix of t in $L(\alpha)$, there exists j $(1 \leq j \leq |x|)$ such that $Pre_j(x) \in L(B)$ and $Suf_j(x)t \in L(\alpha)$. On the other hand, from (ii), $Pre_i(x)t \notin L(B\alpha)$ for any i $(1 \leq i \leq |x| - 1)$. Hence, for any i $(1 \leq i \leq |x| - 1)$, $Pre_i(x) \notin L(B)$. Thus, $j = |x|$. This shows that $Pre_{|x|}(x) = x \in L(B)$. □

In the learning algorithm, new nonterminals are introduced whenever there is a positive counterexample w. The nonterminal-generating routine constructs nonterminals with their appropriate models from w.

Let w be a positive counterexample such that $|w| \geq 2$.

Definition 7.5 A *Nonterminal generated from a positive counterexample* w is a triplet (r, s, t) of strings such that $rst = w$ where $r, s \in \Sigma^+$ and $t \in \Sigma^*$. The set of all nonterminals generated from w is denoted by $N(w)$.

Example 7.2 Let $w = aabb$, then

$$N(w) = \{(a, abb, \varepsilon), (a, ab, b), (a, a, bb), (aa, bb, \varepsilon), (aa, b, b), (aab, b, \varepsilon)\}$$

For each triple $(r, s, t) \in N(w)$, let $\varphi(r, s, t)$ denote the shortest suffix of t in $\overline{rs}L$, i.e.,

$$\varphi(r, s, t) = Suf_i(t) \text{ where } i = \max_{0 \leq j \leq |t|-1} \{j \mid Suf_j(t) \in \overline{rs}L\}.$$

The intended model of each nonterminal in $N(w)$ is defined as follows.

Definition 7.6 For each triple $(r, s, t) \in N(w)$, define

$$M((r, s, t)) = \{x \in \Sigma^+ \mid rx\varphi(r, s, t) \in L \text{ and } rPre_i(x)\varphi(r, s, t) \notin L \text{ for any } i\ (1 \leq i \leq |x| - 1)\}.$$

Let w be a newly given positive counterexample at a stage of learning. Then N is set to $N \cup N(w)$. Let $P_{N(w)}$ be a set of all productions in 2-standard form constructed from N that have never appeared in P, that is, for each $a \in \Sigma$, $P_{N(w)}$ contains productions $A \rightarrow a\alpha$ such that $A\alpha \in N^+$, $|\alpha| \leq 2$ and $A\alpha$ contains at least one element of $N(w)$. Then P is set to $P \cup P_{N(w)}$. Note that, at any point during the learning, P contains at most $|N| \times |\Sigma| \times (|N| + 1)^2$ productions for N generated by the algorithm to that point.

Lemma 7.8 *Let N be the set of known nonterminals. Suppose that w is a new positive counterexample. Then the time required for generating nonterminals and computing new productions is bounded by a non-decreasing polynomial in $|N|$ and $|w|$.*

Proof: There are at most $|w|(|w|-1)/2$ nonterminals in $N(w)$, and $N(w)$ is computable in time polynomial in $|w|$. For each (r,s,t) in $N(w)$, the string $\varphi(r,s,t)$ is computed by making at most $|t|$ membership queries with the strings $rsSuf_j(t)$ $(0 \leq j \leq |t|-1)$. Moreover the set $P_{N(w)}$ is computable in time polynomial in $|N|$ and $|N(w)|$. These facts prove the lemma. □

Lemma 7.9 *Let L be an SDL, w be a string in L, and $G = (N, \Sigma, P, S)$ be an SDG such that $L(G) = L$. For any $A \in N - \{S\}$ that appears in the derivation $S \Rightarrow^* w$, there exists a nonterminal $(r,s,t) \in N(w)$ such that $L(A) = M((r,s,t))$.*

