Rule-Based Abduction for Logic Programming

Hirata, Kouichi
Research Institute of Fundamental Information Science Kyushu University

出版情報：RIFIS Technical Report. 106, 1995-03-25. Research Institute of Fundamental Information Science, Kyushu University

RIFIS-TR-CS-106

RIFIS Technical Report

Rule-Based Abduction for Logic Programming

Kouichi Hirata

March 25, 1995

Research Institute of Fundamental Information Science

Kyushu University 33

Fukuoka 812, Japan

E-mail:hirata@rifis.kyushu-u.ac.jp Phone:092-641-1101 ex. 4478

Abstract

In order to capture the nature of inference, a philosopher Peirce classified inference into three fundamental kinds: *deduction*, *induction*, and *abduction*. In this classification, which based on the form of syllogisms, abduction is characterized as the inference of a case A from a rule $A \to C$ and a result C. Furthermore, he also placed these three kinds of inference at each stage of scientific inquiry. According to him, every scientific inquiry begins with an observation of a surprising fact. The first stage, *abduction*, of scientific inquiry proposes a hypothesis to explain why the fact arises. The second stage, *deduction*, derives new conclusions from the hypothesis. The third stage, *induction*, tests empirically or corroborates the hypothesis and the conclusions. Hence, abduction is not only a kind of inference, but also a method of scientific discovery. The inference schema of abduction as the first stage of scientific inquiry is described in the following three steps:

1. A surprising fact C is observed.
2. If A were true, then C would be a matter of course.
3. Hence, there is reason to suspect that A is true.

In computer science, the second stage, deduction, has been developed from viewpoints of automated theorem proving and logic programming. The third stage, induction, has been studied from viewpoints of inductive inference and machine learning. For the first stage, abduction, there are also many researches in various fields. In order to systematically understand them and clearly discuss abduction, first we classify abduction into five types: *rule-selecting abduction*, *rule-finding abduction*, *rule-generating abduction*, *theory-selecting abduction*, and *theory-generating abduction*. In this thesis, we examine such various researches on abduction so far developed, and show that most of them can be placed in our classification. Furthermore, we investigate the first three types of abduction, which we call together *rule-based abduction*, for logic programming.

The *rule-selecting abduction* for logic programming is abduction which selects a rule in a program and proposes a hypothesis to explain a surprising fact. From the philosophical viewpoint we mentioned above, we should consider the process of abduction which terminates. Hence, it is a main purpose in this thesis to identify the class

of logic programs for which the process of abduction terminates. We first introduce the concept of *head-reducing* programs. Then, we show that all the derivations for a head-reducing program and a surprising fact are finite. Hence, all the processes of rule-selecting abduction for a head-reducing program are finite.

In general, abduction is closely related to nonmonotonic reasoning. Thus, in this thesis, we compare rule-selecting abduction with default logic. In order to formulate the rule-selecting abduction for default logic, we define a surprising fact and a hypothesis in the default logic. We show that, if there exists a hypothesis which explains a surprising fact, then there also exists an extension of a given default theory, which includes the surprising fact. This extension is corresponding to the least Herbrand model of the definite program obtaining from the default theory.

Furthermore, we extend the concept of head-reducingness to that of *breadth-first head-reducing* programs, and the rule-selecting abduction to the *breadth-first rule-selecting abduction*. We also show that there exists a finite derivation for a breadth-first head-reducing program and a surprising fact. Hence, the process of breadth-first rule-selecting abduction for a breadth-first head-reducing program is finite.

The *rule-finding abduction* for logic programming is abduction which finds a rule in a program in the set of programs and proposes a hypothesis to explain a surprising fact. In rule-finding abduction, we are interested in how to choose programs from the set of programs. Then, we pay our attention to choosing programs for which the process of rule-finding abduction terminates.

We introduce two concepts of *loop-pair* and *loop-elimination*. The *loop-pair* syntactically determines whether or not there exists an infinite process of rule-finding abduction for the choice of programs. We show that, if a *loop-pair* appears in a derivation, then the derivation becomes infinite. On the other hand, the *loop-elimination* is a transformation of programs. By using loop-elimination, we can choose the programs for which rule-finding abduction terminates. We also show that, for given two programs, if we transform one program by *loop-elimination*, then all the derivations for union of the transformed program and the rest are finite. In other words, by *loop-elimination*, we can choose the programs whose proof trees have no infinite branches.

In this thesis, we also discuss analogical reasoning from the viewpoint of abduction. In this thesis, we adopt the formulation of analogical reasoning by Haraguchi and Arikawa. In their formulation, the main problem is how to detect an analogy. In order to solve this problem, we also adopt the concept of *partial isomorphic generalizations*. By using these concepts, we introduce the concept of *deducible hypotheses*, and formulate *rule-finding abduction with analogy*. We show that a deducible hypothesis is correct in the sense of analogical reasoning, and show that it is polynomial time computable with respect to the length of a surprising fact and the size of a proof tree. We design an algorithm of rule-finding abduction with analogy, and realize it as a Prolog program.

The *rule-generating abduction* for logic programming is abduction which generates a rule and proposes a hypothesis to explain a surprising fact. In rule-generating abduction, only one surprising fact is given. In order to generate a rule and propose a hypothesis, we need to generalize the surprising fact.

When we deal with generalizations, we should avoid overgeneralization. It should be determined whether or not a generalization is overgeneral by an intended model. However, it is hard to give in advance such an intended model in our rule-generating abduction. Hence, we introduce a syntactical generalization of one atom, called a *safe generalization*. In general, an atom is regarded as a relation between its arguments. Then, for safe generalizations, common ground terms are replaced by common variables.

If the class of definite programs is not restricted to some subclass, there may be infinitely many meaningless hypotheses. Hence, we introduce the subclass of head-reducing programs, called *weakly 2-reducing* programs. However, without any heuristic, the number of weakly 2-reducing rules for rule-generating abduction also increases in exponential order with respect to the length of a surprising fact. On the other hand, safe generalizations in this class are characterized by only two types of substitutions.

Hence, by using two types of safe generalizations, we design an efficient algorithm of rule-generating abduction for weakly 2-reducing programs. The number of rules and hypotheses obtained by this algorithm is at most the number of the arguments in a surprising fact. We show that this algorithm generates rules and proposes hypotheses in polynomial time with respect to the length of a surprising fact. Furthermore, we show that the selected common list in some argument of a surprising fact appears in the same argument of the hypothesis proposed by this algorithm.

Acknowledgments

First and foremost I would like to thank Prof. Setsuo Arikawa who supervised me from the beginning of my research activity. At any time, in all points, he gave me instructive suggestions and warm encouragement.

