九州大学学術情報リポジトリ
Kyushu University Institutional Repository

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：

Generalization and Predicate Invention in Learning Logic Programs

石 坂 裕 毅

Generalization and Predicate Invention in Learning Logic Programs

Hiroki Ishizaka

International Institute for Advanced Study of Social Information Science (IIAS-SIS)

FUJITSU LABORATORIES LTD.

140, Miyamoto, Numazu, Shizuoka 410-03, Japan

E-mail : hiro@iias.flab.fujitsu.co.jp

Abstract

We consider the problem of learning logic programs from examples. In this thesis the problem is treated as a model inference problem given by Shapiro. The theory of model inference is known to be a very elegant framework for inductive inference over the first order predicate logic. Since the class of logic programs is an instance of the first order predicate logic, the theory is also applicable to inductive inference of logic programs. We reconsider the model inference problem for logic programs from the following two viewpoints: using generalization and inventing auxiliary predicates in learning logic programs.

In the first half of the thesis, the applicability of least generalizations in learning logic programs is considered. One aspect of learning is to memorize a lot of experiences and generalize them appropriately. Such an appropriate generalization over the domain of first order words was given by Plotkin. We discuss the usage of the least generalization for constructing part of a target logic program, that is, each head of clauses in the program.

Furthermore, we give an algorithm for learning the class of very restricted logic programs called primitive Prologs using minimal multiple generalizations. The minimal multiple generalization is a natural extension of the least generalization. The minimal multiple generalization generalizes given first order words by several words, while the least generalization does by a single word. The property of the minimal multiple generalization makes it possible to perform fine generalization and to construct the heads of several clauses in a target program at the same time. One of the learning algorithm presented in this thesis constructs the heads of a program using the minimal multiple generalization and will be shown to learn the class of primitive Prologs efficiently.

The second half of the thesis is concerned with the problem of inventing auxiliary predicates in learning logic programs. In general, a logic program consists of several predicates. However, the main concept to be defined by the program is represented by one of them. In learning from examples, the given examples are focused on the main concept. Thus, another predicates concerned with auxiliary concepts are not observed in the given examples, so that the learning algorithm is obliged to invent such a predicate for itself if it is necessary.

In this thesis, we consider the problem in the framework of learning formal languages.

Formal languages such as regular languages can be represented by grammars or automata and the grammar or the automaton can be represented as a very restricted logic program, in which each predicate symbol corresponds to a nonterminal of the grammar or a state of the automaton. That is, the problem of inventing predicates in learning logic programs corresponds to that of inventing nonterminals or states in learning formal languages. We shall present two algorithms for learning formal languages: one for regular languages and the other for simple deterministic languages. For each algorithm, we develop methods for inventing new states of an automaton and new nonterminals of a grammar respectively. Both algorithms are shown to perform efficient learning with inventing states or nonterminals.

Acknowledgements

First and foremost, I am deeply grateful to my advisor Setsuo Arikawa of Kyushu University. He directed my research during my last three years at Kyushu University and gave me instructive suggestions and ceaseless encouragement. He was also willing to serve as my thesis supervisor, wade through poorly written drafts, and offered numerous improvements and suggestions. I would like to thank other members of my judging committee, Nagata Furukawa, Yasuo Kawahara, and Hiroto Yasuura for their many valuable comments.

I also express my gratitude to Takeshi Shinohara for many discussions on inductive inference and much helpful advice. In particular, the study developed in Chapter 4 was conducted by him. The chapter is also based on a joint work with Hiroki Arimura. Without his elegant framework of the minimal multiple generalization, the content of the chapter will not appear. Especially, the proofs of Lemma 4.5 and Lemma 4.6 are given by him. Akihiro Yamamoto read the initial draft of Chapter 3 and made useful comments. I appreciate the support from the people at RIFIS, Kyushu University, especially, Satoru Miyano and Ayumi Shinohara.

Thanks to Makoto Haraguchi for introducing me to least generalization and analogical reasoning. Takashi Yokomori directed my research during my first two years at Fujitsu. Discussions with him contributed to the content of Chapter 6 and Chapter 7. Four years at ICOT Research Center were also good experiences for me. I would like to thank Kazuhiro Fuchi, the director of ICOT Research Center, Koichi Furukawa, and Ryuzo Hasegawa for giving me the opportunity to conduct a part of this work into the Fifth Generation Computer Systems Project. Conversations with David Haussler, Klaus Jantke, Philip Laird, Stephen Muggleton, and Rolf Wiehagen during the course of my research were very helpful.

Thanks to the members of IIAS-SIS, Fujitsu Laboratories Ltd.. I would like to thank Toshio Kitagawa, the former president of IIAS-SIS, Hajime Enomoto, the former director of IIAS-SIS, Shigeru Sato, the director of IIAS-SIS, Mitsuhiko Toda, Kozo Sugiyama, Hajime Sawamura, for giving me the opportunity to pursue this work and their warm encouragement. Discussions with the members of Machine Learning Group at IIAS-SIS, Yuji Takada, Yasubumi Sakakibara, Kunihiko Hiraishi, Masahiro Matsuoka, and Takeshi Koshiba were

very fruitful. I am also grateful to all people at IIAS-SIS for providing an environment of friendship and stimulation.

Finally, I thank my parents.

Contents

1 Introduction 1

1.1 Generalization 2

1.2 Predicate Invention 3

1.3 Outline of Thesis 5

2 Preliminaries 7

2.1 Logic Programs 7

2.1.1 Basic definitions 7

2.1.2 Model theory for logic programs 10

2.1.3 Proof trees 10

2.2 Model Inference 11

2.3 Formal Languages 14

2.3.1 Finite-state automata and regular languages 14

2.3.2 Context-free grammars and languages 15

2.3.3 Model theory for context-free grammars 16

3 Least Generalization in Learning Logic Programs 19

3.1 Least Generalization 20

3.2 Model Preserving Instantiation 21

3.3 Program Heads and Least Generalizations 24

4 Learning Primitive Prologs from Positive Facts 27

4.1 Primitive Prologs and Model Inference from Positive Facts 29

4.2 Minimal Multiple Generalization 30

4.3 DMPLG's of Primitive Prologs 35
4.4 A Greedy Search Algorithm for the Body 41
4.5 Polynomial Update Time Inferability from Positive Facts 44

5 Model Inference with Predicate Invention **49**
5.1 An Extended Model Inference Problem 51
5.2 Problems in Extended Model Inference 52
5.3 A Simple Approach to the Problems 54
5.3.1 When a new predicate is necessary 54
5.3.2 The model of a new predicate 55
5.4 Examples of Restricted Programs 56
5.4.1 DRLP 56
5.4.2 LMLP 57
5.4.3 SDG 58
5.4.4 Simple recursive programs 59

6 Learning Regular Languages **63**
6.1 Regular Model Inference Problem 64
6.2 An Extended Model of a Regular Model 68
6.3 A Regular Model Inference Algorithm 70
6.4 Correctness of the Algorithm 74
6.5 Time Complexity of the Algorithm 81

7 Learning Simple Deterministic Languages **83**
7.1 SDG and SDL 85
7.2 Learning via Queries 86
7.3 A Learning Algorithm for SDL 88
7.4 Diagnosing an Incorrect Hypothesis 89
7.5 Generating Nonterminals and Productions 91
7.6 Correctness and Complexity 93

8 Conclusion **97**

Chapter 1

Introduction

The problem of machine learning is one of the most important and difficult problems in developing intelligent systems. In this thesis, we consider the problem of learning logic programs from examples.. The theoretical framework concerned with the problem was first given by Shapiro [Sha81]. He formalized the problem as an *inductive inference* over the first order logic. Our framework of learning logic programs based on his formalization.

Here, we illustrate the framework of learning logic programs considered in this thesis. In learning from examples, the *example* indicates an input-output example of the program. For a logic program, such an example is represented by a ground atom. For example, suppose the following simple logic program for appending two lists:

$$\begin{array}{l} append([X|Y], Z, [X|W]) \leftarrow append(Y, Z, W). \\ append([], X, X). \end{array}$$

For the program, ground atoms such as

$$append([a, b], [c], [a, b, c]),\ append([a], [b], [a, b]),\ append([a], [], [a])$$

are correct examples which are computed (or proved) by the above logic program and ground atoms such as

$$append([], [a], []),\ append([], [], [a]),\ append([a, b], [c], [c, a, b])$$

are incorrect ones. The problem of learning logic programs considered in this thesis is to construct a program as above from some information about its correct and incorrect examples.

By the theory of logic programs, the set of all correct examples for a program can be identified with the *least Herbrand model* of the program. Thus the learning logic programs from examples is formalized as the problem to find a logic program from information about the least Herbrand model of the program. Shapiro called such a problem a *model inference problem*. In this thesis, we reconsider the problem from the two viewpoints: using generalization and inventing auxiliary predicates in learning logic programs.

1.1 Generalization

We often use our previous experience to solve the problem we are now faced with. The behavior can be seen as a process to solve the problem using something learned from the experiences. However, each individual knowledge obtained from the experiences seldom completely match the situation of the current problem. The direct use of the collection of individual knowledge is not so flexible to be applied to actual problem solving. Thus, the knowledge should be generalized to be applicable even in a different situation. On the other hand, every knowledge cannot be applicable to solve a problem. Thus, the generalization should be made moderately.

Plotkin [Plo70] formalized such a moderate generalization over the domain of first order words (a word is a term or an atom) as a *least generalization*. For example, suppose that the following facts about the predicate $append(_,_,_)$ are known:

$$append([a,b],[c],[a,b,c]),\ append([a],[b],[a,b]),\ append([a],[],[a]),\ append([b],[],[b]).$$

Then, the atoms $append(X,Y,Z)$ or $append([X|Y],Z,W)$ are generalizations of them and $append([X|Y],Z,[X|W])$ is a least generalization, where X,Y,Z,W are variables. That is, a generalization is obtained from the individual atoms by replacing each different term occurring at the same position in the atoms by a different variable. On the other hand, the least generalization is a generalization preserving the properties common to the all atoms as much as possible.

We can find from the above example that the least generalization corresponds to the head of a clause in the *append* program. This leads to one motivation of our study. That

is, we consider the applicability of the least generalization to construct a part of a target program.

Recently, Arimura [ASO91b] proposed the notion of *minimal multiple generalization* which is a natural extension of the least generalization. The minimal multiple generalization generalizes given ground atoms by several atoms, while the least generalization does by a single atom. This extension realizes more flexible generalization. For example, suppose the following correct examples of the *append* program.

$$append([],[],[]),\ append([a],[],[a]),\ append([],[b],[b]),$$
$$append([a],[b],[a,b]),\ append([a,b],[c],[a,b,c]).$$

Then a least generalization of them is $append(X,Y,Z)$. On the other hand, the pair $\{append([],X,X), append([X|Y],Z,[X|W])\}$ of atoms is a minimal multiple generalization of them. Although the least generalization represents all ground atoms with predicate symbol $append(_,_,_)$, the minimal multiple generalization represents more restricted ones. Thus, a minimal multiple generalization is finer than a least generalization.

Furthermore, since a logic program consists of several clauses in general, it is preferable to generalize ground atoms by several atoms. That is, we can expect to construct the several heads of clauses at the same time by minimal multiple generalization. In fact, the above minimal multiple generalization corresponds to the pair of heads of clauses in the *append* program.

In Chapter 3 and Chapter 4, we discuss the problem of applicability of the least generalization and the minimal multiple generalization in learning logic programs.

1.2 Predicate Invention

In learning logic programs from examples, the most difficult problem is to find auxiliary predicates which are necessary for a target program but not observed in examples of the target program. For example, we know that a program for reversing a list can be written

using an auxiliary predicate $concat(_, _, _)$ as follows:

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

Of course, we can make another *reverse* program using different auxiliary predicates, e.g. $append(_, _, _)$. However, the important point is that no information about such auxiliary predicates explicitly appears in the examples of $reverse(_, _)$. Thus, in learning logic programs in which an auxiliary predicate is essential, the learner has to invent the predicate for itself.

Shapiro called such auxiliary predicates *theoretical terms* and assumed, in the theory of model inference, that the information about them is also given to the learner. That is, the given information in model inference is extended to the entire model of some specific program but not focused on examples of some specific predicate in the program. From the viewpoint of learning from examples, the assumption seems to be so restrictive. The problem of inventing new predicates is inevitable in learning logic programs.

Recently, several researchers are trying to solve this challenging problem [MB88, Mug90, Ban88, Lin89b, Lin89a]. The second part of this thesis is also motivated by the problem. Especially, in this thesis, we consider the problem in learning formal languages. Formal languages such as regular languages can be represented by grammars or automata. A grammar or an automaton can be represented as a restricted logic program, in which each predicate symbol corresponds to a nonterminal of the grammar or a state of the automaton. The restriction is helpful for us to overcome the difficulty of the problem.

In this thesis, we shall present two algorithms for learning formal languages. One is for regular languages and another is for simple deterministic languages. For each algorithm, we shall develop a method for inventing new states of an automaton and new nonterminals of a grammar. We shall show that both algorithms perform efficient learning with inventing states or nonterminals.

1.3 Outline of Thesis

In Chapter 2, we make preparations for the discussions developed in the succeeding chapters. First, we give definitions concerned with logic programs. Then we define the problem of learning logic programs as a model inference problem. Basic notions and the notation concerned with formal language theory are also introduced for the discussions in Chapter 6 and Chapter 7.

In Chapter 3, we discuss the applicability of the least generalization in learning logic programs. We introduce a notion of *d-model preserving instance* of a logic program. The d-model preserving instantiation is a kind of program transformation and shown to preserve the least Herbrand model of an original program. We show a relation between a least generalization and a d-model preserving instantiation.

In Chapter 4, we discuss the applicability of minimal multiple generalization in learning logic programs. We show that a class of very restricted logic programs called *primitive Prologs* is efficiently learned from only correct examples. The class of primitive Prologs is a sub-class of k-clause linear Prologs which is known to be learnable from only correct examples but not proven to be efficiently learnable. We present an algorithm which learns the class of primitive Prologs efficiently. The algorithm constructs the heads of clauses in a desired program using the minimal multiple generalization.