Proof: Suppose that $S \Rightarrow^* rA\alpha \Rightarrow^* rs\alpha \Rightarrow^* rst = w$. Then, from the definition of $N(w)$, the triple (r,s,t) is in $N(w)$. (Since G is an SDG and $A \neq S$, neither r nor s is ε.) Since $L(S) = L(G) = L$, by Proposition 7.2, $L(\alpha) = \overline{rs}L(S) = \overline{rs}L$. By the definition of $\varphi(r,s,t)$, $\varphi(r,s,t) \in L(\alpha)$ and $Suf_j(\varphi(r,s,t)) \notin L(\alpha)$ for any j $(1 \leq j \leq |\varphi(r,s,t)| - 1)$. Hence, by Lemma 7.7 and the definition of $M((r,s,t))$, $L(A) = M((r,s,t))$. □

The above lemma ensures that if the learning algorithm is given a positive counterexample w, then it can make all nonterminals with appropriate models that are necessary for generating w. As a result, nonterminal membership queries used by Angluin's [Ang87a] or Shapiro's [Sha82] algorithm can be replaced by membership queries. For any $x \in \Sigma^*$ and $A \in N(w)$, the diagnosis routine can accomplish each inquiry as to whether $x \in M(A)$ by making $|x|$ membership queries.

7.6 Correctness and Complexity

In what follows, let $G = (N, \Sigma, P, S)$ be the current hypothesis of the algorithm and

$$M = \{M(S), M((r_1,s_1,t_1)), \ldots, M((r_{|N|-1}, s_{|N|-1}, t_{|N|-1}))\}$$

be the model for G defined in the previous section.

Lemma 7.10 *At any point during the learning, the time required by the diagnosis routine on an input derivation tree for a negative counterexample w is bounded by a non-decreasing polynomial in $|w|$ and ℓ_p, where ℓ_p is the length of the longest positive counterexample returned by any equivalence query seen to that point.*

Proof: Since G is in 2-standard form, there are at most $|w|$ occurrences of nonterminals in the derivation tree. Thus, the number of inquiries made by the diagnosis routine is at most $|w|$. For each inquiry as to whether $x \in M(A)$ or not, if $A = S$, then only one membership query "$x \in L$?" is made. Otherwise, that is, if $A = (r, s, t)$, the routine makes at most $|x|$ membership queries "$rPre_i(x)\varphi(r, s, t) \in L$?" for $1 \leq i \leq |x|$. Since x is a sub-string of w, the total number of queries made in a diagnosing process is at most $|w|^2$. Since the main operations performed in the diagnosis routine are forming strings $rPre_i(x)\varphi(r, s, t)$ and making membership queries, it is clear that the claim of the lemma holds. □

Lemma 7.11 *Let $G_0 = (N_0, \Sigma, P_0, S)$ be a minimal SDG for the target language L. The total number of given positive counterexamples is bounded by $|N_0|$.*

Proof: Let w_n be the nth positive counterexample given to the learning algorithm. We define $N_0(w_n)$ and $P_0(w_n)$ as follows:

$$N_0(w_n) = \{A \in N_0 \mid \exists u \in \Sigma^+, \exists \alpha \in N^*, S \Rightarrow_{G_0}^* uA\alpha \Rightarrow_{G_0}^* w_n\},$$
$$P_0(w_n) = \{A \to a\alpha \in P_0 \mid a \in \Sigma, A\alpha \in (\bigcup_{i=1}^{n} N_0(w_i))^+\}.$$

When w_n is given, the learning algorithm computes $N(w_n)$ and sets N to $N \cup N(w_n)$. Then it computes all new candidate productions and adds them to P as described in the previous section.