I am deeply grateful to Prof. Satoru Miyano and Prof. Takeshi Shinohara for their guidance and strong support. They always encouraged me to try everything and gave me a lot of constructive comments.

I would thank Prof. Yasuo Kawahara. He supervised me very kindly when I was an undergraduate student at Department of Mathematics. I would thank Prof. Fumihiro Matsuo for helpful comments.

I am deeply indebted to many people who helped and supported me. I particularly appreciate to Hiroki Ishizaka, Tetsuhiro Miyahara, and Akihiro Yamamoto for their helpful discussions. I would thank to Hiroki Arimura and Yasuhito Mukouchi for insightful comments on the various generalizations.

I was interested in Philosophy of Science since Masayuki Takeda sent me the book 'What is this thing called science?'. The philosophical background of this thesis was strengthened by that book and his comments.

I also express my gratitude to Eiju Hirowatari for many discussions and much helpful advice about partially isomorphic generalization and analogical reasoning. Special thanks are due to him for valuable suggestions and comments on Section 5.6 in particular on implementing by Prolog.

I appreciate the help from Takeo Okazaki and Ayumi Shinohara on the computer environments. Conversations with Tomoyuki Uchida, Shinichi Shimozono, Osamu Maruyama, Masayuki Mori, Kazuhiro Kiwata, Noriko Sugimoto, and Erika Tateishi were very helpful and enjoyable.

Finally, I express my deepest thanks to my parents.

Contents

Chapter 1

Introduction

> *"I have already explained to you that what is out of the common is usually a guide rather than a hindrance. In solving a problem of this sort, the grand thing is to be able to reason backward."* — "A Study in Scarlet"

1.1 What Is Abduction?

The notion of *abduction* was first introduced by Charles Sanders Peirce, who was a philosopher, scientist and logician. He held that there were three fundamental kinds of inference: *deduction*, *induction*, and *abduction*. He classified three kinds of inference by the forms of syllogisms ([Pei65]).

(1) *Deduction* is an inference of a *result* from a *rule* and a *case*. For example, by deduction, we infer the result "these beans are white" from the rule "all the beans from this bag are white" and the case "these beans are from this bag". Deduction is characterized as follows:

rule	All the beans from this bag are white.
case	These beans are from this bag.
result	These beans are white.

By using logical formulas, the above syllogism is represented by the following one:

rule	$\forall X(\mathit{from\_this\_bag}(X) \rightarrow \mathit{white}(X))$
case	$\mathit{from\_this\_bag}(\mathit{these\_beans})$
result	$\mathit{white}(\mathit{these\_beans})$

(2) *Induction* is an inference of a *rule* from a *case* and a *result*. By induction, we infer the rule "all the beans from this bag are white" from the case "these beans are from this bag" and the result "these beans are white". Induction is characterized as follows:

case	These beans are from this bag.
result	These beans are white.
rule	All the beans from this bag are white.

By using logical formulas, the above syllogism is represented by the following one:

case	$\mathit{from\_this\_bag}(\mathit{these\_beans})$
result	$\mathit{white}(\mathit{these\_beans})$
rule	$\forall X(\mathit{from\_this\_bag}(X) \rightarrow \mathit{white}(X))$

(3) *Abduction* is an inference of a *case* from a *rule* and a *result*. By abduction, we infer the case "these beans are from this bag" from the rule "all the beans from this bag are white" and the result "these beans are white". Abduction is characterized as follows:

rule	All the beans from this bag are white.
result	These beans are white.
case	These beans are from this bag.

By using logical formulas, the above syllogism is represented by the following one:

rule	$\forall X(\mathit{from\_this\_bag}(X) \rightarrow \mathit{white}(X))$
result	$\mathit{white}(\mathit{these\_beans})$
case	$\mathit{from\_this\_bag}(\mathit{these\_beans})$

According to Peirce, deduction is called *analytic inference*, while induction and abduction are called *synthetic inference*. Analytic inference is merely an application of general rules to particular cases, which is logically valid. On the other hand, synthetic inference brings on an extension of our empirical knowledge, which is not always logically valid.

Hence, we classify these three kinds of inference as follows (*ibid.*):

$$\text{inference}\begin{cases}\textit{analytic}\text{ inference} - & \textit{deduction}\\ \textit{synthetic}\text{ inference} & \begin{cases}\textit{induction}\\ \textit{abduction}\end{cases}\end{cases}$$

Peirce not only classified these three kinds of inference, but also placed them at each stage of scientific inquiry. He asserted that every scientific inquiry consists of the following three stages.

(1) *Every inquiry whatsoever takes its rise in the observation of some surprising phenomenon. At length a conjecture arises that furnishes a possible explanation* (*ibid.*). Then, the first stage *abduction is the process of forming an explanatory hypothesis* (*ibid.*).

(2) The second stage, *deduction*, is the process *of collecting consequences of the hypothesis* (*ibid.*).

(3) The third stage, *induction*, is the process *of ascertaining how far those consequents accord with experience, and of judging accordingly whether the hypothesis is sensibly correct, or requires some inessential modification, or must be entirely rejected* (*ibid.*).

In other words, every scientific inquiry begins with an observation of a surprising fact. The first stage, *abduction*, of scientific inquiry proposes a hypothesis to explain why the fact arises. The second stage, *deduction*, derives new conclusions from the hypothesis. Finally, the third stage, *induction*, tests empirically or corroborates the hypothesis and the conclusions.

Peirce claimed that *abduction, although it is very little hampered by logical rules, nevertheless is logical inference* (*ibid.*). Then, the inference schema of abduction as the first stage of scientific inquiry is described in the following three steps (*ibid.*).

1. A surprising fact C is observed.

2. If A were true, then C would be a matter of course.

3. Hence, there is reason to suspect that A is true.

In general, the above inference schema is depicted by the following syllogisms:

$$\frac{C \quad A \rightarrow C}{A} \quad \text{or} \quad \frac{C \quad C \leftarrow A}{A}.$$

1.2 Philosophy of Science and Mathematics

In the philosophy of science, many philosophers discussed whether there could be a *logic of discovery*, after Peirce has discussed the logic of abduction.

Reichenbach [Reic38, Bro77, Cha79, Tha88] proposed a sharp distinction between the *context of discovery* and the *context of justification*. He claimed that the philosophy of science should be concerned only with questions of confirmation and acceptance that belong in the context of justification, and that the topic of discovery should be relegated to psychology and sociology.

Furthermore, Popper [Popp59, Cha79] pointed deeply that the work of the scientist consists in putting forward and testing theories. He also distinguished sharply between the process of conceiving a new idea, and the methods and results of examining it logically. In the former, there is no such thing as a logical method of having new ideas, or a logical reconstruction of this process. Every discovery contains an *irrational* element. In the later, the scientific knowledge is never verified, and it is only falsified. In other words, the work of the scientist consists of the context of discovery and the context of falsification.