In Chapter 5, we discuss the problem of predicate invention in learning logic programs. We consider what problem is essential in inventing new predicates. Then we propose several classes of restricted logic programs for which the problem might be made clear. In fact, two of them, the class of *deterministic regular logic programs* and the class of *simple deterministic grammars* will be shown to be learned efficiently by overcoming the problem in the succeeding two chapters.

In Chapter 6, we discuss the problem of learning regular languages. A regular language can be represented by a very simple logic program, called a deterministic regular logic program, which exactly simulates a deterministic finite-state automaton. We give an algorithm which efficiently learns any deterministic regular logic program. The algorithm is based on the model inference algorithm given by Shapiro [Sha81, Sha82]. However, the presented

algorithm has the ability, which Shapiro's algorithm does not have, to invent new necessary predicates (states) automatically. We develop a method for extracting the information about the invented predicates from the information about one target predicate, which corresponds to the initial state of the target automaton. We show the validity of the method and the correctness of the algorithm. We also analyze the time complexity of the algorithm.

In Chapter 7, we extend the method in Chapter 6 for learning simple deterministic languages. We give an algorithm for learning simple deterministic languages with inventing necessary nonterminals. Angluin [Ang87a] showed that the class of k-bounded context-free grammars is learnable in polynomial time using membership queries, nonterminal membership queries and equivalence queries. Contrastively, the algorithm presented in Chapter 7 acquires necessary nonterminals for itself. Hence it does not require the nonterminal membership queries. We show the correctness of the algorithm and discuss its time complexity.

Finally, in Chapter 7, we conclude this thesis by summarizing the results presented in this thesis and stating some future problems.

Chapter 2

Preliminaries

In this chapter, we give some basic notions and notational conventions needed in this thesis. We use the fundamental concepts from first order logic and formal language theory. More precise information on these concepts would be found in [CL73, Lov78, Llo84, Har79].

In Section 2.1, we give definitions concerned with logic programs. In Section 2.2, we define a problem of learning logic programs according to Shapiro's theory of model inference [Sha81, Sha82]. In Section 2.3, we introduce some definitions on formal language theory necessary for the discussions in Chapter 6 and Chapter 7.

2.1 Logic Programs

2.1.1 Basic definitions

Let $\mathcal{L}$ be a first order language. Γ, Π, and X denote the set of function symbols, the set of predicate symbols, and the set of variable symbols of $\mathcal{L}$, respectively. We regard a constant symbol as a 0-ary function symbol and assume that Γ has at least one constant symbol throughout this thesis.

We adopt some informal notational conventions for the symbols. Variable symbols will normally be denoted by the letters U, V, W, X, Y, and Z (possibly subscripted). Constant symbols, that is, 0-ary function symbols will normally be denoted by the letters a, b, and c (possibly subscripted). Other function symbols will normally be denoted by the letters

f, g, and h (possibly subscripted). Predicate symbols will normally be denoted by the letters p, q, and r (possibly subscripted). Occasionally, it will not be convenient to apply these conventions rigorously. In such a case, possible confusion will be avoided by the context.

A *clause* is a well-formed formula of the form:

$$\forall(A_1 \vee \ldots \vee A_m \vee (\neg B_1) \vee \ldots \vee (\neg B_n))$$

where $A_1, \ldots, A_m, B_1, \ldots, B_n$ are atoms and $m, n \leq 0$. We denote the above clause by

$$A_1, \ldots, A_m \leftarrow B_1, \ldots, B_n.$$

The clause with $m = n = 0$ is called the *empty clause* and denoted by $\Box$.

A *definite clause* is a clause of the form:

$$A \leftarrow B_1, \ldots, B_n.$$

A is called the *head* of the clause and the sequence $B_1, \ldots, B_m$ of atoms is called the *body* of the clause. We mainly deal with definite clauses. So, hereafter, we call them just *clauses.* For a clause C, the head of C is referred by $head(C)$ and the body of C by $body(C)$. A clause with empty body, that is in the case $n = 0$, is called a *unit clause.* We identify a unit clause $A \leftarrow$ with the atom A. A *logic program* (program, for short) is a finite set of clauses. For a program $P = \{C_1, \ldots, C_n\}$, the collection of $head(C_i)$ $(1 \leq i \leq n)$ is said to be the *program heads* of P.

Example 2.1 The following is a typical program for appending two lists:

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

where $[]$ is a constant and $[_|_]$ is a binary function interpreted as an empty list and a list constructor, respectively. Usually, a list $[a_1|[a_2| \cdots [a_n|[]] \cdots]$ is abbreviated as $[a_1, a_2, \ldots, a_n]$.

A *word* is either a term or an atom. An *expression* is either a word, a clause, or a program. The size of an expression e, denoted by $size(e)$, is the number of occurrences of symbols appearing in the expression e. For a set S of expression, $size(S)$ is defined as $\sum_{e \in S} size(e)$ and $|S|$ as the number of elements in S.

A (possibly empty) finite sequence of integers of the form $\langle i_1, \ldots, i_n \rangle$ is called an *index*. For a word W and each sub-word t of W, the *index* I of t in W is defined inductively as follows:

1. If $W = t$, then $I = \langle\ \rangle$.
2. If W is of the form $\varphi(t_1, \ldots, t_m)$ and t is the sub-word of t_i such that the index of t in t_i is $\langle j_1, \ldots, j_n \rangle$, then $I = \langle i, j_1, \ldots, j_n \rangle$.

Let I be the index of a sub-word in a word W. Then $W(I)$ denotes the sub-word.

For an expression E, $var(E)$ denotes the set of variables appearing in the expression E. An expression with no variables is said to be *ground*. The *Herbrand base* of $\mathcal{L}$, denoted by $\mathcal{B}_{\mathcal{L}}$, is the set of all ground atoms over $\mathcal{L}$. The set of all ground terms over $\mathcal{L}$ is called the *Herbrand universe* of $\mathcal{L}$ and denoted by $\mathcal{U}_{\mathcal{L}}$.

A *substitution* θ is a finite set of the form $\{X_1/t_1, \ldots, X_n/t_n\}$, where each X_i is a variable, each t_i is a term distinct from X_i and the variables $X_1, \ldots, X_n$ are mutually distinct. Each element X_i/t_i is called a *binding* for X_i. The set of variables $\{X_1, \ldots, X_n\}$ is called the *domain* of the substitution θ and denoted by $dom(\theta)$. A substitution θ is called a *ground substitution* if the t_i's are all ground terms. For two substitution $\theta = \{X_1/t_1, \ldots, X_n/t_n\}$ and $\sigma = \{Y_1/s_1, \ldots, Y_m/s_m\}$, the *composition* of θ and σ, denoted by $\theta\sigma$, is defined as the substitution obtained from the set

$$\{X_1/t_1\sigma, \ldots, X_n/t_n\sigma, Y_1/s_1, \ldots, Y_m/s_m\}$$

by deleting any binding $X_i/t_i\sigma$ for which $X_i = t_i\sigma$ and deleting any binding Y_j/s_j for which $Y_j \in dom(\theta)$.

Let $\theta = \{X_1/t_1, \ldots, X_n/t_n\}$ be a substitution and E be an expression. Then $E\theta$, the *instance* of E by θ, is the expression obtained from E by simultaneously replacing each occurrence of the variable X_i by the term t_i $(1 \leq i \leq n)$. If $E\theta$ is ground, then $E\theta$ is called a *ground instance* of E and $G(E)$ denotes the set of all ground instances of E. Let E and F be expressions. E (F, respectively) is said to be a *variant* of F (E, respectively), denoted by $E \equiv F$, if there exist substitutions θ and σ such that $E\theta = F$ and $E = F\sigma$.

Let $S = \{w_1, \ldots, w_n\}$ be a finite set of words. A substitution θ is a *unifier* of S if

$$w_1\theta = w_2\theta = \cdots = w_n\theta.$$

If there exists a unifier for S, then S is said to be *unifiable*. Also, a word w_1 is said to be *unifiable* with a word w_2 if the set $\{w_1, w_2\}$ is unifiable. For a unifiable set S, a unifier θ of S is called a *most general unifier* if, for every unifier σ of S, there exists a substitution γ such that $\sigma = \theta\gamma$.

2.1.2 Model theory for logic programs

An *Herbrand interpretation* (interpretation, for short) is a subset of $\mathcal{B}_{\mathcal{L}}$. Let $C = A \leftarrow B_1, \ldots, B_n$ $(n \geq 0)$ be a clause and M be an interpretation. A ground atom α is said to be *covered by C with respect to M* if there exists a substitution θ such that $\alpha = A\theta$ and $B_i\theta \in M$ for each i $(1 \leq i \leq n)$. When $n = 0$, the substitution θ is sufficient to satisfy only the first condition $\alpha = A\theta$. The set of all ground atoms covered by C with respect to M is called the *derivative interpretation* (d-interpretation, for short) *of C with respect to M* and denoted by $C(M)$. Note that, for a unit clause C, it holds that $C(M) = G(C)$, since we do not distinguish between the unit clause and the atom appearing in the head of the clause.

A clause C is said to be *true* in an interpretation M if $C(M) \subseteq M$, otherwise C is said to be *false*. If every clause in a program P is true in M, then the program P is said to be *true* in M and *false* otherwise. If a program P is true in an interpretation M, then M is called an *Herbrand model* (model, for short) of P. When M is known to be a model of some program, the above defined derivative interpretation of C with respect to M is restated as the *derivative model* (d-model, for short) *of C with respect to M*.

Van Emden and Kowalski [vEK76] showed the *model intersection property*, that is ,the intersection of models of a program P is also a model of P. The intersection of all models of P is called the *least Herbrand model* of P and denoted by $M(P)$. M_P will be often used instead of $M(P)$ to decrease the number of parentheses.

2.1.3 Proof trees

For a program P and a ground atom α, a *derivation tree* of α on P is a finite tree that satisfies the following conditions:

1. Each node of the tree is a ground atom.

2. The root node is α.

3. For each internal node A and its children $B_1, \ldots, B_n$ $(n \geq 1)$, $A \leftarrow B_1, \ldots, B_n$ is a ground instance of a clause in P.

For a program P and a ground atom α, a *proof tree* of α on P is a derivation tree of α on P such that each leaf of the tree is a ground instance of a unit clause in P.

For a ground atom α and a program P, $P \vdash \alpha$ denotes that there exists a proof tree of α on P and $P \vdash_n \alpha$ denotes that there exists such a proof tree with n nodes.

In this thesis, we assume some effective procedure which, for any given ground atom α and any given program P, constructs a proof tree of α on P if it exists. Such a procedure can be implemented using SLD-resolution [Llo84]. From the equivalence between the declarative semantics of logic programs defined by the least Herbrand model and the procedural semantics defined by SLD-resolution [Llo84], it holds that $\alpha \in M(P)$ if and only if $P \vdash \alpha$ for any ground atom α and a program P. Thus, the assumed procedure constructs a proof tree of any $\alpha \in M(P)$ on P. Conversely, if the procedure succeeds to construct a proof tree of $\alpha \in \mathcal{B}_{\mathcal{L}}$ of P, then it is ensured that $\alpha \in M(P)$.

Unfortunately, for a ground atom $\alpha \notin M(P)$, it is not decidable whether α is in $M(P)$ in general. In other words, the membership problem of a ground atom in $M(P)$ is semi-decidable. However, some restrictions on programs turn the problem to be decidable. Such restrictions are found in [ASY89, Yam89, Shi91]. Especially, in his paper [Ari91], Arimura showed that the problem for the class of *weakly reducing* programs is decidable. Each class of logic programs we shall discuss its learnability in this thesis is a sub-class of weakly reducing programs. Thus, in this thesis, we assume some procedure which, for a given program P and a ground atom α, returns a proof tree of α on P if $\alpha \in M(P)$, and "No" otherwise.

2.2 Model Inference

The model inference problem and algorithm were originally introduced by Shapiro [Sha81]. He formalized the learning problem on first order logic and gave some interesting learning strategies that takes full advantage of syntactic and semantic properties of logic. In this section, we review the model inference problem and algorithm for logic programs according

to [Sha82, IA91].

Let M be the the least Herbrand model of an unknown program. A *fact* about M is a pair of the form $\langle \alpha, V \rangle$ where $\alpha \in \mathcal{B}_{\mathcal{L}}$ and $V = true$ if $\alpha \in M$, $V = false$ otherwise. The atom α in a fact $\langle \alpha, V \rangle$ is called a *positive fact* (*negative fact*) if $V = true$ ($V = false$). An *enumeration of facts about* M is an infinite sequence $f_1, f_2, \ldots$ where each f_i is a fact about M and, for any $\alpha \in \mathcal{B}_{\mathcal{L}}$, α occurs in a fact $e_i = \langle \alpha, V \rangle$ for some $i \geq 1$. Usually, we assume some device called an *oracle for* M as a source of information about a target least Herbrand model M. The enumeration of facts about M is given by the oracle. In Chapter 5, Chapter 6, and Chapter 7, we assume an oracle which can give the information in different way from an enumeration of facts.

Let $\mathcal{L}$ be a first order language and M be the least Herbrand model of an unknown program over $\mathcal{L}$. Then a *model inference problem* is defined as follows:

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

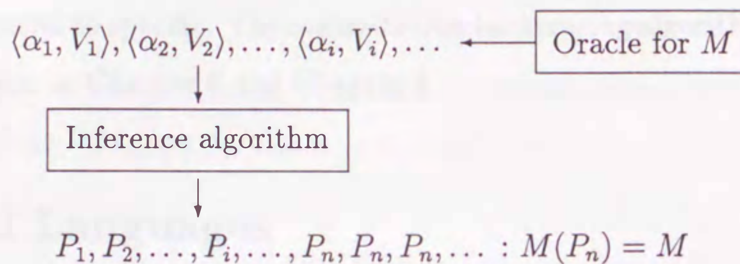
There are several criteria for success of inference and ways of giving information about M. Actually, in Chapter 4, Chapter 6, and Chapter 7, we consider the model inference algorithms under three different situations in which the adopted criterion and the way of giving information are different respectively. In this section, we give only the original situation in which a model inference algorithm is defined.