By Lemma 7.9, for each nonterminal $A \in N_0(w_n)$, there exists a nonterminal $A' \in N(w_n)$ such that $L(A) = M(A')$. Under this correspondence of A and A', for every production in $P_0(w_n)$, a corresponding production is added to P at least once. By Proposition 2.2, these corresponding productions are correct for M. Since correct productions are never removed from P, whenever the $n+1$st positive counterexample is given, there exists at least one nonterminal $A \in N_0$ such that

$$A \in N_0(w_{n+1}) \quad \text{and} \quad A \notin \bigcup_{i=1}^{n} N_0(w_i).$$

Thus, the number of given positive counterexamples is at most $|N_0|$. □

Lemma 7.12 *At any point during the learning, the number of nonterminals introduced by the learning algorithm is bounded by* $|N_0|\ell_p(\ell_p - 1)/2$, *where* ℓ_p *is the length of the longest positive counterexample returned by any equivalence query seen to that point.*

Proof: For each positive counterexample w_i, $|N(w_i)|$ is at most $|w_i|(|w_i| - 1)/2$ as stated in the previous section. By Lemma 7.11, the total number of nonterminals introduced by the algorithm is bounded by $|N_0|\ell_p(\ell_p - 1)/2$. □

Proof of Theorem 7.5: From the method of introducing new productions and Lemma 7.12, the total number m of productions introduced into P is at most

$$m = \frac{|N_0|\ell_p(\ell_p - 1)}{2} \times |\Sigma| \times \left(\frac{|N_0|\ell_p(\ell_p - 1)}{2} + 1\right)^2.$$

By Lemma 7.6, for each given negative counterexample, at least one incorrect production is found and it is removed from P. With Lemma 7.11, this implies that, after given at most $|N_0|$ positive counterexamples and at most m negative ones, the learning algorithm outputs a grammar G such that $L(G) = L$.

By Lemma 7.12, at any point during the learning, the size of G is bounded by a non-decreasing polynomial in $|N_0|$ and ℓ, where ℓ is the length of the longest counterexample given to that point. From the assumption on the parsing sub-procedure, the algorithm can determine whether a given counterexample is positive or negative in time polynomial in $|N_0|$ and ℓ. The total number of given counterexamples is at most $|N_0| + m$. With Lemma 7.8 and Lemma 7.10, this proves the claim, made in Theorem 7.5, on the complexity of the learning algorithm. □

Yokomori [Yok88] gives another algorithm for learning SDL's in polynomial time. His algorithm conjectures only SDG's. In his setting, however, a very powerful teacher is assumed. The teacher can answer the following two types of queries: prefix membership queries and derivatives equivalence queries. A prefix membership query is an extension of the membership query. A derivatives equivalence query proposes two pairs of strings $(u_1, w_1), (u_2, w_2)$ and asks whether $\overline{u_1}L\overline{w_1} = \overline{u_2}L\overline{w_2}$, where L is the target language. It is clear that derivatives equivalence queries can be used, in our algorithm, to test whether two candidate nonterminals are identical. For example, for two nonterminals (u_1, v_1, w_1) and (u_2, v_2, w_2), if

$\overline{u_1}L\overline{w_1} = \overline{u_2}L\overline{w_2}$, then they are identical. Thus, the number of nonterminals generated by our algorithm will be reduced. The relationship between the power of the teacher and the efficiency of the learning algorithm remains an interesting open question.

Chapter 8

Conclusion

We briefly summarize the results presented in this thesis with some future problems.

In Chapter 3, we introduced a d-model preserving instance of a program and show that the instantiation preserves the least Herbrand model of the program. We showed that a substitution defined as the difference between the head of an original clause C and the head obtained by the least generalization of $C(M_P)$ gives the d-model preserving instantiation. The result suggests the applicability of the least generalization to inferring program heads.

In Chapter 4, we presented two inference algorithms which identify the class of primitive Prologs in the limit from positive data. The first is a consistent and conservative polynomial update time algorithm that, given the unit clause of the target program, identifies the class in the limit from positive facts with polynomial time updating hypotheses. The second is a consistent but not conservative polynomial update time algorithm that identifies the class in the limit from positive facts. The second inference algorithm employing a natural technique, that is, the 2-mmg algorithm to infer heads of clauses in a target program. The technique is considered as an extension of the method proposed by Ishizaka in [Ish88a].