Reichenbach and Popper adopted this sharp distinction in order to eliminate psychologism. However, some philosophers, for example Kuhn [Kuh70] and Brown [Bro77], have resisted this restriction. Brown [Bro77] claimed that, in a scientific discovery, the context of justification is a part of the context of discovery, and we cannot draw a line clearly between the context of discovery and that of justification. In the philosophy of science, the relation between justification and discovery has left unclear.

Peirce's philosophy of science in Section 1.1 is compatible with the above philosophy of Reichenbach or Popper. The first stage, abduction, of scientific inquiry is corresponding to the context of discovery. The second and the third stage, deduction and induction, are also corresponding to the context of justification or falsification.

Note that the word "logical" in the above assertion of Popper can be interpreted as universal validity in formal logic. Then, Popper's assertion can be considered that the context of discovery is not necessarily universally valid. As mentioned in Section 1.1, abduction is not valid, and also causes a *fallacy of affirming the consequent*. Hence, we can regard abduction as the context of discovery, that is, the method of scientific discovery. Hanson [Han58] advanced the claim that abduction constitutes a logic of discovery.

For the methodology of mathematics, it is also an important problem to investigate the way of discovery of mathematics. Polya [Pol54a, Pol54b, Pol57] pointed out that there exist no *infallible* rules of discovery leading to the solution of all possible mathematical problems. Furthermore, he introduced the notion of *heuristic reasoning* or *heuristic*, which appears so baffling and elusive when approached from the viewpoint of purely demonstrative logic.

According to Polya, *heuristic is reasoning not regarded as final and strict but as provisional and plausible only, whose purpose is to discover the solution of the present problem. We may need the provisional before we attain the final* [Pol57].

He characterized such heuristic as the following *heuristic syllogism*:

$$\frac{\begin{array}{c}\textit{If } A \textit{ is true, then } B \textit{ is also true, as we know.}\\ \textit{Now, it turns out that } B \textit{ true.}\end{array}}{\textit{Therefore, } A \textit{ becomes more credible.}}$$

Still shorter:

$$\frac{\begin{array}{c}\textit{If } A \textit{ then } B\\ B \textit{ true}\end{array}}{A \textit{ more credible}}$$

Furthermore, Lakatos [Lak76] developed the above Polya's mathematical discovery by incorporating with Popper's logic of scientific discovery.

It is obvious that Polya's heuristic reasoning is almost corresponding to Peirce's abduction. Hence, abduction is a suitable concept for discovery in not only the philosophy of science but also the methodology of mathematics.

1.3 Computer Science

In computer science, especially in computational logic and logic programming, many researchers have extensively studied the abduction from various viewpoints.

Plotkin [Plo71] studied abduction together with inductive generalization. There are researches of Shapiro's model inference system [Sha81] and inductive logic programming [MB88, Mug92, Lin89, LU89] as the extensions of Plotkin's work.

Muggleton [MB88] has introduced the method of *inverting resolution* to construct logic programming from finite examples. Such a methodology is called *inductive logic programming*. Inductive logic programming has been developed by Muggleton [Mug92]. Ling [Lin89, LU89] has paid his attention on the *constructive* method for inductive logic programming. These are also a kind of abduction, because they really propose hypotheses.

Genest *et al.* [GMP90] and Duval [Duv91] have suggested abduction for *explanation-based generalization*, which is an efficient technique for obtaining a general concept from examples and a background knowledge. Thagard [Tha88] has introduced *analogical abduction*, which is a kind of abduction incorporating with analogical reasoning.

Pople [Popl73, Kun87, Ino92] gave one direction for researches of abduction. There are researches of Poole's Theorist [Poo88], hypothesis-based reasoning [Kun87], and abductive logic programming [Dun91, EK89, KM90, KKT92] as the extensions of Pople's work.

Poole [Poo88] has discussed the relationship between Reiter's default logic [Reit80] and abduction, and shown that abduction can be viewed as a default logic, and implemented the system Theorist. Kunifuji [Kun87] has developed Poole's Theorist as a hypothesis-based reasoning system.

Eshghi and Kowalski [EK89] discussed the relationship between negation as failure and abduction in logic programming. Kakas and Mancarella [KM90], Dung [Dun91], and Kakas *et al.* [KKT92] have defined an abductive framework in nonmonotonic logic programming, and studied the semantics in that framework. Out of these studies there has emerged a new field of *abductive logic programming*.

There are researches of abduction in terms of a model of *belief*, which is a kind of modal logic, by Levesque [Lev89] and Selman and Levesque [SL90]. These are regarded as general extensions of Poole's logic [Poo88]. Furthermore, they have claimed that different models of belief give rise to different forms of abductive reasoning, and have constructed a model of belief for abduction. They have also studied the relationship between the models of belief, default logic, and assumption-based truth-maintenance system.

Concerning expert systems, Cox and Pietrzykowski [CP87] have investigated *diagnosis problems* by abductive inference. Pirri and Pizzuti [PP90] have also combined the diagnosis problem with the stable model semantics which is one of the semantics of logic programming. Konolige [Kon92] has introduced a *causal theory*, and compared it with abduction. Bylander *et al.* [BATJ91] has formulated abduction in order to analyze the computational complexity of abduction for propositional logic and for the diagnosis problem.

In computational linguistics, Hobbs *et al.* [HSME88] has introduced abduction in order to interpret natural language. Stickel [Sti91] has also investigated abduction deeply, and suggested a Prolog-like inference system to interpret natural language.

1.4 Classification of Abduction

In order to systematically understand the above mentioned various researches of abduction in computer science and clearly discuss abduction, we classify abduction into five types: *rule-selecting abduction*, *rule-finding abduction*, *rule-generating abduction*, *theory-selecting abduction*, and *theory-generating abduction*. The first three types of abduction is called *rule-based abduction*, and the other two types of abduction *theory-based abduction*. This new classification is based on the interpretations of syllogism and the definitions of hypothesis. We examine such various researches on abduction so far developed, and show that most of them can be placed in our classification. Furthermore, we investigate rule-based abduction for logic programming.

The *rule-selecting abduction* for logic programming is abduction which selects a rule in a program and proposes a hypothesis to explain a surprising fact. We can easily realize the rule-selecting abduction as a Prolog program, which is a variant of partial evaluation [vHB88] or meta interpreter [SS86, SS94].

From the philosophical viewpoint we mentioned in Section 1.1, abduction is the first stage of scientific inquiry. Then, we should consider the process of abduction which terminates. Hence, in this thesis, we identify the class of logic programs for which the process of abduction terminates. In order to characterize such a class, we introduce two concepts of *head-reducing* and *breadth-first head-reducing* programs. The head-reducing program is a program for which all the processes of rule-selecting abduction terminate. On the other hand, the breadth-first head-reducing program is a program for which the process of breadth-first rule-selecting abduction terminates.