A *model inference algorithm* (inference algorithm, for short) $\mathcal{A}$ is an algorithm that iterates the process "input request $\rightarrow$ computation $\rightarrow$ output". Each output of $\mathcal{A}$, called a *conjecture* of $\mathcal{A}$, is a program. During computing a conjecture, such an inference algorithm usually produces several programs as candidates for the conjecture. Such candidate programs are called *hypotheses*. Let $P_1, P_2, \ldots$ be a sequence of conjectures of $\mathcal{A}$ given an enumeration of facts about the least Herbrand model of an unknown program as its input sequence. $\mathcal{A}$ is said to *converge* to a program P for the enumeration if there exists an integer $n \geq 1$ such that $P_i = P$ for any $i \geq n$. $\mathcal{A}$ is said to *identify* the model M *in the limit* if $\mathcal{A}$ converges to a program P such that $M(P) = M$ for any enumeration of facts about M. A class $\mathcal{M}$ of least Herbrand models of logic programs is said to be *inferable* if there exists an inference algorithm which identifies any model in $\mathcal{M}$. Identification in the limit defined

above, originally given by Gold [Gol67], is a typical criterion for the success of inductive inference. Figure 2.1 illustrates the framework of the model inference.

In this thesis, we treat the problem of learning logic program from examples as a model inference problem. Hence, in what follows, we often use the terms "infer" or "inferable" instead of the terms "learn" or "learnable". Furthermore, we often identify the class of programs and the class of least Herbrand models of the programs. Hence, sometimes, we say like "a class of programs is inferable".

Figure 2.1: The framework of the model inference



Let P be a conjecture of an inference algorithm. Then P is said to be *consistent* with a fact $\langle\alpha, V\rangle$ if it holds that $\alpha \in M(P)$ if and only if $V = true$. P is said to be *consistent* with a set of facts if P is consistent with every fact in the set. Let $P_1, P_2, \ldots$ be a sequence of conjectures of $\mathcal{A}$ for an enumeration $e_1, e_2, \ldots$ of facts about a model M and S_i be the set $\{e_1, e_2, \ldots, e_i\}$. The inference algorithm $\mathcal{A}$ is said to be *consistent* if P_i is consistent with S_i for any $i \geq 1$. $\mathcal{A}$ is said to be *conservative* if $P_i = P_{i-1}$ for any i such that P_{i-1} is consistent with e_i. $\mathcal{A}$ is said to be a *polynomial update time inference algorithm* if there exists some polynomial f such that, for any stage i, after $\mathcal{A}$ feeds the input e_i it produces the conjecture P_i in $f(size(S_i))$ steps.

Here, we give a general result on model inference obtained by Shapiro [Sha81, Sha82]. A program P is called *h-easy* if there exists a total recursive function h such that, for any $\alpha \in M(P)$, there exists a proof tree of α on P with at most $h(size(\alpha))$ nodes. A model M of a program is called *h-easy* if there exists an h-easy program such that $M(P) = M$.

Theorem 2.1 *Let h be a total recursive function. Then there exists a consistent and conservative inference algorithm that identifies any h-easy model in the limit.*

A simple inference algorithm based on the identification by enumeration technique [Gol67] supports the above theorem. Such an algorithm, however, is known to be too inefficient. In order to improve the inefficiency, Shapiro developed an incremental inference algorithm. Although his algorithm is essentially based on the enumeration technique, it can perform fairly efficient enumeration by taking full advantage of several logical properties desirable for inductive inference.

Shapiro gave two important sub-procedures for the incremental model inference. The one is the *contradiction backtracing algorithm* that detects a false clause in a false hypothesis in the target model M. The other is the *refinement operator* that enumerates clauses according to the order from general to specific. The contradiction backtracing algorithm is also essential in our algorithm given in Chapter 6 and Chapter 7.

2.3 Formal Languages

In this section, we introduce some basic definitions on formal language theory necessary for the discussions in Chapter 6 and Chapter 7.

An *alphabet* is a finite non-empty set of distinct symbols. For a given alphabet X, the set of all finite strings of symbols from X is denoted by X^*. The empty string is denoted by ε. X^+ denotes the set $X^* - \{\varepsilon\}$. For two string x and y, $x \cdot y$ denotes the concatenation of x and y. We may often omit $\cdot$, that is, the concatenation of x and y is simply denoted by xy. For a string $x = a_1 a_2 \cdots a_n$, $Pre_i(x)$ denotes the string $a_1 a_2 \cdots a_i$, and $Suf_i(x)$ denotes the string $a_{i+1} a_{i+2} \cdots a_n$. For a string x, $|x|$ denotes the length of x. If S is a finite set, then $|S|$ denotes the cardinality of S.

Let Σ be an alphabet. A *language* L over Σ is a subset of Σ^*. For a string x in Σ^* and a language L over Σ, let $\overline{x}L = \{y \mid xy \in L\}$ ($L\overline{x} = \{y \mid yx \in L\}$). The set $\overline{x}L$ ($L\overline{x}$) is called the *left(right)-derivative of L with respect to x*.

2.3.1 Finite-state automata and regular languages

A *deterministic finite-state automaton* (DFA, for short) is a 5-tuple $A = (Q, \Sigma, \delta, q_0, F)$, where

1. Q is a finite non-empty set. Each element of Q is called a *state.*
2. Σ is an alphabet such that $Q \cap \Sigma = \phi$.
3. δ is a function from $Q \times \Sigma$ into Q called the *transition function.*
4. q_0 is an element of Q called the *initial state.*
5. F is a subset of Q called the *final states.*

The transition function δ in the above definition can naturally be extended to the function from $Q \times \Sigma^*$ into Q as follows:

$$\delta(q, \varepsilon) = q \quad \text{and} \quad \delta(q, ax) = \delta(\delta(q, a), x).$$

The *size* of a DFA A, denoted by $size(A)$, is defined as the number of states of A.

A string $x \in \Sigma^*$ is said to be *accepted* by a DFA $A = (Q, \Sigma, \delta, q_0, F)$ if $\delta(q_0, x) \in F$. The set of all strings accepted by the DFA A is denoted by $L(A)$. That is, $L(A) = \{x \in \Sigma^* \mid \delta(q_0, x) \in F\}$. A language $L \subseteq \Sigma^*$ is called *regular* if there exists a DFA A such that $L = L(A)$. For any regular language L, the minimum size DFA accepting L is unique up to an isomorphism (i.e., a renaming of the states).

2.3.2 Context-free grammars and languages

A *context-free grammar* (CFG, for short) is a 4-tuple $G = (N, \Sigma, P, S)$, where

1. N is a finite non-empty set. Each element of N is called a *nonterminal symbol.*
2. Σ is an alphabet such that $N \cap \Sigma = \phi$. Each element of Σ is called a *terminal symbol.*
3. S is an element of N called the *start symbol.*
4. P is a finite set of *production rules* of the form $A \to \alpha$, where $A \in N$ and $\alpha \in (N \cup \Sigma)^*$.

A CFG G is in *Greibach normal form* if each production rule of G is of the form $A \to a\alpha$, where $A \in N, a \in \Sigma$ and $\alpha \in N^*$. Note that, in Chapter 7, we consider only ε-free grammars and languages. A CFG G is said to be in *m-standard form* if G is in Greibach normal form and, for each production $A \to a\alpha$ of G, $|\alpha| \leq m$. The *size* of a grammar G, denoted by $size(G)$, is the sum of $|N|$, $|\Sigma|$, $|P|$, and the sum of the lengths of the right-hand sides of all the productions in P.

For $\beta, \gamma \in (N \cup \Sigma)^*$, binary relation $\Rightarrow$ is defined as follows: $\beta \Rightarrow \gamma$ if and only if there exist $\delta_1, \delta_2 \in (N \cup \Sigma)^*$ and a production rule $A \to \alpha \in P$ such that $\beta = \delta_1 A \delta_2$ and $\gamma = \delta_1 \alpha \delta_2$. A *derivation from β to γ* is a finite sequence of strings $\beta = \beta_0, \beta_1, \cdots, \beta_n = \gamma$ such that, for each i, $\beta_i \Rightarrow \beta_{i+1}$. If there exists a derivation from β to γ, then we denote it by $\beta \Rightarrow^* \gamma$ and γ is said to be *generated* from β. That is, the relation $\Rightarrow^*$ is the reflexive and transitive closure of $\Rightarrow$.

In each step of a derivation, if the left-most nonterminal occurrence in β_i is replaced, then such a derivation is said to be a *left-most derivation* of γ from β. In what follows, unless otherwise stated, $\beta \Rightarrow^* \gamma$ denotes a left-most derivation of γ from β.

The *language of a nonterminal* A, denoted by $L(A)$, is the set of all $x \in \Sigma^*$ such that $A \Rightarrow^* x$. Similarly, for $\alpha \in N^*$, $L(\alpha)$ denotes the set of all $x \in \Sigma^*$ such that $\alpha \Rightarrow^* x$. (To emphasize the grammar being used, we sometimes use the subscript G, e.g., $S \Rightarrow_G x$ or $L_G(A)$.) The *language of a grammar* G, denoted by $L(G)$, is just $L(S)$, where S is the start symbol of G. A language L is called *context-free* if there exists a CFG G such that $L = L(G)$.

A *derivation tree* T of a grammar $G = (N, \Sigma, P, S)$ is a tree such that each internal node of T is labeled with an element of N, each leaf of T is labeled with an element of Σ and, for each internal node labeled with $A \in N$, there exists a production $A \to \alpha$ in P, where $\alpha \in (N \cup \Sigma)^*$ is the concatenation of the labels of its children in left-to-right order. Let T be a derivation tree of a grammar. The root label of T is denoted by $rt(T)$. The *frontier* of T, denoted by $fr(T)$, is the concatenation of the labels of its leaves in left-to-right order. A derivation tree T illustrates a derivation from $rt(T) \in N$ to $fr(T) \in \Sigma^*$.

2.3.3 Model theory for context-free grammars

As in the case of logic programs, we can define the notion of models for CFG's.

Let $G = (N, \Sigma, P, S)$ be a CFG. For each nonterminal $A \in N$, a *model* of A, denoted by $M(A)$, is a subset of Σ^+. A *model* M for the grammar G consists of a model of each nonterminal.

$$M = \{M(A_1), M(A_2), \ldots, M(A_{|N|})\}.$$

A *replacement* is a finite tuple (possibly empty) of pairs of a terminal string $y_i \in \Sigma^*$ and

a nonterminal $A_i \in N$:

$$\langle (y_1, A_1), \ldots, (y_n, A_n) \rangle.$$

Let $\rho = \langle (y_1, A_1), \ldots, (y_n, A_n) \rangle$ be a replacement and β be a string in $(N \cup \Sigma)^*$. ρ is said to be *compatible* with β if there are finite strings $x_0, \ldots, x_n \in \Sigma^*$ such that $\beta = x_0 A_1 x_1 A_2 \cdots A_n x_n$. If ρ is compatible with β, then the *instance* of β by ρ, denoted by $\rho[\beta]$, is the terminal string obtained from β by replacing each occurrence of A_i in β by the terminal string y_i. An empty replacement ρ is compatible with any terminal string x and $\rho[x] = x$.

Let M be a model for a grammar G. A production $A \to \alpha$ is said to be *incorrect* for M if there exists a replacement $\rho = \langle (y_1, A_1), \ldots, (y_n, A_n) \rangle$ that is compatible with α such that, for each i, $y_i \in M(A_i)$[1], but $\rho[\alpha] \notin M(A)$. A production is said to be *correct* for M if it is not incorrect for M.

Example 2.2 Consider the CFG $G = (\{S, A, B, C\}, \{a, b\}, P, S)$ with

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

Let M be a model such that, for each nonterminal $X \in N$, $M(X) = L(X)$, that is,

$$\begin{aligned} M = \{ \; & M(S) = \{a^m b^m \mid 1 \leq m\}, \; M(A) = \{a^{m-1} b^m \mid 1 \leq m\}, \\ & M(B) = \{a^{m-2} b^m \mid 2 \leq m\}, \; M(C) = \{b\}\}. \end{aligned}$$

Then, a production $A \to aBC$ is incorrect for M, because there exists a replacement $\rho = \langle (bb, B), (b, C) \rangle$ that is compatible with aBC such that $bb \in M(B), b \in M(C)$, but the string $\rho[aBC] = abbb$ is not in $M(A)$.

More generally, we have the following proposition.

Proposition 2.2 *Let $G = (N, \Sigma, P, S)$ be a CFG and M be a model for G such that, for each nonterminal $A \in N$, $M(A) = L(A)$. Then every production in P is correct for M.*

[1] When α has no nonterminal, then this condition is not necessary.

Chapter 3

Least Generalization in Learning Logic Programs

Generalization is one of the most important concepts in learning. Theoretical studies on a generalization over words are found in [Plo70, Lav78, MUR90].

By Plotkin's definition, a word w_1 is more general than a word w_2 if w_2 is an instance of w_1. For any clause C, since any ground atom a which is covered by C is an instance of the head of C, the head is a generalization of ground atoms in $C(M_P)$. Although, there exist several generalizations for a set of ground atoms, a typical generalization called a least generalization which uniquely exists in it and can be computed in time proportional to the size of the set. However, in general, the head of a clause C is not always the least generalization of $C(M_P)$. The problem considered in this chapter is, preserving the least Herbrand model of a program P, whether each clause C in P can be replaced by C' whose head is a least generalization of $C(M_P)$.

In order to answer the problem, we introduce a notion of Herbrand preserving transformation. A Herbrand preserving transformation is a kind of program transformations and allows us to preserve the least Herbrand model of an original program. We show that a substitution defined on the difference between the head of an original clause C and the alternative head obtained by the least generalization of $C(M_P)$ gives the Herbrand preserving transformation. That is, for any program P, there exists an instance P' of P such that $M(P) = M(P')$ and each instance C' in P' of $C \in P$ has a least generalization of $C(M_P)$ as its head.

Chapter 3

Least Generalization in Learning Logic Programs

Generalization is one of the most important concepts in learning. Theoretical studies on a generalization over words are found in [Plo70, Ley70, JLMM88].

By Plotkin's definition, a word w_1 is more general than a word w_2 if w_2 is an instance of w_1. For any clause C, since any ground atom α which is covered by C is an instance of the head of C, the head is a generalization of ground atoms in $C(M_P)$. Although, there exist several generalizations for a set of ground atoms, a typical generalization called a least generalization exists uniquely modulo $\equiv$ and can be computed in time polynomial in the size of the set. However, in general, the head of a clause C is not always the least generalization of $C(M_P)$. The problem considered in this chapter is, preserving the least Herbrand model of a program P, whether each clause C in P can be replaced by C' whose head is a least generalization of $C(M_P)$.