For the second type of polynomial update time inference algorithms, there is a problem pointed out by Pitt [Pit89]. That is, lacking one of the two conditions, consistency and conservativeness, allows the existence of an tricky polynomial update time inference algorithm. If one of the two is not required, then any exponential update time inference algorithm can perform polynomial update time inference. The tricky algorithm continues to output dummy conjectures to postpone outputting a genuine conjecture until they have enough size

of examples. It is our future work to realize a consistent and conservative polynomial update time inference algorithm that identifies the class of primitive Prologs without hint.

The difficulty of accomplishing conservativeness of the inference algorithm using 2-mmg essentially originates in non-uniqueness of 2-mmg for the entire model $M(P)$ of a primitive Prolog P. For example, consider the following primitive Prolog P.

$$p([a,b,a]).$$
$$p([b|X]) \leftarrow p(X).$$

For the least Herbrand model

$$M(P) = \{p([a,b,a]), p([b,a,b,a]), p([b,b,a,b,a]), p([b,b,b,a,b,a]), \ldots\},$$

there exist two kinds of 2-mmg of $M(P)$:

$$\{p([a,b,a]), p([b,X,Y,Z|W])\} \text{ and } \{p([b,a,b,a]), p([X,b,Y|Z])\}.$$

Actually the former is an instance of the heads of the program P. For any non-empty finite subset S of $M(P)$, it holds that

$$\begin{aligned} P(S, \langle p([a,b,a]), p([b,X,Y,Z|W])\rangle) &= \{\ p([a,b,a]). \\ &\qquad p([b,X,Y,Z|W]) \leftarrow p([X,Y,Z|W]).\} \\ P(S, \langle p([b,a,b,a]), p([X,b,Y|Z])\rangle) &= \{\ p([b,a,b,a]). \\ &\qquad p([X,b,Y|Z]).\}. \end{aligned}$$

Hence, an inference algorithm that uses the 2-mmg algorithm and Algorithm 4.2 as its sub-procedures has a chance to meet the former correct instance of P. Since we know the target program P, we know the former is correct but the latter is overgeneralized. However, it seems difficult for the algorithm to decide which is better, because the algorithm is given only positive examples and both candidates are consistent with all of them. If the algorithm can efficiently (that is, in polynomial time) decide which of competitive hypotheses has a smaller model, then it may avoid producing an overgeneralized hypothesis and achieve consistent and conservative polynomial update time inference. However, it is still open whether a model containment problem for primitive Prologs is solved efficiently.

Contrastively, in [AIS92], we introduced another sub-class of linear Prologs of which element has only one 2-*mmg* of its model and presented a consistent and conservative polynomial update time inference algorithm for the class. The class is also a sub-class of *context-free transformations* that was originally introduced by Shapiro in his study on MIS [Sha81]. Although the sub-class is so restrictive, it can be shown that the sub-class still includes several non-trivial programs in context-free transformations such as *append*, *plus*, $prefix$ etc. For the class of primitive Prologs, it is still open if there exists a consistent and conservative polynomial update time inference algorithm.

In Chapter 5, we discussed the problems in extended model inference, especially, the problems concerned with inventing new predicates. We also proposed a very simple approach to the problems.

Because of the strict restrictions on syntax, the expressive power of the programs introduced in Section 5.4 are restricted to some particular domain. Conversely, however, if an domain can be represented by some particular representation which overcomes the problems mentioned in the chapter, then we might develop a method for an efficient inductive learning over the domain. On the other hand, some kind of programming patterns can be found in practical programs. By combining some patterns which have the properties mentioned in Section 5.4.4, it will be possible to construct programs with more flexible syntax.

As another approach to inventing new necessary predicates, it can be considered to use some kind of analogy with known programs. It may be helpful for inventing new predicates to see how auxiliary predicates are used in the known programs. Since we also refer to several known programs when we make a program, such an approach seems to be more natural.

The problem of inventing new predicates seems to concern with an essential part of intelligent information processing that human beings do. It is certainly important for many intelligent AI systems to overcome the problem.