In general, abduction is closely related to nonmonotonic reasoning, because both abduction and nonmonotonic reasoning are a kind of plausible inference. Thus, in this thesis, we compare rule-selecting abduction with Reiter's *default logic* [Reit80, Poo88]. Poole [Poo88] has already developed the relationship between abduction and default logic. We extend the result in Poole [Poo88] in a sense.

The *rule-finding abduction* for logic programming is abduction which finds a rule in a program in the set of programs and proposes a hypothesis to explain a surprising fact. Here, the set of programs is given in advance. In rule-finding abduction, we are interested in how to choose programs from the set of programs. Then, we pay our attention to choosing programs for which the process of rule-finding abduction terminates. It is our purpose to avoid an infinite process of rule-finding abduction when we choose the programs. Hence, we introduce two concepts of *loop-pair* and *loop-elimination*. The *loop-pair* syntactically determines whether or not there exists an infinite process of rule-finding abduction for the choice of programs. On the other hand, the *loop-elimination* is a transformation of programs. By using loop-elimination, we can choose the programs for which the process of rule-finding abduction terminates.

In computer science, there exist various researches for analogical reasoning, which is an important tool for machine learning and knowledge acquisition. In this thesis, we also discuss analogical reasoning from the viewpoint of abduction, which is called *rule-finding abduction with analogy*.

Thargad [Tha88] and Duval [Duv91] have tried to discuss abduction and analogy into the same framework. However, even in such researches, the relationship between abduction and analogy are not clear, because the concepts of abduction and analogy they used are ambiguous.

Hence, in this thesis, we adopt the formulation of analogical reasoning by Haraguchi and Arikawa [Har85, HaA86, HiA94b]. They have defined a formal analogy for definite programs as a relation between elements in Herbrand universes. In their formulation, the main problem is how to detect an analogy. In order to solve this problem, we adopt the concept of *partial isomorphic generalizations*, which has been introduced by Hirowatari and Arikawa [HiA94b]. By using these concepts, we introduce the concept of *deducible hypotheses*, and formulate *rule-finding abduction with analogy*, which is an extension of rule-finding abduction. We also design an algorithm for rule-finding abduction with analogy, and realize it as a Prolog program.

The *rule-generating abduction* for logic programming is abduction which generates a rule and proposes a hypothesis to explain a surprising fact. In rule-generating abduction, only one surprising fact is given. In order to generate a rule and propose a hypothesis, we need to generalize a surprising fact.

A *generalization* is an important tool for inductive logic programming, program synthesis, and machine learning. Plotkin introduced and developed the *least generalization* and the *relative least generalization* [Plo70, Plo71]. Arimura *et al.* have developed Plotkin's least generalization as *minimal multiple generalization* [ASO91]. Note that all of these researches are on the generalization of at least two atoms. Thus, the following problem arises: Is the generalization of one atom worth or worthless? Hirowatari and Arikawa [HiA94b] have answered this problem affirmatively in the framework of analogical reasoning, by using the concept of *partially isomorphic generalizations*.

When we deal with generalizations, we should avoid overgeneralization. It should be determined whether or not a generalization is overgeneral by an intended model. However, it is hard to give in advance such an intended model in our rule-generating abduction. Hence, we introduce a syntactical generalization of one atom, called a *safe generalization*. In general, an atom is regarded as a relation between its arguments. Then, for safe generalizations, common ground terms are replaced by common variables.

In rule-generating abduction, if the class of definite programs is not restricted to some subclass, there may be infinitely many meaningless hypotheses. Hence, we introduce the subclass of head-reducing programs, called *weakly 2-reducing* programs. Many typical Prolog programs are included in this class. However, without any heuristic, the number of weakly 2-reducing rules for rule-generating abduction also increases in exponential order with respect to the length of a surprising fact. In order to obtain the hypotheses efficiently by using safe generalizations, we design an algorithm of rule-generating abduction for weakly 2-reducing programs.

1.5 Outline of This Thesis

This thesis is organized as follows:

In Chapter 2, we prepare some notions to be necessary in the following chapters.

In Chapter 3, we classify abduction into five types. We examine various researches of abduction in computer science, and show that most of them can be placed in our classification.

In Chapter 4, we investigate rule-selecting abduction for logic programming. First, we prepare the notions of *recursive definition* and *recursive program*, which are valuable tools in order to analyze abduction for logic programming. By using these notions, we introduce the concept of *head-reducing* programs. Note that, in this thesis, we characterize the termination of abduction as the finiteness of derivations. Then, we show that all the derivations for a head-reducing program and a surprising fact are finite.

Furthermore, we compare rule-selecting abduction with default logic. In order to formulate rule-selecting abduction for default logic, we define a surprising fact and a hypothesis in a default logic. We show that, if there exists a hypothesis which explains a surprising fact, then there also exists an extension of a given default theory, which includes the surprising fact. This extension of the default theory is corresponding to the least Herbrand model of the definite program obtained from the default theory.

Since the class of *head-reducing* programs is not so large, we extend this concept to that of *breadth-first head-reducing* programs, and the rule-selecting abduction to the *breadth-first rule-selecting abduction*. Then, we also show that there exists a finite derivation for a breadth-first head-reducing program and a surprising fact.

In Chapter 5, we investigate rule-finding abduction for logic programming. First, we introduce two concepts of *loop-pair* and *loop-elimination*. We show that, if a *loop-pair* appears in a derivation, then the derivation becomes infinite. We also show that, for given two programs, if we transform one program by *loop-elimination*, then all the derivations for union of the transformed program and the rest are finite. In other words, by *loop-elimination*, we can choose the programs whose proof trees have no infinite branches.

Furthermore, we introduce the concept of *deducible hypotheses*, and formulate *rule-finding abduction with analogy*. In rule-finding abduction with analogy, the main problem is how to detect an analogy while constructing a deducible hypothesis. Then, we adopt the concept of *partially isomorphic generalizations*. In this concept, an analogy is regarded as a function. We show that a deducible hypothesis is correct in the sense of analogical reasoning. Also we show that a deducible hypothesis is polynomial time computable with respect to the length of a surprising fact and the size of a proof tree. We design an algorithm of rule-finding abduction with analogy concretely, and realize it by a Prolog program.

In Chapter 6, we investigate rule-generating abduction for logic programming. We formulate a *safe generalization*, which is based on the forms of atoms and substitutions instead of an intended model, and show some properties of safe generalizations. Also we introduce the subclass of head-reducing programs, called *weakly 2-reducing* programs. Unfortunately, we show that the number of hypotheses in this class also increases in exponential order with respect to the length of a surprising fact.