In order to answer the problem, we introduce a notion of d-model preserving instantiation. A d-model preserving instantiation is a kind of program transformation and shown to preserve the least Herbrand model of an original program. We show that a substitution defined as the difference between the head of an original clause C and the alternative head obtained by the least generalization of $C(M_P)$ gives the d-model preserving instantiation. That is, for any program P, there exists an instance P' of P such that $M(P) = M(P')$ and each instance $C' \in P'$ of $C \in P$ has a least generalization of $C(M_P)$ as its head.

In Section 3.1, we review the least generalization according to [Plo70]. In Section 3.2, we introduce a d-model preserving instance of a program and show that the instantiation preserves entire model of a program, that is, the least Herbrand model of the program. In Section 3.3, we show that a least generalization of $C(M_P)$ brings a d-model preserving instantiation.

This chapter is based on the paper [Ish88a].

3.1 Least Generalization

For two words w_1 and w_2, w_1 is said to be *more general than* w_2, denoted by $w_1 \succeq w_2$, if w_2 is an instance of w_1, that is, there exists a substitution θ such that $w_1\theta = w_2$. Note that if $w_1 \succeq w_2$ and $w_2 \succeq w_1$, then it holds that $w_1 \equiv w_2$. If $w_1 \succeq w_2$ but $w_1 \not\preceq w_2$, then it is denoted by $w_1 \succ w_2$. For a set S of words, a *generalization* of S is a word w such that, for any $u \in S$, $w \succeq u$.

Definition 3.1 For a set S of words, a *least generalization* of S, denoted by $lg(S)$, is a generalization w of S such that, for any generalization u of S, $u \succeq w$

From the above definition, if w_1 and w_2 are any two least generalizations, then it holds that $w_1 \equiv w_2$. That is, $lg(S)$ is unique modulo $\equiv$ if it exists.

Example 3.1 For the set

$$S = \{append([a,b],[c],[a,b,c]), append([a],[b],[a,b]), append([a],[],[a])\},$$

$append(X,Y,Z)$ or $append([X|Y],Z,W)$ are generalizations of S and $append([a|X],Y,[a|Z])$ is a least generalization of S.

Two words are said to be *compatible* if they are both terms or have the same predicate symbol. A set S of words is said to be *compatible* if any two words in S are compatible.

Theorem 3.1 (Plotkin [Plo70]) *Every non-empty finite set S of words has a least generalization if and only if S is compatible.*

Sub-words t_1 of a word w_1 and t_2 of w_2 such that $t_1 \neq t_2$ are said to be *replaceable* if they satisfy the following two conditions:

1. The index of t_1 in w_1 is equal to the index t_2 in w_2.
2. t_1 and t_2 begin with different function symbols or at least one of them is a variable.

Algorithm 3.1 given by Plotkin [Plo70] computes a least generalization of two compatible words w_1, w_2.

Algorithm 3.1: A least generalization algorithm

Input: Compatible words w_1, w_2.
Output: $lg(\{w_1, w_2\})$.
Procedure:

$W_1 := w_1$; $W_2 := w_2$;
while there exist t_1 and t_2 replaceable in V_1, V_2 **do**
  choose a variable x which does not occur in V_1, V_2;
  while there exists an index I such that $V_1(I) = t_1$, $V_2(I) = t_2$ **do**
    $V_1(I) := x$; $V_2(I) := x$
output V_1 $(= V_2)$

For any finite compatible set $S = \{w_1, w_2, \ldots, w_n\}$ of words, the least generalization $lg(S)$ can be computed by applying the above algorithm $n-1$ times iteratively, that is, $lg(S) = lg(w_1, lg(w_2, lg(\ldots, lg(w_{n-1}, w_n)\cdots)))$. Since the above algorithm runs in time polynomial in $size(w_1) + size(w_2)$, $lg(S)$ can be computed in time polynomial in $size(S)$.

3.2 Model Preserving Instantiation

In this section, we consider a d-model preserving instance of a program P which is a kind of program transformation. We show that the transformation preserves the least Herbrand model of an original program.

Let P be a program and C be a clause.

Definition 3.2 A *d-model preserving instance* (DMP, for short) of C with respect to P is an instance $C\theta$ of C such that, for any $M \subseteq M(P)$, $C\theta(M) = C(M)$.

Definition 3.3 A *d-model preserving instance* (DMP, for short) of P, denoted by $dmp(P)$, is a program obtained from P by replacing $C \in P$ with its d-model preserving instance with respect to P.

Example 3.2 Consider the program $P = \{C_0, C_1\}$ where

$$C_0 = member(X, [X|Y]),$$
$$C_1 = member(X, [Y|Z]) \leftarrow member(X, Z).$$

Let M be any subset of $M(P)$ and C_1' be the clause

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

Then, for any element $member(t, list)$ of M, the length of the *list* is at least 1. Hence, any ground atom covered by C_1 with respect to M is also covered by C_1' with respect to M, that is, $C_1'(M) \supseteq C_1(M)$. Since C_1' is an instance of C_1, it is clear that $C_1'(M) \subseteq C_1(M)$. Thus, the program $P' = \{C_0, C_1'\}$ is a DMP of P.

A similar notion to the DMP has been introduced by Marriott et al. [MNL88]. They called the notion a *more specific version*. A more specific version of a clause C is an instance of C that preserves all successful derivations concerned with C on a program P. That is, they defined the notion from the point of view of procedural semantics of logic programs. Since our definition seems to be more restrictive than that of Marriott's, the two notions may not be identical. However, the above definition of a DMP is sufficient and comfortable for our discussion below.

In the following, we show that a DMP of a program P has the same least Herbrand model as P's. In order to show the equivalence between two least Herbrand models, we use the mapping T_P which gives the fixpoint semantics of logic programs [vEK76, Llo84]. For a program P, the mapping $T_P : 2^{\mathcal{B}_\mathcal{L}} \to 2^{\mathcal{B}_\mathcal{L}}$, where $2^{\mathcal{B}_\mathcal{L}}$ is the power set of the Herbrand base, is defined by

$$T_P(I) = \bigcup_{C \in P} C(I).$$

For a mapping $f : 2^{\mathcal{B}_\mathcal{L}} \to 2^{\mathcal{B}_\mathcal{L}}$, $I \in 2^{\mathcal{B}_\mathcal{L}}$ is said to be a *fixpoint* of f if $f(I) = I$. The *least fixpoint* of f, denoted by $lfp(f)$, is a fixpoint $I \in 2^{\mathcal{B}_\mathcal{L}}$ of f such that $I \subseteq I'$ for any fixpoint I' of f.

Let n be a non-negative integer and ω be the set of all non-negative integers. Then $T_P \uparrow n$ and $T_P \uparrow \omega$ are defined as follows:

$$T_P \uparrow 0 = \phi$$
$$T_P \uparrow n = T_P(T_P \uparrow (n-1))$$
$$T_P \uparrow \omega = \bigcup_{n \in \omega} T_P \uparrow n$$

The following theorem gives the equivalence between model theoretic semantics and fixpoint semantics of logic programs.

Theorem 3.2 (van Emden and Kowalski [vEK76]) *For any program P, there exists $lfp(T_P)$ and $M(P) = lfp(T_P) = T_P \uparrow \omega$.*

Corollary 3.3 *For any program P, it holds that $M(P) = \bigcup_{C \in P} C(M(P))$.*

Here we show that a d-model preserving instantiation preserves the least Herbrand model.

Theorem 3.4 *For any program P, it holds that $M(P) = M(dmp(P))$.*

Proof: Suppose that P consists of clauses C_i $(1 \leq i \leq m)$ and $dmp(P)$ consists of clauses C'_i $(1 \leq i \leq m)$ where $C'_i(M) = C_i(M)$ for any $M \subseteq M(P)$. We show by induction that $T_{dmp(P)} \uparrow n = T_P \uparrow n$ for any non-negative integer n.

For $n = 0$, $T_{dmp(P)} \uparrow 0 = \phi = T_P \uparrow 0$.

Suppose that $T_{dmp(P)} \uparrow n = T_P \uparrow n$ for some non-negative integer $n > 0$. From Theorem 3.2, it holds that $T_P \uparrow n \subseteq M(P)$. On the other hand, since $C'_i(M) = C_i(M)$ for any $M \subseteq M(P)$, it holds that

$$C'_i(T_{dmp(P)} \uparrow n) = C'_i(T_P \uparrow n) = C_i(T_P \uparrow n).$$

Thus it holds that

$$T_{dmp(P)} \uparrow (n+1) = \bigcup_{C'_i \in dmp(P)} C'_i(T_{dmp(P)} \uparrow n) = \bigcup_{C_i \in P} C_i(T_P \uparrow n) = T_P \uparrow (n+1).$$

Hence, it holds that $T_{dmp(P)} \uparrow n = T_P \uparrow n$ for any non-negative integer n. With Theorem 3.2, this proves the theorem. □

3.3 Program Heads and Least Generalizations

For a clause C and a set of ground atoms M, from the definition of $C(M)$, it is clear that $head(C)$ is a generalization of M. Hence, for a program P and $C \in P$, $head(C)$ is a generalization of $C(M_P)$. Our interest in this section is whether $head(C)$ can be a least generalization of $C(M_P)$, that is, preserving the least Herbrand model of a program P, whether each clause C in P can be replaced by C' whose head is $lg(C(M_P))$.

In general, program heads are not always the least generalizations $lg(C(M_P))$. For example, consider the following well-known program that reverses a list:

$$P = \left\{ \begin{array}{l} reverse([X|Y], Z) \leftarrow reverse(Y, Z_1), concat(X, Z_1, Z). \\ reverse([], []). \\ concat(X, [Y|Z], [Y|W]) \leftarrow concat(X, Z, W). \\ concat(X, [], [X]). \end{array} \right\}.$$

Let C be the first clause in the above program. Then $C(M_P)$ consists of ground atoms of the form $reverse(list1, list2)$ where $list1 \neq []$. Since the length of $list2$ is equal to that of $list1$, it must be greater than 1. Hence, it holds that $lg(C(M_P)) \preceq reverse([X|Y], [Z|W]) \prec head(C)$.

Here, we show the validity of inferring program heads as least generalizations by showing the substitution θ such that $lg(C(M_P)) = head(C)\theta$ derives a d-model preserving instantiation. That is, even if each clause C in an original program is replaced by $C\theta$ whose head is a least generalization of $C(M_P)$, the least Herbrand model of the program is preserved.

Lemma 3.5 *Let P be a program and $C \in P$ be a clause. For the substitution θ such that $lg(C(M_P)) = head(C)\theta$, $C\theta$ is a DMP of C with respect to P.*

Proof: Suppose C be a clause of the form:

$$A \leftarrow B_1, B_2, \ldots, B_n. \quad (0 \geq n),$$

and A' be a least generalization of $C(M_P)$. Without loss of generality, we may assume that A' does not share variables with A and θ operates only the variables occurring in A, that is, $dom(\theta) \subseteq var(A)$.

We show that $C\theta(M) = C(M)$ for any $M \subseteq M(P)$.

For any $\alpha \in C(M)$, there exists substitution σ such that $\alpha = A\sigma$ and $B_i\sigma \in M$ for $1 \leq i \leq n$, where we may assume that $dom(\sigma) \subseteq var(C)$. From the definition of $C(M)$, if $M_1 \subseteq M_2$ then $C(M_1) \subseteq C(M_2)$. Hence, it holds that $C(M) \subseteq C(M_P)$. Since A' is a least generalization of $C(M_P)$, there exists a substitution η such that $\alpha = A'\eta = A\theta\eta$ and $dom(\eta) \subseteq var(A')$. Thus we have $\alpha = A\sigma = A\theta\eta$ for any $\alpha \in C(M)$. From the assumption on the domains of θ, σ, and η, there exists a substitution η' such that $\theta\eta\eta' = \sigma$. Such a substitution η' can be obtained from σ by deleting the binding X/t for each variable $X \in var(A)$. Hence, there exists a substitution $\theta\eta\eta'$ such that, for any $\alpha \in C(M)$, $\alpha = A\theta\eta\eta'$ and $B_i\theta\eta\eta' \in M$ for $(1 \leq i \leq n)$. Thus, it holds that $\alpha \in C\theta(M)$ for any $\alpha \in C(M)$ and $M \subseteq M(P)$.

Conversely, for any $\alpha \in C\theta(M)$, there exists a substitution σ such that $\alpha = A\theta\sigma$ and $B_i\theta\sigma \in M$ for $(1 \leq i \leq n)$. Thus $\alpha \in C(M)$ for any set M of ground atoms and $\alpha \in C\theta(M)$.

□

Corollary 3.6 *Let $P = \{C_1, \ldots, C_m\}$ be a program and θ_i be a substitution such that $lg(C_i(M(P))) = head(C_i)\theta_i$ $(1 \leq i \leq m)$. Then, the program $P' = \{C_1\theta_1, \ldots, C_m\theta_m\}$ is a DMP of P.*

We call the DMP of P defined in the above corollary as the *d-model preserving instance by least generalization* (DMPLG, for short) and denote it by $dmplg(P)$.

Theorem 3.7 *For any program P, $M(P) = M(dmplg(P))$.*

Proof: The theorem follows immediately from Theorem 3.4 and Corollary 3.6. □

By the above theorem, for each clause $C \in P$, $lg(C(M_P))$ can be the head of a clause in a program equivalent to P. On the other hand, from the argument about a least generalization of an infinite set [JLMM88], the following proposition holds.

Proposition 3.8 *For any set S of words, there exists a finite subset S' of S such that $lg(S) \equiv lg(S')$.*

With Corollary 3.3 this implies that each program head is inferable as a least generalization of an appropriate finite subset of positive facts. This suggests applicability of the least generalization to inferring program heads in learning logic programs.

Especially, for a deterministic program P in which $C_i(M_P) \cap C_j(M_P) = \phi$ for any two clauses in P such that $i \neq j$, the corresponding program heads

$$lg(C_1(M_P)), lg(C_2(M_P)), \ldots, lg(C_m(M_P))$$

can be obtained as a partition of sufficiently large given positive facts. At each step of inference, since there are only finitely many positive facts, all candidates of such a partition is effectively computable.