The main idea presented in Chapter 6 and Chapter 7 was how to introduce necessary predicates (states) or nonterminals with their appropriate models (interpretations). The problem of introducing new, unobserved sub-concepts that are necessary for representing a target concept is one of the most important and difficult problems in machine learning. Although there have been several approaches to this problem [Ban88, MB88], it seems that

none of the solutions proposed to date is satisfactory. Our results presented in the chapters are not exceptions. The class shown to be learnable is too restricted for many practical applications. The presented methods depend heavily upon the structural properties of DRLP's or SDG's. For example, the uniqueness of a left-most derivation is one of them. Hence, the methods are not directly applicable to (at least) the target class containing an ambiguous grammar. It is one of the most important future works to find more general and practical solutions to this challenging problem.

In Chapter 6, we considered the problem of regular languages. We proposed an inference algorithm for the class of regular models that performs polynomial update time inference using membership queries.

Shapiro [Sha81] treated the problem of model inference for regular languages by using the following type of program as a finite representation of a regular language.

$$
\begin{aligned}
&in([]).\\
&in([0,0|X]) \leftarrow in(X).\\
&in([1,1|X]) \leftarrow in(X).\\
&in([0,1,0|X]) \leftarrow in([1|X]).\\
&in([0,1,1|X]) \leftarrow in([0|X]).\\
&in([1,0,0|X]) \leftarrow in([1|X]).\\
&in([1,0,1|X]) \leftarrow in([0|X]).
\end{aligned}
$$

The program corresponds to the acceptor of strings over $\{0,1\}$ with an even number of 0's and an even number of 1's. If the target program to be inferred is such a program with only one predicate symbol, then it is sufficient that the given oracle can answer the truth about only the predicate symbol. Therefore, as mentioned in the introduction of Chapter 5, it is also important in considering extended model inference problems to investigate the power of logic programs with only one predicate symbol or with fully restricted number of predicate symbols.

Shapiro [Sha84] showed that an arbitrary alternating Turing machine can be simulated by a logic program with only one 3-ary predicate symbol. In such a logic program, however, the information of each state of the alternating Turing machine is embedded in one of the arguments of the predicate. Therefore, the problem of inferring such a program results in

that of inferring program over the language with countably many predicate symbols. Of course, this kind of reduction on the number of predicate symbols is out of our interest. We are interested in the essential relation between the power of logic programs and the number of predicate symbols allowed.

In Chapter 7, we considered the problem of learning SDL's. The efficiency of the algorithm given in the chapter might not be optimal. As shown in the proof of Theorem 7.5, it is ensured that the algorithm runs in time polynomial in $|N_0|$ and ℓ. The polynomial has a rather high degree. The polynomial is larger than, at least, the size of the largest hypothesis, $O(|N_0|^3\ell^6)$. If we can set each intermediate hypothetical grammar to an SDG, we may decrease the degree. While a grammar G in 2-standard form has, in the worst case, $|N| \times |\Sigma| \times (|N|+1)^2$ productions, an SDG G has at most $|N| \times |\Sigma|$ productions. Since the operation performed most frequently by the algorithm is to parse given counterexamples on each hypothesis G, this reduction in size of each hypothetical grammar will decrease the complexity of the learning algorithm. Obviously, such a restriction on hypothetical grammars also results in the development of an algorithm that produces an SDG as its output using normal equivalence queries and membership queries. The efficient learnability of SDL's from a minimally adequate teacher is still open.

Bibliography

[AI87] Setsuo Arikawa and Hiroki Ishizaka. Program synthesis by inductive inference. *Journal of Information Processing Society of Japan*, 28(10):1312–1319, 1987. (In Japanese).

[AIS92] Hiroki Arimura, Hiroki Ishizaka, and Takeshi Shinohara. Polynomial time inference of a subclass of context-free transformations. In *Proceedings of 5th Workshop on Computational Learning Theory*, 1992.

[AISO92] Hiroki Arimura, Hiroki Ishizaka, Takeshi Shinohara, and Setsuko Otsuki. A generalization of the least general generalization. Technical Report RIFIS-TR-CS-63, Kyushu University, 1992.