On the other hand, in weakly 2-reducing programs, there are only two types of terms, constant symbols and lists. Then, safe generalizations in this class are characterized by only two types of substitutions, *constant substitutions* and *list substitutions*. A constant substitution θ_c consists of bindings $X := c$, where c is a constant symbol, while a list substitution θ_l consists of bindings $X := l$, where l is a list. For these substitutions, we investigate the condition under which the generalization is safe with respect to the composition $\theta_c\theta_l$ of θ_c and θ_l.

Hence, by using such two types of safe generalizations, we design an algorithm of rule-generating abduction for weakly 2-reducing programs. The number of rules and hypotheses obtained by this algorithm is at most the number of the arguments in a surprising fact. We show that this algorithm generates rules and proposes hypotheses in polynomial time with respect to the length of a surprising fact. Furthermore, we show that the selected common list in some argument of a surprising fact appears in the same argument of the hypothesis proposed by this algorithm.

Chapter 2

Preliminary

"The case," said Sherlock Holmes, ..., "is one where, as in the investigations which you have chronicled under the names of the 'Study in Scarlet' and of the 'Sign of Four', we have been compelled to reason backward from effects to causes." — *'The Adventure of the Cardboard Box'*

"The Memories of Sherlock Holmes"

In this chapter, we give some basic notions and notational conventions needed in this thesis. We use fundamental concepts from first order logic and logic programming. More precise information on these concepts would be found in [CL73, Llo87, Men87, SS86, SS94].

In Section 2.1, we give definitions concerned with logic programming. In Section 2.2, we introduce Reiter's default logic [Reit80] for the discussion in Section 4.2. In Section 2.3, we introduce the formal definition of analogical reasoning by Haraguchi and Arikawa [Har85, HaA86] for the discussion in Section 5.6. In Section 2.4, we discuss the partially isomorphic generalization [HiA94b] for the discussion in Section 5.6 and Section 6.4.

2.1 Logic Programming

2.1.1 Basic definitions

A *first order theory* consists of an alphabet, a first order language, a set of axioms, and a set of inference rules. A first order language $\mathcal{L}$ consists of the well-formed formulas of the theory. The axioms are a designated subset of well-formed formulas. The axioms and rules of inference are used to derive the theorems of the theory. We now proceed to define the alphabet and the first order language.

Definition 2.1 *An alphabet consists of the following symbols:*

1. *Variables, denoted by the letters* X, Y, Z, W, U *and* V *possibly subscripted.*

2. *Function symbols, denoted by the letters* f, g *and* h *possibly subscripted.*

3. *Constant symbols, which are* 0*-ary function symbols, denoted by the letters* a, b *and* c *possibly subscripted.*

4. *Predicate symbols (or predicates, for short), denoted by the letters* p, q *and* r *possibly subscripted.*

5. *Logical symbols, which are* $\neg, \vee, \wedge, \rightarrow, \forall$ *and* $\exists$.

6. *Punctuation symbols, which are "(", ")" and ",".*

In logic programming, the symbol "$\rightarrow$" of logical implication is represented by the symbol "$\leftarrow$" with inverse direction.

Definition 2.2 *A term, denoted by the letters* t, s, u, v, *and* w *possibly subscripted, is defined inductively as follows:*

1. *A variable is a term.*

2. *A constant symbol is a term.*

3. *If* f *is an* n*-ary function symbol and* $t_1, \cdots, t_n$ *are terms, then* $f(t_1, \cdots, t_n)$ *is a term.*

For a term t, $|t|$ denotes the *length* of t, that is, the number of all occurrences of symbols in t except punctuation symbols. For example, $|a|$ is 1 and $|f(f(a))|$ is 3.

Definition 2.3 *A (well-formed) formula is defined inductively as follows:*

1. *If p is an n-ary predicate symbol and $t_1, \cdots, t_n$ are terms, then $p(t_1, \cdots, t_n)$ is a formula, called an atomic formula or an atom, and is denoted by α, β, and γ.*

2. *If F and G are formulas, then so are $\neg F, F \vee G, F \wedge G, F \rightarrow G$.*

3. *If F is a formula and X is a variable, then $\forall X(F)$ is a formula.*

For any atom α, we denote the predicate symbol of α by $pred(\alpha)$.

The *first order language* $\mathcal{L}$ is given by an alphabet consists of the set of all formulas constructed from the symbols of the alphabet.

The notions of $\vdash$ and $\models$ represent the provability and the satisfiability as in the general first order logic [CL73, Llo87, Men87]. The set G of formulas is *consistent* if there exists no formula α such that $G \vdash \alpha$ and $G \not\vdash \alpha$.

The *scope* of $\forall X$ in $\forall X(F)$ is F. A *bound occurrence* of a variable in a formula is an occurrence immediately following a quantifier or an occurrence within the scope of a quantifier, which has the same variable immediately after the quantifier. Any other occurrence of a variable is *free* . If F is a formula, then $\forall(F)$ denotes the *universal closure* of F, which is the closed formula obtained by adding a universal quantifier for every variables having a free occurrence in F.

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

$$\forall(A_1 \vee \cdots \vee A_m \vee \neg B_1 \vee \cdots \vee \neg B_n),$$

where $A_1, \cdots, A_m, B_1 \cdots, B_n$ are atoms and $n, m \geq 0$. We denote the above clause by the following forms:

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

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

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

$$C = A \leftarrow B_1, \cdots, B_n.$$

Here, A is called the *head* of C, denoted by $head(C)$, and $B_1, \cdots, B_n$ is called the *body* of C. A clause with an empty body, that is, in the case $n = 0$, is called a *unit* *clause* or a *fact*. In particular, the clause whose head has the predicate symbol p is called a *definition clause* of p. We identify a unit clause $A \leftarrow$ with an atom A. A *definite program* (*program*, for short) is a finite set of definite clauses. We sometimes represents $P = R \cup F$ for a definite program P, where F is a set of all unit clauses in P and $R = P - F$, that is the set of all definite clauses without unit clauses in P.

A *goal* is the clause of the form:

$$\leftarrow B_1, \cdots, B_n.$$

In Prolog programs, the symbol "$\leftarrow$" of logical implication is represented by the symbol "`:-`". In particular, a goal $\leftarrow B_1, \cdots, B_n$ is represented by the following form:

```
?:- B1, ··· ,Bn.
```

In this thesis, we use the typewriter font with the symbol "`:-`" for Prolog programs.

A *word* is either a term or an atom. An *expression* is either a word, a clause, or a definite program. When no variable appears in an expression, we sometimes call it a *ground expression* to emphasis this fact. Thus, we may use a *ground term*, a *ground atom*, and a *ground clause* to mean that no variable occurs in the respective expression.