Of course, since the number of all partitions increases exponentially, such a naive search strategy is useless for an efficient model inference algorithm. Recently, however, a very useful notion and algorithm for the problem were given by Arimura [ASO91b]. The notion is called a *minimal multiple generalization* and the algorithm to calculate minimal multiple generalizations finds the program heads like $lg(C_1(M_P)), \ldots, lg(C_m(M_P))$ more directly and efficiently. In the next chapter, we give an inference algorithm equipped with the minimal multiple generalization algorithm.

Chapter 4

Learning Primitive Prologs from Positive Facts

In this chapter, we consider the problem of efficient inductive inference of primitive Prologs from positive facts. A primitive Prolog is a very restricted logic program. It has at most two clauses that consist of one unary predicate symbol. The class of primitive Prologs is a sub-class of *k-clause linear Prologs* that is known to be inferable from only positive facts but not proven to be polynomial update time inferable [Shi90]. One aim of this study is to investigate what sub-class of linear Prologs is inferable in polynomial update time.

On the other hand, Arimura[ASO91b, ASO91a] gave a notion of minimal multiple generalization that is a natural extension of Plotkin's least generalization. While a least generalization precisely covers a given set of words by one word, an minimal multiple generalization minimally covers the set by several words. For example, suppose that

$$\begin{aligned} M = \{ \; & app([],[],[]), app([a],[],[a]), app([],[b],[b]), \\ & app([a],[b],[a,b]), app([a,b],[c],[a,b,c]), \ldots \} \end{aligned}$$

is the least Herbrand model of a program for appending lists $app(X, Y, Z)$. Then a least generalization of M is $app(X, Y, Z)$. On the other hand, the pair

$$\{app([], X, X), app([X|Y], Z, [X|W])\}$$

of atoms is a minimal multiple generalization of M. Note that the pair corresponds to the heads of clauses in a normal append program.

In some inductive inference algorithms for logic programs such as GEMINI [Ish88a] or CIGOL [MB88], the least generalization plays a very important role in inferring heads of clauses. However, in general, a program consists of several clauses. In order to infer several heads using least generalizations, inference algorithm has to divide at first a given set of positive examples (finite subset of the least Herbrand model of a target program) into several appropriate subsets then it can get candidates for heads of clauses by computing a least generalization of each subset. This process, that is, dividing a set of positive examples appropriately and then generalizing each obtained subset of examples, exactly corresponds to the calculation of a minimal multiple generalization. Hence, minimal multiple generalization is more useful than least generalization in inductive inference of logic programs. Another aim of this study is to investigate an effective application of the minimal multiple generalization to inductive inference of logic programs.

As described in the previous chapter, appropriate program heads can be obtained only from positive facts. On the other hand, in this chapter, we develop an algorithm to search for an appropriate body of a clause under the situation in which only positive facts are given. The algorithm takes full advantage of properties of the DMPLG of a primitive Prolog P.

In Section 4.1, we introduce the definition of primitive Prologs and an additional definition for model inference from positive facts. In Section 4.2, we introduce the notion of a minimal multiple generalization. Then, we discuss the properties of minimal multiple generalizations of a least Herbrand model of a primitive Prolog and show that the heads of a DMPLG of the program is contained in the minimal multiple generalizations for a sufficiently large set of positive facts. In Section 4.3, we show some properties of a DMPLG of a primitive Prolog. The properties are useful for an algorithm which searches for the body of the DMPLG. In Section 4.4, we give the greedy search algorithm for the body of a DMPLG. If the algorithm is given the heads of the DMPLG and a sufficiently large set of positive facts, then it can reconstruct the DMPLG. Two model inference algorithms for primitive Prologs from positive facts are given in Section 4.5. One is a consistent and conservative polynomial update time inference algorithm which requires a unit clause in a target program as a hint. The other is a consistent but not conservative polynomial update time inference algorithm which requires no hint.

This chapter is based on the paper [IAS92, AIS92, AISO92].

4.1 Primitive Prologs and Model Inference from Positive Facts

First, we give the definition of primitive Prologs.

Definition 4.1 A *primitive Prolog* P is a program that satisfies the following conditions:

(a) Only one unary predicate symbol appears in P.

(b) P consists of at most two clauses.

(c) If P consists of two clauses, then both heads of the clauses have no common instance.

(d) Atoms appearing in the body of a clause are most general atoms, as $p(X)$.

(e) Variables appearing in the body of a clause are mutually distinct and also appear in the head of the clause.

In a word, a primitive Prolog is a program of the form:

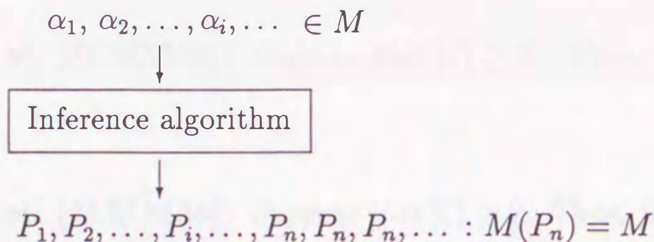
$$p(t[X_1, \ldots, X_m]) \leftarrow p(X_1), \ldots, p(X_m).$$
$$p(s).$$

where $G(t[X_1, \ldots, X_m]) \cap G(s) = \phi$. We use $t[X_1, \ldots, X_m]$ to denote a term which contains mutually distinct variables $X_1, \ldots, X_m$. Note the difference between the notations $t[X_1, \ldots, X_m]$ and $t(X_1, \ldots, X_m)$. While the latter denotes the term whose principal functor is t and its arguments are $X_1, \ldots, X_m$, the former may denote any term which contains mutually distinct variables $X_1, \ldots, X_m$. For example, $t[X, Y, Z]$ may denote a term $f(X, f(X, Y, f(Y, Z, U)), Z)$ or a list $[X, Y, U, U, Z]$.

In this chapter, we consider the polynomial update time inferability from only positive facts. The difference between the original model inference defined by Shapiro and the model inference considered here is just the way of giving information about M. In *model inference from positive facts*, given information about M to an inference algorithm are restricted to only positive facts. A sequence $\alpha_1, \alpha_2, \ldots,$ of the elements of a model M is said to be an *enumeration* of M if, for any $\alpha \in M$, there exists $i \geq 1$ such that $\alpha = \alpha_i$. The inference

algorithms described in this chapter is assumed to be given any enumeration of M instead of any enumeration of facts about M. Other definitions, e.g. *identifiability* or *consistency* of an inference algorithm etc., are the same as those defined in Chapter 2. Figure 4.1 illustrates the framework of the model inference from positive facts.

Figure 4.1: The framework of the model inference from positive facts



4.2 Minimal Multiple Generalization

In this section, first we review the results on Arimura's minimal multiple generalization [ASO91b, ASO91a]. Then we will give some properties of minimal multiple generalizations for the least Herbrand model of a primitive Prolog. We fix a first order language $\mathcal{L}$ and denote the set of function symbols in $\mathcal{L}$ by Γ throughout this chapter.

Let k be a fixed integer and S be a set of ground words. A *k-multiple generalization* of S is a set of words $\{w_1, w_2, \ldots, w_m\}$ such that $m \leq k$ and $S \subseteq G(w_1) \cup G(w_2) \cup \cdots \cup G(w_m)$.

Definition 4.2 For a fixed integer k and a set S of ground words, a *k-minimal multiple generalization* (k-MMG, for short) of S is a k-multiple generalization $\{w_1, \ldots, w_m\}$ of S such that $G(u_1) \cup \cdots \cup G(u_n) \not\subset G(w_1) \cup \cdots \cup G(w_m)$ for any k-multiple generalization $\{u_1, \ldots, u_n\}$ of S.

Example 4.1 Let S be the set

$$\{reverse([a],[a]), reverse([a,b],[a,b]), reverse([],[]), reverse([c,b],[b,c])\}.$$

Then, the pair $\{reverse(X,X), reverse([X|Y],[Z|W])\}$ of atoms is a 2-multiple generalization of S and the pair $\{reverse([],[]), reverse([X|Y],[Z|W])\}$ is a 2-MMG of S.

Here, we introduce some results from [JLMM88, ASO91b, ASO91a] that are necessary for our discussion. From Theorem 4.2, the minimal multiple generalization defined above can be understood as a natural extension of a least generalization. The property shown in Theorem 4.3 is called *compactness* of unions of tree pattern languages [ASO91a]. The property is very useful for proving not only the following Theorem 4.4 but also several theorems given in this chapter.

Theorem 4.1 (Lassez et al. [JLMM88]) *Suppose that* $|\Gamma| \geq 2$. *Then, for any word* w, *it holds that* $lg(G(w)) \equiv w$.

Theorem 4.2 (Lassez et al. [JLMM88]) *Suppose that* $|\Gamma| \geq 2$. *Then, for any two words* w_1 *and* w_2, *it holds that* $G(w_1) \subseteq G(w_2)$ *if and only if* $w_1 \preceq w_2$.

Theorem 4.3 (Arimura et al. [ASO91b]) *Suppose that* $|\Gamma| \geq k+1$. *Let* $w, u_1, u_2, \ldots, u_k$ *be words. If* $G(w) \subseteq G(u_1) \cup G(u_2) \cup \cdots \cup G(u_k)$, *then* $G(w) \subseteq G(u_i)$ *for some* $1 \leq i \leq k$.

Theorem 4.4 (Arimura et al. [ASO91a]) *Let* k *be any fixed integer and* S *be a set of ground words. If* $|\Gamma| \geq k+1$, *then a* k*-MMG of* S *is computable in polynomial time of* $size(S)$.

Algorithm 4.1: A 2-*mmg* algorithm

Input:A set S of ground words
Output:A set of 2-MMG's of S
Procedure:
    $Ans := \phi$;
    for each ordered pair $\langle w_1, w_2 \rangle$ of elements in S **do**
        Let W be the set of all maximal words consistent with $\langle w_1, w_2 \rangle$;
        for each $U \in W$ **do**
            $V := lg(S - G(U))$;
            if the pair U, V of words are reduced **then**
                $Ans := Ans \cup \{\{V, lg(S - G(V))\}\}$
    if $Ans \neq \phi$ **then** output Ans
    else output $Ans := \{lg(S)\}$

In general, a k-MMG of S is not unique. In fact, for the set S in Example 1.1, the pair $\{reverse(X, X), reverse([c, b], [b, c])\}$ of atoms is also a 2-MMG of S. Arimura's algorithm [ASO91b] to compute a k-MMG of S is not exhaustive for all k-MMG's of S. However, it is ensured that when the algorithm is given a set S of ground words for its input, it finds out at least one k-MMG of S in time polynomial in $size(S)$ and it outputs only k-MMG of S. We denotes the set of all possible outputs of the algorithm for an input S by $k\text{-}mmg(S)$. In what follows, since we apply the k-mmg algorithm to infer the program heads of a primitive Prolog, we fix k to 2. Algorithm 4.1 is a slightly modified version of the original algorithm in [ASO91b] to output $2\text{-}mmg(S)$ directly. It is ensured that the modified algorithm still work in time polynomial in $size(S)$.

Let S be a set of ground words. Two words w_1, w_2 are said to be *reduced with respect to* S if $S \subseteq G(w_1) \cup G(w_2)$ but $S \not\subseteq G(w_i)$ $(i = 1, 2)$. For an ordered pair $\langle w_1, w_2 \rangle$ of words, a word w is said to be *consistent with* $\langle w_1, w_2 \rangle$ if $w_1 \in G(w)$ but $w_2 \notin G(w)$. A *maximal word consistent with* $\langle w_1, w_2 \rangle$ is a maximally general word w consistent with $\langle w_1, w_2 \rangle$, that is, $w \not\prec u$ for any word u consistent with $\langle w_1, w_2 \rangle$. For example, $f(X, a)$ and $f(X, X)$ are maximal words consistent with an ordered pair $\langle f(a, a), f(a, b) \rangle$.

Here we give the properties of $2\text{-}mmg(S)$ for a subset of the least Herbrand model of a primitive Prolog.

Lemma 4.5 *Suppose that $|\Gamma| \geq 3$. Then, for any primitive Prolog $P = \{C_0, C_1\}$, there exists a finite set $S \subseteq M_P$ that satisfies the following two conditions for any S_Δ such that $S \subseteq S_\Delta \subseteq M_P$.*

(1) $\{lg(C_0(M_P)), lg(C_1(M_P))\} \in 2\text{-}mmg(S_\Delta)$.

(2) *For any pair of atoms $\{q_0, q_1\} \in 2\text{-}mmg(S_\Delta)$, either*

  (i) $q_0 \succeq lg(C_0(M_P))$ *and* $q_1 \preceq lg(C_1(M_P))$ *or*

  (ii) $q_1 \succeq lg(C_0(M_P))$ *and* $q_0 \preceq lg(C_1(M_P))$

holds.

To prove the above lemma, we need the following lemma.

Lemma 4.6 *Suppose that $|\Gamma| \geq 2$. Then, for any word w, there exists a finite set $S \subseteq G(w)$ such that, for any words w_0, w_1 and for any S_Δ where $S \subseteq S_\Delta \subseteq G(w)$, if $S_\Delta \subseteq G(w_0) \cup G(w_1)$ then $S_\Delta \subseteq G(w_i)$ for some $i \in \{0,1\}$.*

Proof: As in the same way of defining the set $\Theta_0(p)$ in the proof of Theorem 3 in [ASO91b], we can construct a finite subset $\Theta_0(w)$ of $G(w)$. The finite subset satisfies the condition of S in the lemma. □

Proof of Lemma 4.5: Without loss of generality, we may assume that C_0 is a unit clause, that is, $C_0(M_P) = G(C_0)$.

Proof of (1): From Proposition 3.8, there exist finite sets $S_0 \subseteq C_0(M_P)$ and $S_1 \subseteq C_1(M_P)$ such that $lg(S_i) \equiv lg(C_i(M_P))$ $(i = 0, 1)$. Furthermore, from the assumption that $C_0(M_P) = G(C_0)$, there exists a finite set $S'_0 \subseteq C_0(M_P)$ that satisfies the condition in Lemma 4.6. Let S be the union $S_0 \cup S_1 \cup S'_0$ of the finite sets. By Corollary 3.3, S is ensured to be a finite subset of M_P. For the set S, the following two propositions hold.

(a) $lg(S \cap C_0(M_P)) \equiv lg(C_0(M_P))$ and $lg(S \cap C_1(M_P)) \equiv lg(C_1(M_P))$.