[Ang87a] Dana Angluin. Learning k-bounded context-free grammars. Research Report 557, Yale University Computer Science Dept., 1987.

[Ang87b] Dana Angluin. Learning regular sets from queries and counterexamples. *Information and Computation*, 75:87–106, 1987.

[Ang88] Dana Angluin. Queries and concept learning. *Machine Learning*, 2(4):319–342, 1988.

[Ari91] Hiroki Arimura. Completeness of depth-bounded resolution for weakly reducing programs. In Ikuo Nakata and Masami Hagiya, editors, *Software Science and Engineering*, pp. 227–245. World Scientific, 1991. World Scientific Series in Computer Science, Vol. 31.

[ASO91a] Hiroki Arimura, Takeshi Shinohara, and Setsuko Otsuki. A polynomial time algorithm for finite unions of tree pattern languages. In *Proc. of the 2nd International Workshop on Nonmonotonic and Inductive Logics*, 1991. To appear in LNCS.

[ASO91b] Hiroki Arimura, Takeshi Shinohara, and Setsuko Otsuki. Polynomial time inference of unions of tree pattern languages. In S. Arikawa, A. Maruoka, and T. Sato, editors, *Proc. ALT '91*, pp. 105–114. Ohmsha, 1991.

[ASY89] Setsuo Arikawa, Takeshi Shinohara, and Akihiro Yamamoto. Elementary formal system as a unifying framework for language learning. In *Proceedings of the Second Annual Workshop on Computational Learning Theory*, pp. 312–327. Morgan Kaufmann, 1989.

[Ban88] Ranan B. Banerji. Learning theories in a subset of a polyadic logic. In *Proc. Computational Learning Theory '88*, pp. 281–295, 1988.

[BR87] Piotr Berman and Robert Roos. Learning one-counter langugaes in polynomial time. In *Proceedings of the Twenty-Eighth IEEE Symposium on Foundations of Computer Science*, pp. 61–67, New York, 1987. The Institute of Electrical and Electronics Engineers.

[CL73] Chin-Liang Chang and Richard Char-Tung Lee. *Symbolic Logic and Mechanical Theorem Proving.* Academic Press, 1973.

[Gol67] E Mark Gold. Language identification in the limit. *Information and Control*, 10:447–474, 1967.

[Har79] Michael A. Harrison. *Introduction to Formal Language Theory.* Addison-Wesley, 1979.

[IA91] Hiroki Ishizaka and Setsuo Arikawa. Model inference. *Journal of Information Processing Society of Japan*, 32(3):236–245, 1991. (In Japanese).

[IAS92] Hiroki Ishizaka, Hiroki Arimura, and Takeshi Shinohara. Efficient inductive inference of primitive prologs from positive data. In S. Doshita, K. Furukawa, and T. Nishida, editors, *Proc. ALT '92*, pp. 135–146, 1992.

[Ish88a] Hiroki Ishizaka. Model inference incorporating generalization. *Journal of Information Processing*, 11(3):206–211, 1988.

[Ish88b] Hiroki Ishizaka. A note on predicate invention. Technical Memorandum TM-0631, ICOT, 1988. (In Japanese).

[Ish89a] Hiroki Ishizaka. Inductive inference of regular languages based on model inference. *International journal of Computer Mathematics*, 27:67–83, 1989.

[Ish89b] Hiroki Ishizaka. Learning simple deterministic languages. In *Proceedings of the 2nd Annual Workshop on Computational Learning Theory*, pp. 162–174. Morgan Kaufmann, 1989. To appear in Machine Learning.

[Ish90] Hiroki Ishizaka. Polynomial time learnability of simple deterministic languages. *Machine Learning*, 5(2):151–164, 1990.

[JLMM88] J-L.Lassez, M. J. Maher, and K. Marriott. Unification revisited. In J. Minker, editor, *Foundations of Deductive Databases and Logic Programming*, pp. 587–625. Morgan Kaufmann, 1988.

[Ley70] J. C. Leynolds. Transformational systems and the algebraic structure of atomic formulas. In B. Meltzer and D. Michie, editors, *Machine Intelligence 5*, pp. 135–152. Edinburgh University Press, 1970.