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

$$\{X_1 := t_1\sigma, \cdots, X_n := t_n\sigma, Y_1 := s_1, \cdots, Y_m := s_m\}$$

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 $\theta = \{X_1 := t_1, \cdots, X_n := t_n\}$ 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.

Let S be a finite set $\{w_1, \cdots, w_n\}$ of words. A substitution θ is a *unifier* of S if $w_1\theta = \cdots = w_n\theta$. If there exists a unifier for S, then S is said to be *unifiable*. Also, words w_1 and w_2 are said to be unifiable if the set $\{w_1, w_2\}$ is unifiable. For a unifiable set, a unifier θ of S is called *most general unifier* (*mgu*, for short) if, for every unifier σ of S, there exists a substitution λ such that $\sigma = \theta\lambda$.

2.1.2 Herbrand model

For a definite program P, $K(P), F^n(P)$, and $\Pi^n(P)$ denote all constant symbols in P, all n-ary function symbols in P, and all n-ary predicate symbols in P respectively. $\Pi(P)$ denotes all predicate symbols in P.

For a definite program P, $H_i(P)$ $(i \geq 0)$ is defined as follows:

$$H_0(P) = \begin{cases} K(P) & \text{if } K(P) \neq \phi, \\ \{a\} & \text{otherwise,} \end{cases}$$

$$H_{i+1}(P) = H_i(P) \cup \{f(t_1 \cdots, t_n) \mid f \in F^n(P), t_j \in H_i(P), n \geq 1\} \ (i \geq 0).$$

Then, the *Herbrand universe* $H(P)$ of P is:

$$H(P) = \bigcup_{i \geq 0} H_i(P).$$

For a definite program P, the *Herbrand base* $B(P)$ of P is:

$$B(P) = \{p(t_1, \cdots, t_n) \mid p \in \Pi^n(P), t_i \in H(P), n \geq 1\}.$$

The *Herbrand interpretation* I_P of P is the interpretation given as follows:

1. The domain of the interpretation is the Herbrand universe $H(P)$.

2. For any $a \in H_0(P)$, $I_P(a) = a$.

3. For any $f \in F^n(P)$ and $t_1, \cdots, t_n \in H(P)$,

$$I_P(f)(I_P(t_1), \cdots, t_n)) = f(t_1, \cdots, t_n).$$

4. For any $p \in \Pi^n(P)$, $I_P(p)$ is a mapping from $B(P)$ to $\{0,1\}$.

For a definite program P, the *Herbrand model* $M(P)$ of P is the Herbrand interpretation I_P of P such that $I_P \models P$.

Let P be a definite program and $\{M_i(P)\}_{i \in I}$ be a non-empty set of Herbrand models of P. Then, the intersection $\bigcap_{i \in I} M_i(P)$ of $M_i(P)$, called *the least Herbrand model* of P and denoted by $M(P)$, is also an Herbrand model of P [Llo87]. Hence, we adopt this model as the model-theoretic semantics for logic programming.

2.1.3 SLD-resolution, SLD-tree, and proof tree

Let G be a goal $\leftarrow A_1, \cdots, A_m, \cdots, A_k$ and C be a definite clause $A \leftarrow B_1, \cdots, B_q$. Then, a goal G' is *derived* from G and C using mgu θ if the following conditions hold:

1. A_m is an atom, called the *selected* atom, in G.

2. θ is an mgu of A_m and A.

3. G' is the goal $\leftarrow (A_1, \cdots, A_{m-1}, B_1, \cdots, B_q, A_{m+1}, \cdots, A_k)\theta$

In resolution terminology, G' is called a *resolvent* of G and C.

Let P be a definite program and G be a definite goal. An *SLD-derivation*, or *derivation*, of $P \cup \{G\}$ consists of a (finite or infinite) sequence $G = G_0, G_1, \cdots$ of goals, a sequence $C_1, C_2, \cdots$ of variants of definite clauses of P, and a sequence $\theta_1, \theta_2, \cdots$ of mgu's such that each G_{i+1} is derived from G_i and C_{i+1} using θ_i. Each definite clause $C_1, C_2, \cdots$ is called an *input clause* of the derivation.

An SLD-derivation may be *finite* or *infinite*. A finite SLD-derivation may be successful or failed. A *successful* SLD-derivation is one that ends in the empty clause. A *failed* SLD-derivation is one that ends in a non-empty goal with the property that the selected atom in this goal does not unify with the head of any definite clause.

For a program P and a goal G, an *SLD-tree* for $P \cup \{G\}$ is a tree satisfying the following conditions:

1. Each node if the tree is a (possible empty) definite goal.

2. The root node is G.

3. Let $\leftarrow A_1 \cdots, A_m, \cdots, A_k$ $(k \geq 1)$ be a node in the tree and suppose that A_m is the selected atom. Then, for each input clause $A \leftarrow B_1, \cdots, B_q$ such that A_m and A are unifiable with mgu θ, the node has a child

$$\leftarrow (A_1, \cdots, A_{m-1}, B_1, \cdots, B_q, A_{m+1}, \cdots, A_k)\theta.$$

4. Nodes which are the empty clause have no children.

Each branch of an SLD-tree is corresponding to an SLD-derivation of $P \cup \{G\}$.

For a program P and a ground atom α, a *proof tree* of α on P is a tree which satisfies the following conditions:

1. Each node of the tree is an atom.

2. The root node is α.

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

Note that the condition 1 of proof tree is different from the following general definition of proof tree:

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

Under the above conditions 1', 2, and 3, it is a problem whether or not the nodes of the proof tree are elements of an Herbrand model of P. On the other hand, in this thesis, since we are interested in the forms of the nodes of the proof tree, we adopt the condition 1 instead of the condition 1'. A proof tree is corresponding to an SLD-derivation, and a branch of an SLD-tree.

2.2 Default Logic

Default logic, introduced by Reiter [Reit80], is an important tool for nonmonotonic reasoning. Poole [Poo88] investigated the relationship between default logic and abductive framework. This thesis deeply investigates this relationship in Section 4.2. In this section, we prepare the basic notions for default logic.

A *default* is an expression of the following form:

$$\frac{\alpha(\overline{X}) : \beta_1(\overline{X}), \cdots, \beta_m(\overline{X})}{w(\overline{X})},$$

where $\alpha(\overline{X}), \beta_i(\overline{X}), w(\overline{X})$ are atoms whose free variables are among those of $\overline{X} = X_1, \cdots, X_n$. In particular, a default with the following form is called a *normal default*:

$$\frac{\alpha(\overline{X}) : w(\overline{X})}{w(\overline{X})}.$$

A *default theory* is a pair (D, W), where D is a set of defaults and W is a set of closed formulas. A *normal default theory* is a pair (D, W), where D is a set of normal defaults and W is a set of closed formulas. In this thesis, since we deal with a definite program, W is assumed a definite program.