(b) If $S \cap C_0(M_P) \subseteq G(q_0) \cup G(q_1)$ then $S \cap C_0(M_P) \subseteq G(q_i)$ for some $i \in \{0,1\}$.

From the definition of a primitive Prolog, it holds that $C_0(M_P) \cap C_1(M_P) = \phi$. Hence it holds that $S - C_0(M_P) = S \cap C_1(M_P)$ and $S - C_1(M_P) = S \cap C_0(M_P)$.

Let w_0 and w_1 be elements of S such that $w_0 \in C_0(M_P)$ and $w_1 \in C_1(M_P)$. Since each $head(C_i)$ is a generalization of $C_i(M_P)$, it holds that $head(C_i) \succeq lg(C_i(M_P))$ $(i = 0, 1)$. From Theorem 4.2, it holds that $G(head(C_i)) \supseteq G(lg(C_i(M_P)))$ $(i = 0, 1)$. On the other hand, since it follows from the definition of a primitive Prolog that $G(head(C_0)) \cap G(head(C_1)) = \phi$, $G(lg(C_0(M_P))) \cap G(lg(C_1(M_P)))$ is also empty. As a result, $lg(C_1(M_P))$ is consistent with the ordered pair $\langle w_1, w_0 \rangle$. Hence, there exists a maximal word u consistent with $\langle w_1, w_0 \rangle$ such that $u \succeq lg(C_1(M_P))$. Such u appears as U in the inner **for** loop of the algorithm computing $2\text{-}mmg(S)$.

Here, let $v \equiv lg(S - G(u))$. Since $S \supseteq G(u) \cup G(v)$, it holds that $S \cap C_0(M_P) \subseteq G(v) \cup G(u)$. Hence, from the condition (b) on S, either $S \cap C_0(M_P) \subseteq G(v)$ or $S \cap C_0(M_P) \subseteq G(u)$ holds. By the assumption for u, since $w_0 \notin G(u)$, the latter case is impossible. Thus, it follows

from the condition (a) on S and Theorem 4.1 that

$$lg(C_0(M_P)) \equiv lg(S \cap C_0(M_P)) \preceq lg(G(v)) \equiv v.$$

Furthermore, since $C_1(M_P) \subseteq G(u)$ by the assumption for u, it holds that $S - G(u) \subseteq S - C_1(M_P)$. Since $S - C_1(M_P) = S \cap C_0(M_P)$, it holds that $S - G(u) \subseteq S \cap C_0(M_P)$. Hence it follows from Theorem 4.2 and the condition (a) that

$$v \equiv lg(S - G(u)) \preceq lg(S \cap C_0(M_P)) \equiv lg(C_0(M_P)).$$

Thus $v \equiv lg(C_0(M_P))$ holds.

Since $G(lg(C_0(M_P))) \subseteq G(head(C_0))$ and $G(head(C_0)) \cap C_1(M_P) = \phi$, it holds that $G(v) \cap C_1(M_P) = \phi$ and $G(v) \not\supseteq S$. On the other hand, since $w_0 \notin G(u)$, it also holds that $G(u) \not\supseteq S$. Since $S \subseteq G(u) \cup G(v)$, two words u and v are reduced with respect to S. Hence, the algorithm adds the pair $\{v, lg(S - G(v))\}$ to Ans.

From the first assumption that $C_0(M_P) = G(C_0)$ and Theorem 4.1, $lg(C_0(M_P)) \equiv lg(G(C_0)) \equiv C_0$ holds. Hence, it holds that

$$lg(S - G(v)) \equiv lg(S - G(lg(C_0(M_P)))) \equiv lg(S - G(C_0)) \equiv lg(S - C_0(M_P)).$$

Since $S - C_0(M_P) = S \cap C_1(M_P)$, it holds that

$$lg(S - G(v)) \equiv lg(S - C_0(M_P)) \equiv lg(S \cap C_1(M_P)) \equiv lg(C_1(M_P)).$$

As a result, we have $\{lg(C_0(M_P)), lg(C_1(M_P))\} = \{v, lg(S - G(v))\} \in 2\text{-}mmg(S)$.

Proof of (2): For any $\{q_0, q_1\} \in 2\text{-}mmg(S)$, since $S \subseteq G(q_0) \cup G(q_1)$, it holds that $S \cap C_0(M_P) \subseteq G(q_0) \cup G(q_1)$. Hence, from the condition (b), either $S \cap C_0(M_P) \subseteq G(q_0)$ or $S \cap C_0(M_P) \subseteq G(q_1)$ holds. It is sufficient to show the case (i) by assuming $S \cap C_0(M_P) \subseteq G(q_0)$.

From the condition (a) on S, Theorem 4.2, and Theorem 4.1, it holds that

$$lg(C_0(M_P)) \equiv lg(S \cap C_0(M_P)) \preceq lg(G(q_0)) \equiv q_0.$$

On the other hand, from the way of calculation in the algorithm, $q_1 \equiv lg(S - G(q_0))$ holds. Since $S \cap C_1(M_P) = S - C_0(M_P) \supseteq S - G(q_0)$, it holds that

$$lg(C_1(M_P)) \equiv lg(S \cap C_1(M_P)) \succeq lg(S - G(q_0)) \equiv q_1.$$

This completes the proof of Lemma 4.5. □

Lemma 4.5 ensures that, for any given enumeration of positive facts in the target model, the pair of atoms $lg(C_0(M_P))$, $lg(C_1(M_P))$ can be found in the limit using the 2-*mmg* algorithm. As described in the previous chapter, the pair $lg(C_0(M_P))$, $lg(C_1(M_P))$ are the heads of $dmplg(P)$ such that $M(P) = M(dmplg(P))$. Hence, the lemma suggests applicability of the 2-*mmg* algorithm to infer heads of clauses of the target program.

4.3 DMPLG's of Primitive Prologs

In this section, we show the properties of a $dmplg(P)$ for a primitive Prolog P. The target properties we would like to show will appear in Lemma 4.14 and Lemma 4.15. They ensure that, for the given heads $lg(C_0(M_P))$ and $lg(C_1(M_P))$ of the $dmplg(P)$, any hypothesis H obtained by adding any inappropriate body to the heads has a positive counter-example, that is, there exists a positive fact $\alpha \in M(P)$ such that $\alpha \notin M(H)$. An appropriate body means just the body of the $dmplg(P)$. As a result, if the heads of a $dmplg(P)$ are known then the $dmplg(P)$ can be reconstructed from only positive information. Furthermore, Lemma 4.10 gives a basic strategy for searching for the body of a $dmplg(P)$. By the lemma, each atom appearing in the body of the $dmplg(P)$ is ensured to be a variant of a least generalization of the heads of the $dmplg(P)$. With Lemma 4.5, these lemmata complete preparation for constructing an inference algorithm from positive facts.

Let $P = \{C_0, C_1\}$ be a primitive Prolog, where

$$C_0 = p(s),$$
$$C_1 = p(t[X_1, \ldots, X_m]) \leftarrow p(X_1), \ldots, p(X_m).$$

Lemma 4.7 *Suppose that* $|\Gamma| \geq 2$. *Then,* $dmplg(P)$ *is of the form* $\{C_0, C_1\theta\}$ *where* $\theta = \{X_1/r_1, \ldots, X_m/r_m\}$.

Proof: Since C_0 is a unit clause, $C_0(M_P) = G(C_0)$. Hence, it follows from Theorem 4.1 that $lg(C_0(M_P)) \equiv lg(G(C_0)) \equiv C_0$.

Let V_{head} be the set of variables in C_1 appearing only in $head(C_1)$, that is, $V_{head} = var(C_1) - \{X_1, \ldots, X_m\}$. If $V_{head} = \phi$ then there is nothing to be proved. Thus we assume

that $V_{head} \neq \phi$. Let σ be a substitution such that $lg(C_1(M_P)) = head(C_1)\sigma$ and γ be a ground substitution such that $dom(\gamma) \subseteq var(C_1)$ and $p(X_i)\gamma \in M(P)$ for any $1 \leq i \leq m$. Note that $head(C_1)\gamma \in C_1(M_P)$ holds.

Suppose that, for some $Y \in V_{head}$, σ contains a binding Y/t for a non-variable term t. From the assumption that $|\Gamma| \geq 2$, there exists a ground term α whose principal functor differs from that of t. Let Y/β be the binding in γ and γ' be the ground substitution obtained from γ by replacing the binding Y/β by Y/α. Then, since $p(X_i)\gamma' = p(X_i)\gamma \in M(P)$ for any $1 \leq i \leq m$, it holds that $head(C_1)\gamma' \in C_1(M_P)$. On the other hand, $head(C_1)\gamma$ is not an instance of $head(C_1)\sigma$, because the principal functor of t differs from that of α. Hence $head(C_1)\gamma' \notin C_1\sigma(M_P)$. This contradicts Lemma 3.5.

Now, suppose that, for some $Y \in V_{head}$, σ contains a binding Y/X for some variable $X \in C_1$. Let X/α_1 and Y/α_2 be the bindings in γ. If $\alpha_1 \neq \alpha_2$ then it immediately follows that $head(C_1)\gamma$ is not an instance of $head(C_1)\sigma$. Otherwise, there exists a ground term $\beta \neq \alpha_1 = \alpha_2$ from the assumption that $|\Gamma| \geq 2$. Let γ' be the substitution obtained from γ by replacing the binding Y/α_2 by Y/β. Then it holds that $head(C_1)\gamma'$ is in $C_1(M_P)$ but not an instance of $head(C_1)\sigma$. Consequently, both cases contradict Lemma 3.5.

A similar argument leads the same contradiction for the case in which σ contains bindings Y_1/X and Y_2/X for $Y_1, Y_2 \in V_{head}$ $(Y_1 \neq Y_2)$ and any variable X. Thus, if σ contains a binding Y/t for some $Y \in V_{head}$, then t must be a variable which does not occur in C_1 and occurs in σ at most once, that is, Y/t is just renaming of variable Y. Hence, $C_1\sigma \equiv C_1\theta$. This completes the proof of the proposition. □

Let $dmplg(P) = \{C_0, C_1\theta\}$ where $\theta = \{X_1/r_1, \ldots, X_m/r_m\}$ and

$$C_0 = p(s),$$
$$C_1\theta = p(t[r_1, \ldots, r_m]) \leftarrow p(r_1), \ldots, p(r_m).$$

Lemma 4.8 *$lg(\{s, t[r_1, \ldots, r_m]\}) \succ s$ and $lg(\{s, t[r_1, \ldots, r_m]\}) \succ t[r_1, \ldots, r_m]$.*

Proof: If $lg(\{s, t[r_1, \ldots, r_m]\}) \equiv s$, then $s \succeq t[r_1, \ldots, r_m]$. This contradicts the condition of a primitive Prolog: $G(s) \cap G(t[X_1, \ldots, X_m]) = \phi$. A similar argument for the alternate claim completes the proof of the lemma. □

Lemma 4.9 *For any* $1 \leq i \leq m$, $p(r_i) \equiv lg(M_P)$.

Proof: Assume that, for some $1 \leq i \leq m$, $p(r_i) \not\succeq lg(M_P)$. Then, there exists $p(\beta) \in M_P$ such that $p(r_i) \not\succeq p(\beta)$. On the other hand, there exists a substitution γ such that $p(t[\beta, \ldots, \beta])\gamma \in C_1(M_P)$. Since $p(r_i) \not\succeq p(\beta)$ for some $1 \leq i \leq m$, it holds that $head(C_1\theta) = p(t[r_1, \ldots, r_m]) \not\succeq p(t[\beta, \ldots, \beta])\gamma$. Hence, $p(t[\beta, \ldots, \beta])\gamma \notin C_1\theta(M_P)$. This contradicts the fact that $C_1\theta(M_P) = C_1(M_P)$. Thus $p(r_i) \succeq lg(M_P)$ for any $1 \leq i \leq m$.

Now, assume that $p(r_i) \succ lg(M_P)$ for some $1 \leq i \leq m$. Let $p(\tau) = lg(M_P)$ and $\sigma = \{X_i/\tau\}$. For any $\alpha \in C_1(M_P)$, there exists a ground substitution η such that $\alpha = head(C_1)\eta$ and $p(X_j)\eta \in M_P$ for any $1 \leq j \leq m$. Let X_i/β be the binding in η. Note that $p(\beta) \in M_P$. Since $p(\tau) = lg(M_P)$, there exists a substitution σ' such that $p(\tau)\sigma' = p(\beta)$. Hence there exists a substitution η' such that $\sigma\sigma'\eta' = \eta$. Such η' can be obtained from η by deleting the binding X_i/β. Thus it holds that $\alpha = head(C_1)\sigma\sigma'\eta'$. Since, for each $\alpha \in C_1(M_P)$, there exist substitutions σ' and η' such that $\alpha = head(C_1)\sigma\sigma'\eta'$, $head(C_1)\sigma$ is a generalization of $C_1(M_P)$. On the other hand, it follows from $r_i \succ \tau$ that

$$p(t[X_1, \ldots, X_{i-1}, \tau, X_{i+1}, \ldots, X_m]) \not\succeq p(t[r_1, \ldots, r_i, \ldots, r_m]),$$

that is, $head(C_1)\sigma \not\succeq head(C_1)\theta$. Since $head(C_1)\sigma$ is a generalization of $C_1(M_P)$, this contradicts that $head(C_1)\theta$ is a *least* generalization of $C_1(M_P)$.

Thus the lemma has been proved. □

The following Lemma 4.10 ensures that, for the pair $\{p(s), p(t')\}$ of heads of clauses in $dmplg(P)$, each atom appearing in the body of a clause in $dmplg(P)$ is of the form $lg(\{p(s), p(t')\})$. A greedy search algorithm for the body of a clause given in the next section is based on this property.

Lemma 4.10 *For any* $1 \leq i \leq m$, $p(r_i) \equiv lg(\{p(s), p(t[r_1, \ldots, r_m])\})$.