[Lin89a] Xiaofeng Ling. Inventing theoretical terms in inductive learning of functions - search and constructive methods. In Zbigniew W. Ras, editor, *Methodologies for Intelligent Systems, 4*, pp. 332–341. North-Holland, October 1989.

[Lin89b] Xiaofeng Ling. Learning and invention of horn clause theories - a constructive method. In Zbigniew W. Ras, editor, *Methodologies for Intelligent Systems, 4*, pp. 323–331. North-Holland, October 1989.

[Llo84] John W. Lloyd. *Foundations of Logic Programming.* Springer-Verlag, 1984.

[Lov78] Donald W. Loveland. *Automated Theorem Proving: A Logical Basis.* North-Holland, 1978.

[MB88] Stephen Muggleton and Wray Buntine. Machine invention of first-order predicates by inverting resolution. In *Proc. 5th International Conference on Machine Learning*, pp. 339–352, 1988.

[MNL88] K. Marriott, L. Naish, and J-L. Lassez. Most specific logic programs. In *Logic Programming: Proceedings of the Fifth International Conference and Symposium*, pp. 910–923. MIT Press, 1988.

[Mug90] Stephen Muggleton. Inductive logic programming. In S. Arikawa, S. Goto, S. Ohsuga, and T. Yokomori, editors, *Proc. ALT '90*, pp. 42–62. Ohmsha, 1990.

[Pit89] Leonard Pitt. Inductive inference, dfas, and computational complexity. In K. P. Jantke, editor, *Proc. AII '89, LNAI 397*, pp. 18–44. Springer-Verlag, 1989.

[Plo70] Gordon D. Plotkin. A note on inductive generalization. In B. Meltzer and D. Michie, editors, *Machine Intelligence 5*, pp. 153–163. Edinburgh University Press, 1970.

[PW88] Leonard Pitt and Manfred K. Warmuth. Prediction preserving reducibility. Technical Report UCSC-CRL-88-26, University of California, Santa Cruz, November 1988. Preliminary version appeared in Proceedings of the 3rd Annual IEEE Conference on Structure in Complexity Theory, pp.60–69, June, 1988.

[Sak90] Yasubumi Sakakibara. Inductive inference of logic programs based on algebraic semantics. *New Generation Computing*, 7:365–380, 1990.

[Sha81] Ehud Y. Shapiro. Inductive inference of theories from facts. Technical Report 192, Yale University Computer Science Dept., 1981.

[Sha82] Ehud Y. Shapiro. *Algorithmic program debugging.* PhD thesis, Yale University Computer Science Dept., 1982. Published by MIT Press, 1983.

[Sha84] Ehud Y. Shapiro. Alternation and the computational complexity of logic programs. *J. Logic Programming*, 1:19–33, 1984.