In default logic, it is a main problem to construct the set of formulas assumed true, which is called an *extension*. Then, we define the concept of an extension for default theory as follows.

Definition 2.4 *Let $\Delta = (D, W)$ be a default theory and*

$$\frac{\alpha(\overline{X}) : \beta_1(\overline{X}), \cdots, \beta_m(\overline{X})}{w(\overline{X})} \in D.$$

For any set S of closed formulas, let $\Gamma(S)$ be the smallest set satisfying the following conditions:

1. $W \subseteq \Gamma(S)$,
2. $Th(\Gamma(S)) = \Gamma(S)$, *and*
3. *if* $\frac{\alpha(\overline{X}) : \beta_1(\overline{X}), \cdots, \beta_m(\overline{X})}{w(\overline{X})} \in D$, $\Gamma(S) \vdash \alpha$, *and* $S \not\vdash \neg\beta_j$ $(1 \leq j \leq m)$, *then* $w \in \Gamma(S)$.

In condition 2, $Th(\Gamma(S))$ means the set $\{\alpha \mid \Gamma(S) \vdash \alpha\}$ of theorems for $\Gamma(S)$. Then, a set E of closed formulas is an extension for Δ if $\Gamma(E) = E$.

Reiter [Reit80] has shown the following two theorems.

Theorem 2.1 (Reiter [Reit80]) *Let E be a set of closed formulas, and $\Delta = (D, W)$ be a default theory. Define*

$$E_0 = W,$$

and for any i

$$E_{i+1} = Th(E_i) \cup \left\{ w \;\middle|\; \frac{\beta : w}{w} \in D, \beta \in E_i, \neg w \notin E \right\}.$$

Then, E is an extension for Δ if and only if $E = \bigcup_{i \geq 0} E_i$.

Theorem 2.2 (Reiter [Reit80]) *Every normal default theory has an extension.*

A default theory has an *inconsistent extension* if one of its extensions is the set of all closed formulas of $\mathcal{L}$. As an immediate corollary of Theorem 2.1 we have:

Corollary 2.1 (Reiter [Reit80]) *A default theory (D, W) has an inconsistent extension if and only if W is inconsistent.*

A default theory is *consistent* if it has a consistent extension. By Corollary 2.1, if (D, W) is consistent, then W is consistent.

2.3 Analogical Reasoning

In computer science, there are various researches for analogical reasoning. In Section 5.6, we discuss abduction and analogical reasoning in the same framework. In this thesis, we adopt the analogical reasoning introduced by Haraguchi and Arikawa [Har85, HaA86, HiA94b].

Haraguchi and Arikawa [Har85, HaA86, HiA94b] formulated analogical reasoning for logic programming, and defined a formal analogy as the relation between elements in Herbrand universes. In this section, we prepare some concepts on analogical reasoning necessary for the discussion in Section 5.6.

Let P_b and P_t be programs. The program P_b is called a *base* program and P_t a *target* program. Then, a finite set $\varphi \subseteq U(P_b) \times U(P_t)$ is called a *pairing*, where $U(P_b)$ and $U(P_t)$ are Herbrand universes for P_b and P_t, respectively. We assume implicitly that $U(P_b) \cap U(P_t) \neq \phi$.

Definition 2.5 *Let $\varphi \subseteq U(P_b) \times U(P_t)$ be a pairing. The set $\varphi^+ \subseteq U(P_b) \times U(P_t)$ is defined to be the smallest set that satisfies the following conditions:*

1. $\varphi \subseteq \varphi^+$.

2. *If $\langle t_1, s_1 \rangle, \cdots, \langle t_n, s_n \rangle \in \varphi^+$, then $\langle f(t_1, \cdots, t_n), f(s_1, \cdots, s_n) \rangle \in \varphi^+$.*

Definition 2.6 *Let α and β be ground atoms, and $\varphi \subseteq U(P_b) \times U(P_t)$ be a pairing. Then, α and β are identical by φ, denoted by $\alpha\varphi\beta$, if α, β, and φ satisfy the following condition:*

$$\begin{aligned}
&\alpha = p(t_1, \cdots, t_n),\\
&\beta = p(s_1, \cdots, s_n),\\
&\langle t_i, s_i \rangle \in \varphi^+ \ (1 \leq i \leq n).
\end{aligned}$$

Definition 2.7 *Let $\varphi \subseteq U(P_b) \times U(P_t)$ be a pairing. Then, φ is a partial identity between P_b and P_t if φ^+ is a one-to-one relation.*

In order to discuss analogical reasoning from the viewpoint of abduction in Section 5.6, we introduce the following notations: Let $\varphi \subseteq U(P_b) \times U(P_t)$ be a partial identity between P_b and P_t.

1. Let t and s be terms in P_b and P_t, respectively. Then, $t\varphi$ is a term which is obtained by replacing any term t' in t such that $\langle t', s' \rangle \in \varphi$ with a term s'. Similarly, φs is a term which is obtained by replacing any term s' in s such that $\langle t', s' \rangle \in \varphi$ with a term t'.

2. Let $\alpha = p(t_1, \cdots, t_n)$ and $\beta = p(s_1, \cdots, s_m)$ be atoms in P_b and P_t, respectively. Then, atoms $\alpha\varphi$ and $\varphi\beta$ are defined as follows:

$$\begin{aligned}
\alpha\varphi &= p(t_1\varphi, \cdots, t_n\varphi),\\
\varphi\beta &= p(\varphi s_1, \cdots, \varphi s_m).
\end{aligned}$$

3. Let $C = A \leftarrow A_1, \cdots, A_n$ and $D = B \leftarrow B_1, \cdots, B_m$ be clauses in P_b and in P_t, respectively. Then, clauses $C\varphi$ and φD are defined as follows:

$$C\varphi = A\varphi \leftarrow A_1\varphi, \cdots, A_n\varphi,$$
$$\varphi D = \varphi B \leftarrow \varphi B_1, \cdots, \varphi B_m.$$

4. Let $P_b = \{C_1, \cdots, C_n\}$ and $P_t = \{D_1, \cdots, D_m\}$. Then, programs $P_b\varphi$ and φP_t are respectively defined as follows:

$$P_b\varphi = \{C_1\varphi, \cdots, C_n\varphi\},$$
$$\varphi P_t = \{\varphi D_1, \cdots, \varphi D_m\}.$$

2.4 Partially Isomorphic Generalization

Hirowatari and Arikawa [HiA94b] introduced the concept of a *partially isomorphic generalization*, which is a generalization of one atom and is the useful tool for analogical reasoning. In this section, we prepare the notions for partially isomorphic generalizations to be necessary in Section 5.6 and 6.4.