Proof: From our definition,

$$lg(\{p(s), p(t[r_1, \ldots, r_m])\}) \equiv lg(\{lg(C_0(M_P)), lg(C_1(M_P))\}).$$

Since $C_0(M_P) \cup C_1(M_P) = M_P$, from Lemma 4 in [JLMM88], it follows that

$$lg(\{lg(C_0(M_P)), lg(C_1(M_P))\}) \equiv lg(C_0(M_P) \cup C_1(M_P)) \equiv lg(M_P).$$

Thus, from Lemma 4.9, it holds that $lg(\{p(s), p(t[r_1, \ldots, r_m])\}) \equiv p(r_i)$. □

The following propositions and lemmata are useful in proving validity of the greedy search for the body of a clause. Since the algorithm is required to find an appropriate body from only positive data, it has to avoid constructing an overgeneralized hypothesis. Proposition 4.13 says that a hypothesis with an appropriate body that corresponds to a subset of the body of the $dmplg(P)$ is consistent with all positive examples. On the other hand, Lemma 4.14 and Lemma 4.15 ensure that, for any hypothesis with an inappropriate body, there exists a positive counter example, even if the body consists of only one atom.

Proposition 4.11 *For any* $1 \leq i \neq j \leq m$, $var(r_i) \cap var(r_j) = \phi$.

Proof: Assume that $var(r_i) \cap var(r_j) \neq \phi$ for some $i \neq j$. Then, there exist indexes I and J such that $p(r_i)(I) = p(r_j)(J) = X$, where X is a variable. Since $p(r_i) \equiv lg(M_P)$, for any $p(\alpha) \in M_P$, $p(\alpha)(I)$ is well-defined as a ground term.

Let a and b be different ground terms. Suppose that there exist atoms $p(\alpha)$ and $p(\beta)$ in M_P such that $p(\alpha)(I) = a$ and $p(\beta)(J) = b$. Since the variables $X_1, \ldots, X_m$ in the clause $C_1 = p(t[X_1, \ldots, X_m]) \leftarrow p(X_1), \ldots, p(X_m)$ differs from each other, for $p(t') = p(t[X_1, \ldots, X_m])\{X_i/\alpha, X_j/\beta\}$, there exists a ground substitution γ such that $dom(\gamma) \subseteq var(p(t'))$ and $p(t')\gamma \in C_1(M_P)$. For $C_1\theta$, since $p(r_i)(I) = p(r_j)(J) = X$, there is no substitution σ such that $p(r_i)\sigma = p(\alpha)$ and $p(r_j)\sigma = p(\beta)$. This means that $p(t')\gamma \notin C_1\theta(M_P)$ and contradicts the fact that $dmplg(P) = \{C_0, C_1\theta\}$. Hence, such α, β do not exist, that is, $p(\alpha)(I) = p(\beta)(J)$ for any $p(\alpha), p(\beta) \in M_P$.

Here, suppose again that there exist $p(\alpha), p(\alpha') \in M_P$ such that $p(\alpha)(I) \neq p(\alpha')(I)$. Then there exists $p(\beta) \in M_P$ such that $p(\beta)(J) = p(\alpha)(I) \neq p(\alpha')(I)$. This contradicts the above consequence. Thus there exists a ground term a such that $p(\alpha)(I) = a$ for any $p(\alpha) \in M_P$. Hence $p(r_i) \equiv lg(M_P)(I) = a$ for some ground term a. This contradicts the first assumption that $p(r_i)(I)$ is a variable. □

A similar discussion as above leads the next proposition.

Proposition 4.12 *For any* $1 \leq i \neq j \leq m$,

$$var(r_i) \cap (var(t[X_1, \ldots, X_m]) - \{X_1, \ldots, X_m\}) = \phi.$$

Proposition 4.13 *Let* $P' = \{C_0, C_1'\}$ *where*

$$C_1' = p(t[r_1, \ldots, r_m]) \leftarrow p(r_{i_1}), \ldots, p(r_{i_k}) \text{ and } \{r_{i_1}, \ldots, r_{i_k}\} \subseteq \{r_1, \ldots, r_m\}.$$

Then it holds that $M(P) \subseteq M(P')$.

Proof: It is clear from the definition of C_1' that $C_1\theta(M) \subseteq C_1'(M)$ for any set M of ground atoms. Hence, we can show that $T_{dmplg(P)} \uparrow \omega \subseteq T_{P'} \uparrow \omega$ along the same line of argument as in the proof of Theorem 3.4. Thus, it holds that $M(dmplg(P)) = M(P) \subseteq M(P')$. □

Lemma 4.14 *Suppose that* $|\Gamma| \geq 3$. *Let* $P' = \{C_0, C_1'\}$ *where* $C_1' = p(t[r_1, \ldots, r_m]) \leftarrow p(r')$ *and* r' *is a sub-term of* $t[r_1, \ldots, r_m]$ *such that* $r' \equiv r_i$ $(1 \leq i \leq m)$ *and* $r' \notin \{r_1, \ldots, r_m\}$. *Then it holds that* $M(P) - M(P') \neq \phi$.

Proof: From Proposition 4.11, Proposition 4.12, and the condition on r', it follows that $var(r') \cap var(r_i) = \phi$ $(1 \leq i \leq m)$. Hence, for any ground substitution σ such that $dom(\sigma) \subseteq var(r')$, there exists a ground substitution γ such that $dom(\gamma) \subseteq p(t[r_1, \ldots, r_m])\sigma$ and $p(t[r_1, \ldots, r_m])\sigma\gamma \in M(P)$.

For proving the lemma, it is sufficient to show that there exists a ground substitution σ such that $dom(\sigma) \subseteq var(r')$ and

$$t[r_1, \ldots, r_m] \not\succeq r'\sigma \quad \text{and} \quad s \not\succeq r'\sigma. \tag{4.1}$$

Since it follows that $p(r')\sigma \notin M(P')$ from (4.1), $p(t[r_1, \ldots, r_m])\sigma\gamma \notin M(P')$ for any substitution γ. With the above fact, this implies an existence of a substitution γ such that

$$p(t[r_1, \ldots, r_m])\sigma\gamma \in M(P) \quad \text{but} \quad p(t[r_1, \ldots, r_m])\sigma\gamma \notin M(P').$$

Since $r' \equiv r_i$, from Lemma 4.8, it holds that $r' \succ s$ and $r' \succ t[r_1, \ldots, r_m]$. In what follows, $t[r_1, \ldots, r_m]$ is abbreviated as t.

Since $r' \succ s$, the following two cases are possible:

(1) There is an index I such that $r'(I) = X$ and $s(I) = u$ where X is a variable and u is a non-variable term.

(2) There are indexes I_1 and I_2 such that $r'(I_1) = X$, $r'(I_2) = Y$, $s(I_1) = Z$ and $s(I_2) = Z$ where X, Y, Z are mutually distinct variables.

Furthermore, since $r' \equiv lg(s,t)$, these cases can be divided into the following four cases:

(1-1) There exists an index I such that $r'(I) = X$, $s(I) = u$ and $t(I) = v$ where X is a variable and u, v are non-variable terms with different principle functors.

(1-2) There exists an index I such that $r'(I) = X$, $s(I) = u$ and $t(I) = Y$ where X, Y are variables and u is a non-variable term.

(2-1) There exist indexes I_1 and I_2 such that $r'(I_1) = X$, $r'(I_2) = Y$, $s(I_1) = Z$, $s(I_2) = Z$, $t(I_1) = u$ and $t(I_2) = v$ where X, Y, Z are mutually distinct variables and u, v are mutually distinct terms such that at least one of them is not variable.

(2-2) There exist indexes I_1 and I_2 such that $r'(I_1) = X$, $r'(I_2) = Y$, $s(I_1) = Z$, $s(I_2) = Z$, $t(I_1) = X_1$ and $t(I_2) = Y_1$ where X, X_1, Y, Y_1, Z are mutually distinct variables.

Case (1-1): From the assumption that $|\Gamma| \geq 3$, there exists a ground term w whose principle functor differs from neither u's nor v's. For any ground substitution σ that binds X with w, (4.1) holds.

Case (1-2): Since $r' \succ t$, there are two possible cases:

(1') There exists an index I' such that $r'(I') = X'$ and $t(I') = u'$ where X' is a variable and u' is a non-variable term.

(2') There exist indexes I'_1 and I'_2 such that $r'(I'_1) = X'$, $r'(I'_2) = Y'$, $t(I'_1) = Z'$ and $t(I'_2) = Z'$ where X', Y', Z' are mutually distinct variables.

In the case (1'), it holds that $X \neq X'$[1]. Let σ be a ground substitution that binds X with a ground term w whose principle functor differs from that of u and X' with a ground term w' whose principle functor differs from that of u'. Then σ satisfies (4.1). In the case (2'), let w_1 be a ground term whose principle functor differs from u's and w_2 be a different ground term from w_1. Then (4.1) holds for any ground substitution σ that binds X with w_1, X' with one of which w_1 or w_2 and Y' with the other[2].

Case (2-1): Without loss of generality, we may assume that u is not a variable. Let w_1 be a ground term whose principle functor differs from u's and w_2 be a different ground term

[1] The term $t(I)$ is a variable and the term $t(I')$ is a non-variable. Thus, if $X = X'$, then $r' \not\succ t$.

[2] When X, X', Y' differs from each other, this choice of bindings can be arbitrary. When X is the same as X', bind Y' with w_2. When X is the same as Y', bind X' with w_2.

from w_1. Then (4.1) holds for any ground substitution σ that binds X with w_1 and Y with w_2.

Case (2-2): Similarly for the case (1-2), there are two possible cases (1') and (2'). The case (1') for the case (2-2) is the same as the case (2') for the case (1-2). In the case (2'), let w and w' be different ground terms. Since $X \neq Y$ and $X' \neq Y'$, it suffices to consider a ground substitution that binds X, X' with one of which w_1 or w_2 and Y, Y' with the other. $\square$

Lemma 4.15 *Let $P = \{C_0', head(C_1\theta)\}$ where $C_0' = p(s) \leftarrow p(r')$ and r' is a sub-term of s such that $r' \equiv r_i$ $(1 \leq i \leq m)$. Then it holds that $M(P) - M(P') \neq \phi$.*

Proof: Since $G(p(s)) \subseteq M(P)$ and r' is a sub-term of s, for any ground substitution σ such that $dom(\sigma) \subseteq var(r')$, there exists a ground substitution γ such that $dom(\gamma) \subseteq var(p(s)\sigma)$ and $p(s)\sigma\gamma \in M(P)$. Hence, the lemma can be proven along the same line of argument as in the proof of the previous lemma. $\square$

4.4 A Greedy Search Algorithm for the Body

In this section, we describe an algorithm that, for a set S of ground atoms and an ordered pair $\langle p(s), p(t)\rangle$ of atoms such that $S \subseteq G(p(s)) \cup G(p(t))$ and $G(p(s)) \cap G(p(t)) = \phi$, searches for a program $P = \{C_0, C_1\}$ where

$$C_0 = p(s),$$
$$C_1 = p(t) \leftarrow p(u_1), \ldots, p(u_k),$$

and P satisfies the following conditions:

1. $S \subseteq M(P)$:
2. Each u_i is a proper sub-term of t such that $u_i \equiv lg(t, s)$ $(1 \leq i \leq k)$ and $var(u_i) \cap var(u_j) = \phi$ $(i \neq j)$.
3. Let $t'[X_1, \ldots, X_k]$ be the term that obtained from t by replacing each occurrence of u_i by X_i $(1 \leq i \leq k)$ where X_i $(1 \leq i \leq k)$ is a variable that does not appear in t and $X_i \neq X_j$ $(i \neq j)$. Then it holds that $G(t'[X_1, \ldots, X_k]) \cap G(s) = \phi$.

4. The body of C_1 is maximal one in such bodies satisfying the above conditions. That is, for any other sub-term u_{k+1} of t that satisfies the condition 2 and for the clause $C = p(t) \leftarrow p(u_1), \ldots, p(u_k), p(u_{k+1})$, C does not satisfy the condition 3 or $S \not\subseteq M(\{C_0, C\})$.

We denote a program that satisfies the above conditions by $P(S, \langle p(s), p(t) \rangle)$. While, in general, there may exist several $P(S, \langle p(s), p(t) \rangle)$'s, one of them can be found in polynomial time by simple greedy search.

Algorithm 4.2: A greedy search algorithm for $P(S, \langle p(s), p(t) \rangle)$

Input: A set of positive examples S and a tuple of atoms $\langle p(s), p(t) \rangle$ such that
$S \subseteq G(p(s)) \cup G(p(t))$ and $G(p(s)) \cap G(p(t)) = \phi$.
Output: $P(S, \langle p(s), p(t) \rangle)$
Procedure:
Let $SUB(t)$ be the set of proper sub-terms of t, that are variant of $lg(\{s, t\})$,
$Body := \phi$; $Pat := t$;
$H := \{p(t) \leftarrow Body., p(s).\}$;
for each $T \in SUB(t)$ s.t. $var(T) \cap var(Body) = \phi$ **do**
  $H := \{p(t) \leftarrow Body \cup \{p(T)\}., p(s).\}$;
  if $S \not\subseteq M(H)$ **then**
    $H := \{p(t) \leftarrow Body., p(s).\}$;
  else Let Pat' be the term which is obtained from Pat by replacing
    the all occurrences of T in Pat with a new variable X;
    if $G(Pat') \cap G(s) \neq \phi$ **then** $H := \{p(t) \leftarrow Body., p(s).\}$;
    else $Body := Body \cup \{p(T)\}$;
      $Pat := Pat'$;
Output H and halt;

Lemma 4.16 *For a set S of ground atoms and an ordered pair $\langle p(s), p(t) \rangle$ of atoms such that $S \subseteq G(p(s)) \cup G(p(t))$ and $G(p(s)) \cap G(p(t)) = \phi$, Algorithm 4.2 outputs a $P(S, \langle p(s), p(t) \rangle)$ in time polynomial in $size(S \cup \{p(s), p(t)\})$.*

Proof: A least generalization $lg(t, s)$ is computable in time polynomial in $size(\{t, s\})$. $SUB(t)$ is computable in time polynomial in $size(t)$ and $|SUB(t)| \leq size(t)$. The number of execution of the body of the **for** loop is at most $SUB(t) \leq size(t)$ and the examination

whether $var(T) \cap var(Body) = \phi$ or not is done in time polynomial in $size(t) \times |SUB(t)| \leq size(t)^2$. The main operations performed in each execution of the **for** loop are testing two **if** statements. Since each T_i in a recursive clause $p(t) \leftarrow p(T_1), \ldots, p(T_j)$ of H is a proper sub-term of t, testing whether $S \subseteq M(H)$ or not can be done in time polynomial in $size(S)$. For testing the second **if** statement, since it suffices to verify the unifiability of Pat' with s, it takes at most time polynomial in $size(\{s,t\})$. Hence, Algorithm 4.2 terminates in time polynomial in $size(S \cup \{p(s), p(t)\})$.