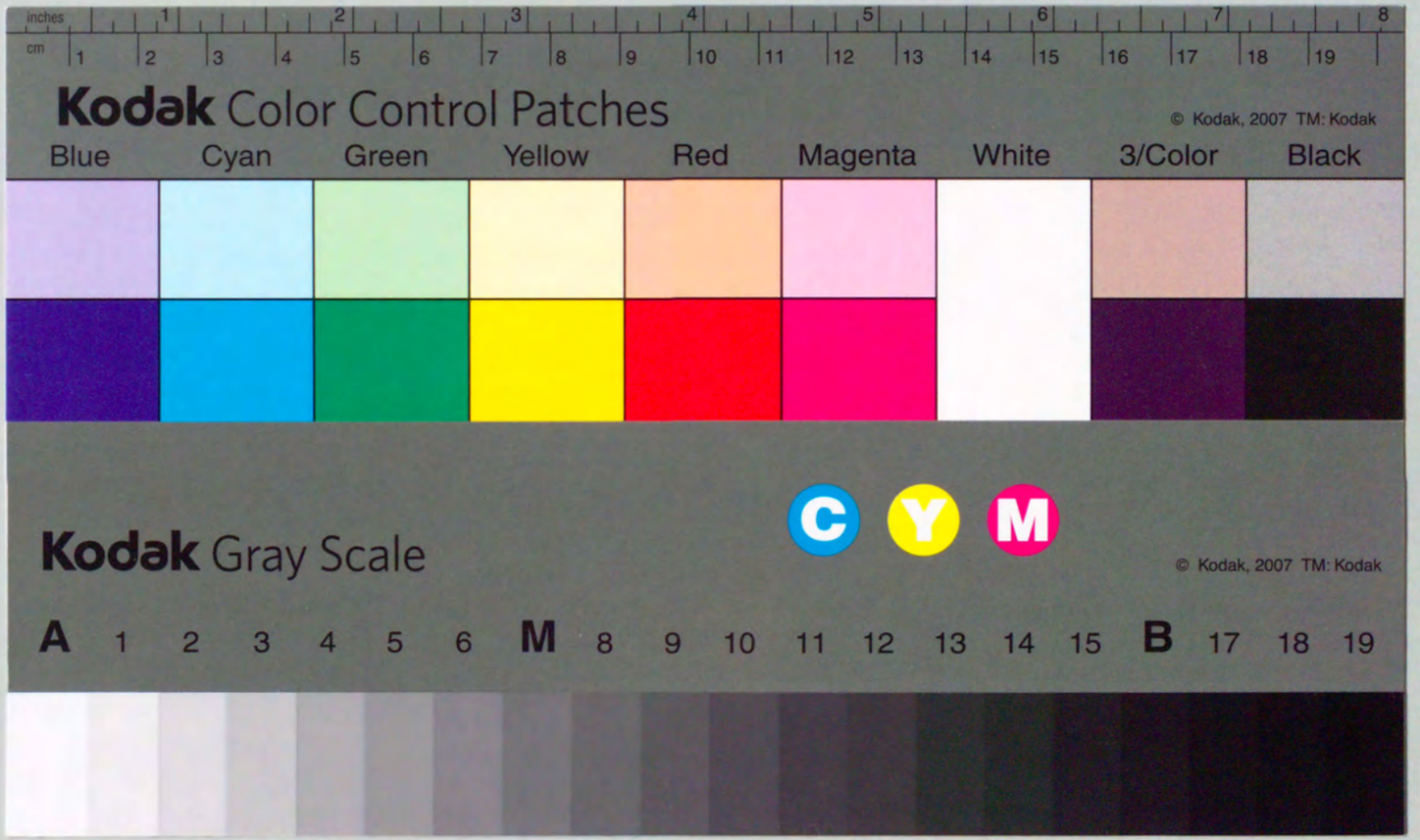
[Shi90] Takeshi Shinohara. Inductive inference of monotonic fomal systems from positive data. In S. Arikawa, S. Goto, S. Ohsuga, and T. Yokomori, editors, *Proc. ALT '90*, pp. 339–351. Ohmsha, 1990.

[Shi91] Takeshi Shinohara. Inductive inference of monotonic formal systems from positive data. *New Generation Computing*, 8:371–384, 1991.

[vEK76] M. H. van Emden and R. A. Kowalski. The semantics of predicate logic as a programming language. *Journal of ACM*, 23(4):733–742, 1976.

[Yam89] Akihiro Yamamoto. *Studies on Unification in Logic Programming*, 1989. Doctor thesis, Kyushu Univeristy.

[Yok88] Takashi Yokomori. Learning simple languages in polynomial time. In *Proc. of SIG-FAI*, pp. 21–30. Japanese Society for Artificial Intelligence, June 1988.


inches
cm
Kodak Color Control Patches
© Kodak, 2007 TM: Kodak
Blue
Cyan
Green
Yellow
Red
Magenta
White
3/Color
Black
Kodak Gray Scale
C
Y
M
© Kodak, 2007 TM: Kodak
A 1 2 3 4 5 6 M 8 9 10 11 12 13 14 15 B 17 18 19