It is clear that the output of the algorithm is a $P(S, \langle p(s), p(t) \rangle)$. □

Let $P(S, \langle p(s), p(t) \rangle) = \{C_0, C_1\}$ be an output of Algorithm 4.2 where $C_0 = p(s)$ and $C_1 = p(t) \leftarrow p(u_1), \ldots, p(u_m)$. Let C_1' be the clause which obtained from C_1 by replacing each occurrence of u_i by X_i $(1 \leq i \leq m)$ where $X_1, \ldots, X_m$ are mutually distinct variables that do not occur in C_1. Since no u_i contains other u_j as its sub-term, C_1' is well-defined and obtained from C_1 in time polynomial in $size(t)$. We denotes $\{C_0, C_1'\}$ by $pr(P(S, \langle p(s), p(t) \rangle))$.

Lemma 4.17 *The program $pr(P(S, \langle p(s), p(t) \rangle))$ is a primitive Prolog which has the same least Herbrand model as that of $P(S, \langle p(s), p(t) \rangle)$.*

Proof: We abbreviate $P(S, \langle p(s), p(t) \rangle)$ as P. From the way of constructing P, it is clear that $pr(P)$ is a primitive Prolog. Thus, we show that $M(P) = M(pr(P))$.

Since C_1 is an instance of C_1', clearly, it holds that $M(P) \subseteq M(pr(P))$.

In order to show the converse, we show that $T_{pr(P)} \uparrow n \subseteq T_P \uparrow n$ for any integer n by induction on n.

For $n = 1$, clearly $T_{pr(P)} \uparrow 0 = G(p(s)) = T_P \uparrow 0$.

Suppose that $T_{pr(P)} \uparrow n \subseteq T_P \uparrow n$ for some n (≥ 2). Let $p(t')$ be the head of C_1'. For any $\alpha \in T_{pr(P)} \uparrow (n+1)$, there exists a substitution σ such that

$$\alpha = p(t')\sigma \text{ and } p(X_i)\sigma \in T_{pr(P)} \uparrow n \ (1 \leq i \leq m).$$

Here we can divide the substitution σ into two substitutions δ and γ such that $\sigma = \gamma\delta$ where δ operates only variables in $\{X_1, \ldots, X_m\}$ and γ operates only variables in $var(t') - \{X_1, \ldots, X_m\}$. From the induction hypothesis, $p(X_i)\sigma \in T_P \uparrow n$ $(1 \leq i \leq m)$. Since $p(u_i) \equiv lg(\{p(s), p(t)\})$, for any $p(X_i)\sigma \in T_P \uparrow n$, there exists a ground substitution δ_i such that

$p(X_i)\sigma = p(u_i)\delta_i$. Since $\{X_1, \dots, X_m\} \cap var(u_i) = var(u_i) \cap var(u_j) = \phi$ $(1 \le i \ne j \le m)$, for $\delta' = \delta_1 \cdots \delta_m$, it holds that $\alpha = p(t)\gamma\delta'$ and $p(u_i)\gamma\delta' \in T_P \uparrow n$ $(1 \le i \le m)$. Thus it holds that $\alpha \in T_P \uparrow (n+1)$. □

4.5 Polynomial Update Time Inferability from Positive Facts

In what follows, let $P = \{C_0, C_1\}$ be any primitive Prolog where

$$C_0 = p(s),$$
$$C_1 = p(t[X_1, \dots, X_m]) \leftarrow p(X_1), \dots, p(X_m),$$

and $dmplg(P) = \{C_0, C_1\theta\}$, where $\theta = \{X_1/r_1, \dots, X_m/r_m\}$. For notational convenience, we abbreviate $t[X_1, \dots, X_m]$ as t and $t[r_1, \dots, r_m]$ as t_r. Let $e_1, e_2, \dots$ be any enumeration of $M(P)$ and S_i be a set $\{e_1, \dots, e_i\}$ of first i elements in the enumeration.

Theorem 4.18 *There exists an integer N such that, for any integer $n \ge N$, Algorithm 4.2 outputs $dmplg(P)$ for given S_n and $\langle p(s), p(t_r)\rangle$ as its inputs.*

Proof: Let $T_1, \dots, T_k$ be all sub-terms of t_r such that $T_i \notin \{r_1, \dots, r_m\}$ and $p(T_i) \equiv lg(\{p(s), p(t_r)\})$. From Lemma 4.14, for any T_i $(1 \le i \le k)$, a program $P_{T_i} = \{C_0, p(t_r) \leftarrow p(T_i)\}$ has a positive counter example $c_i \in M(P)$ such that $c_i \notin M(P_{T_i})$. Thus, for any input S_n such that $\{c_1, \dots, c_k\} \subseteq S_n$, Algorithm 1 outputs a program $P = \{C_0, C_1'\}$ where $C_1' = p(t_r) \leftarrow p(r_{i_1}), \dots, p(r_{i_j})$ and $\{r_{i_1}, \dots, r_{i_j}\} \subseteq \{r_1, \dots, r_m\}$. On the other hand, from Proposition 4.13 and maximality of the body of the clause in $P(S, \langle p(s), p(t_r)\rangle)$ (the condition 4 in the definition of $P(S, \langle p(s), p(t_r)\rangle)$), there is no case in which $\{r_{i_1}, \dots, r_{i_j}\}$ is a proper subset of $\{r_1, \dots, r_m\}$. □

By Theorem 4.18, if we assume that a unit clause $C_0 = p(s)$ is given to an inference algorithm as a hint, we can easily construct a consistent and conservative polynomial update time inference algorithm that identifies the class of models of primitive Prologs in the limit.

Algorithm 4.3: A consistent and conservative polynomial update time inference algorithm with a hint

Given: The unit clause $p(s)$ of a target program P.
Input: An enumeration of $M(P)$: $e_1, e_2, \ldots$.
Output: A sequence of primitive Prologs: $P_1, P_2, \ldots$.
Procedure:

$S := \{\}$; $H := \{p(s)\}$;
Read the next example e; $S := S \cup \{e\}$;
if $e \in M(H)$ **then** output H;
else $S_- := S - G(p(s))$;
Compute $P(S, \langle p(s), lg(S_-)\rangle)$ using Algorithm 1;
$H := pr(P(S, \langle p(s), lg(S_-)\rangle))$; output H;

Theorem 4.19 *Suppose that $|\Gamma| \geq 2$. Then, for any primitive Prolog P, Algorithm 4.3 identifies $M(P)$ in the limit. Furthermore, Algorithm 4.3 is a consistent and conservative polynomial update time inference algorithm.*

Proof: Since $S_- = S - G(p(s)) \subseteq C_1(M_P)$ for any $S \subseteq M(P)$, it holds that $lg(S_-) \preceq lg(C_1(M_P)) \equiv p(t_r)$. From Proposition 3.8, there exists a finite subset S^* of $C_1(M_P)$ such that $lg(S^*) \equiv p(t_r)$. Hence, for any integer n such that $S^* \subseteq S_n$, it holds that $lg(S_-) = p(t_r)$. With Lemma 4.17 and Theorem 4.18, this proves that the algorithm identifies $M(P)$ in the limit.

The initial hypothesis $\{p(s)\}$ is a primitive Prolog. From the facts that $M_P = C_0(M_P) \cup C_1(M_P) = G(p(s)) \cup C_1(M_P)$, $S \subseteq M_P$, and $G(p(s)) \cap G(p(t)) = \phi$, it holds that $S_- = S - G(p(s)) \subseteq C_1(M_P) \subseteq G(p(t))$. Hence, it holds that $G(p(s)) \cap G(lg(S_-)) = \phi$. From Lemma 4.17, every hypothesis H is also primitive Prolog. Hence, it can be decided in time polynomial in $size(e)$ whether $e \in M(H)$ or not. S_- and $lg(S_-)$ is computable in time polynomial in $size(S)$. Since $p(s)$ is a unit clause, for any $\alpha \in M(P)$, it holds that $size(p(s)) \leq size(\alpha)$, that is, $size(p(s)) \leq size(S)$. Also it holds that $size(lg(S_-)) \leq size(S)$. Thus, from Lemma 4.16 and Lemma 4.17, $P(S, \langle p(s), lg(S_-)\rangle)$ and $pr(P(S, \langle p(s), lg(S_-)\rangle))$ is also computable in time polynomial in $size(S)$. Hence Algorithm 3 is a polynomial time inference algorithm.

From Lemma 4.16 and Lemma 4.17, it is clear that the algorithm is consistent. Since the algorithm does not change H when $e \in M(H)$, conservativity of the algorithm is also clear.

□

From the above theorem, if the inference algorithm can find a unit clause in some way, the algorithm identifies a target model efficiently. In general, there is no way to find a unit clause from arbitrary enumeration of a model in bounded finite time. However, there is a way to identify it in the limit (unbounded finite time) using the 2-mmg algorithm. Finally, we describe a polynomial update time inference algorithm that is consistent but not conservative.

Let $P = \{C_0, C_1\}$ be a primitive Prolog where C_0 is a unit clause. If C_1 has nonempty body, then it holds that $min\{size(\alpha) \mid \alpha \in C_0(M_P)\} < min\{size(\beta) \mid \beta \in C_1(M_P)\}$. On the other hand, from Lemma 4.5, for any $\{q_0, q_1\} \in$ 2-$mmg(S)$, either $q_0 \succeq C_0$ or $q_1 \succeq C_0$ holds in the limit. In both case, C_0 is less general. Thus, to identify a unit clause C_0, an inference algorithm have only to keep a minimal size example given so far and find a less general atom, that coves the example, in atoms contained outputs produced by the 2-mmg algorithm.

Algorithm 4.4: A consistent polynomial update time inference algorithm using the 2-mmg algorithm

Input: An enumeration of $M(P)$: $e_1, e_2, \ldots$.
Output: A sequence of primitive Prologs: $P_1, P_2, \ldots$.
Procedure:

$S := \{\}$; $Size := +\infty$;
Read the next example e; $S := S \cup \{e\}$;
if $size(e) < Size$ **then**
  $Min := e$; $Size := size(e)$
Let H be the set of all $\{q_0, q_1\} \in$ 2-$mmg(S)$ such that $G(q_0) \cap G(q_1) = \phi$;
if there exists an atom h'_0 appearing in H such that
$Min \in G(h'_0)$ and $h'_0 \not\succ q$ for any q appearing in H **then**
  Compute $P(S, \langle h'_0, h'_1 \rangle)$ using Algorithm 4.2 where $\{h'_0, h'_1\} \in H$.
  Output $pr(P(S, \langle h'_0, h'_1 \rangle))$
else output $lg(S)$

Theorem 4.20 *Suppose that $|\Gamma| \geq 3$. Then, for any primitive Prolog P, Algorithm 4.4 identifies $M(P)$ in the limit. Furthermore, Algorithm 4.4 is a consistent polynomial update time inference algorithm.*

Proof: If a target program consists of only unit clauses, then the problem is just the same as one for inferring unions of tree pattern languages [ASO91b] and it is easily shown that the sequence of outputs produced by Algorithm 4.4 converges to a primitive Prolog H such that $M(H) = M(P)$.

Thus, we assume that a target program consists of a unit clause C_0 and a non-unit clause C_1. From (1) in Lemma 4.5, there exists a finite subset S of M_P such that, for any $S \subseteq S_\Delta \subseteq M_P$, $\{lg(C_0(M_P)), lg(C_1(M_P))\} \in 2\text{-}mmg(S_\Delta)$. Furthermore, $lg(C_0(M_P)) = C_0$ and $lg(C_1(M_P))$ is an instance of $head(C_1)$. Since $G(C_0) \cap G(head(C_1))$ is empty, $G(lg(C_0(M_P))) \cap G(lg(C_1(M_P)))$ is also empty. Thus, after the algorithm has fed all elements in S, the pair of atoms $\{lg(C_0(M_P)), lg(C_1(M_P))\}$ is always contained in H.

On the other hand, from (2) in Lemma 4.5, for any atom q appearing in $2\text{-}mmg(S_\Delta)$, either $q \succeq lg(C_0(M_P))$ or $q \preceq lg(C_1(M_P))$ holds. Since $G(lg(C_0(M_P))) \cap G(lg(C_1(M_P))) = \phi$, for any atom $q \preceq lg(C_1(M_P))$, it holds that $q \not\preceq lg(C_0(M_P))$. Hence, for any atom q appearing in $2\text{-}mmg(S_\Delta)$, it holds that $lg(C_0(M_P)) \not\succ q$.

Let α be a minimal size element of M_P. Since $\alpha \in C_0(M_P) = G(lg(C_0(M_P)))$ and $G(lg(C_0(M_P))) \cap G(lg(C_1(M_P))) = \phi$, it holds that $\alpha \notin G(lg(C_1(M_P)))$ and also $\alpha \notin G(q)$ for any $q \preceq lg(C_1(M_P))$.

Thus, after the algorithm has fed all elements in S and α, the atom h_0' in the algorithm is always found and satisfies that $h_0' \equiv lg(C_0(M_P))$. From the way to calculate $2\text{-}mmg(S_\Delta)$ [ASO91b] where $S_\Delta \supseteq S$, for any pair $\{h_0', h_1'\} \in 2\text{-}mmg(S_\Delta)$, it necessarily holds that $h_1' \equiv lg(C_1(M_P))$. Hence, after the algorithm has fed all elements in S and α, it continues to output $pr(P(S \cup \{\alpha\}, \langle lg(C_0(M_P)), lg(C_1(M_P))\rangle))$. With Lemma 4.17 and Theorem 4.18, this proves that the algorithm identifies M_P in the limit.

Clearly, the algorithm is consistent, because it outputs either $pr(P(S, \langle h_0', h_1'\rangle))$ or $lg(S)$. Since $|2\text{-}mmg(S)|$ is bounded by a polynomial in $size(S)$, we can show that the algorithm produces each output in time polynomial in $size(S)$ by a similar argument to the proof of the previous theorem. □

Chapter 5

Model Inference with Predicate Invention