Let α be an atom. A term t is a *replaceable term* of α if t is a constant symbol or a term $f(X_1, \cdots, X_n)$, where f is a function symbol and each X_i is a variable which does not appear in the other terms in α. For a replaceable term t of α, let $\alpha[t]$ be an atom obtained by replacing each t in α by a new variable Z which does not appear in α. Then, we write $\alpha \to \beta$ when $\alpha[t]$ is a variant of β. We define $\to^*$ as the reflexive and transitive closure of $\to$.

Definition 2.8 (Hirowatari and Arikawa [HiA94b]) *Let α and β be atoms. Then, β is a partially isomorphic generalization of α if $\alpha \to^* \beta$.*

For a set of atoms S, let $[S]$ denote the equivalence class of all atoms in S. In particular, for any $\alpha \in [S]$ and $\beta \in [S]$, α is a variant of β.

We can develop analogical reasoning [Har85, HaA86] by the notions of partially isomorphic generalizations. Hirowatari and Arikawa [HiA94b] have shown the following three theorems.

Theorem 2.3 (Hirowatari and Arikawa [HiA94b]) *Let α be an atom and S be the set of all partially isomorphic generalizations of α. Then, $[S]$ is a lattice whose partial* *order is $\rightarrow^*$, meet operator is the greatest instantiation, and join operator is the least generalization.*

Theorem 2.4 (Hirowatari and Arikawa [HiA94b]) *Let α be a ground atom $p(t_1, \cdots, t_n)$ and $k = |t_1| + \cdots + |t_n|$. Then, a partially isomorphic generalization of α can be computed in $O(k^2)$ time.*

Theorem 2.5 (Hirowatari and Arikawa [HiA94b]) *Let α and β be ground atoms in P_b and P_t, respectively, and α' be the greatest partially isomorphic generalization of α. If there exists a substitution θ such that $\alpha' = \beta\theta$, then there exists an analogy $\varphi \subseteq U(P_b) \times U(P_t)$ such that $\alpha\varphi\beta$.*

Here, an analogy φ is regarded as a partial function from $U(P_b)$ to $U(P_t)$. By partially isomorphic generalizations, we can obtain the analogy which is guaranteed one direction of partial identity.

Chapter 3

Classification of Abduction

"Deeply interested – yes. There is a thread here which we have not yet grasped, and which might lead us through the tangle."

— *'The Adventure of the Devil's Foot'*

"His Last Bow"

In Chapter 1, the inference schema of abduction has been depicted by the following syllogism:

$$\frac{C \quad A \to C}{A}.$$

The following examples of A and C in the above inference schema are found in literature:

(a) C : 'these beans are white',
A : 'these beans are from this bag' [Pei65, Ino92];

(b) C : 'I heard somebody scream at midnight',
A : 'I thought she was attacked' [Uey79];

(c) C : 'I met a man upon horseback,
surrounded by four horsemen holding a canopy over his head',
A : 'I inferred that he was the personage' [Pei65, Yon82];

(d) C : 'fossil shells are found, but far in the interior of the country',
A : 'the sea once washed over this land' [Pei65, Yon82];

(e) C : 'numberless documents and monuments refer to a conqueror
called Napoleon Bonaparte',
A : 'Napoleon Bonaparte really existed' [Pei65, Yon82];

(f) C : 'the Atlantic coastline in Africa and America are similar',
A : 'the continental drift theory' [Uey79];

(g) C : 'the evolutionary fact remaining of fossil',
A : 'the theory of natural selection in biology' [Uey79];

(h) C : 'the data of observations of planets by Tycho Brahe',
A : 'an orbit of planets is an oval (Kepler's first law)' [Pei65, Uey79, Yon82].

For the above examples, Peirce showed that there exist the following three types of *explanatory hypotheses*, which are proposed by abduction [Yon82].

(1) The first type is an explanatory hypothesis on the facts which *can be confirmed*, even if it is not confirmed at the abduction. The examples (a), (b), and (c) belong to this type.

(2) The second type is an explanatory hypothesis on the facts which *physically cannot be confirmed.* The examples (d) and (e) belong to this type, because we cannot confirm that there used to be a sea and there existed Napoleon Bonaparte.

(3) The third type is an explanatory hypothesis on the facts which *in practice and in principle cannot be confirmed* by our scientific knowledge. The examples (f), (g), and (h) belong to this type, because each hypothesis A cannot be derived from the scientific knowledge they had at that time.

Peirce expressed these three types of abduction by just *one* syllogism. This is obviously unreasonable. These types of abduction should be expressed by different syllogisms, which is a point we want to make in this thesis.

In this chapter, we apply these three types to the abduction in computer science. In Section 3.1, we introduce the new classification of abduction. In Section 3.2, we apply this classification to the researches of abduction in computer science.

This chapter is based on the papers [Hir93a, Hir93b].

3.1 Five Types of Abduction

Various researches about abduction in computer science and computational logic are also related to at least one type of explanatory hypotheses. Hence, in this section, we introduce the classification which is based on three types of explanatory hypotheses. Note that, in the researches of abduction in computer science, a *background theory* is assumed in order to explain a surprising fact. First, by the definition of a background theory, we classify abduction in computer science into two types, abduction of a rule and of a theory. Here, a rule means an element of a background theory, while a theory means a background theory itself. Abduction of a rule is called *rule-based abduction*, while that of a theory *theory-based abduction*.

In rule-based abduction, a hypothesis A in a syllogism is a set of atoms. Then, for a surprising fact C and a hypothesis A, we denote rule-based abduction by $A \to C$ in a syllogism. Hence, rule-based abduction is depicted by the following syllogism:

$$\frac{C \quad A \to C}{A}.$$

On the other hand, in theory-based abduction, a hypothesis A in a syllogism is a theory. Then, for a surprising fact C and a hypothesis A, we denote theory-based abduction by $A \vdash C$ in a syllogism. Hence, theory-based abduction is also depicted by the following syllogism:

$$\frac{C \quad A \vdash C}{A}.$$

For rule-based abduction, it is our purpose to obtain a rule $A \to C$ and a hypothesis A to explain a surprising fact C. In order to capture the properties of rule-based abduction, we apply three types of explanatory hypotheses to rule-based abduction.

According to Peirce, abduction begins with an observation of a surprising fact [Pei65, Uey79, Yon82]. Hence, in rule-based abduction, a surprising fact must be *surprising* with respect to the background theory given in advance. Let P be a background theory, A be a set of atoms A, and C be a surprising fact with respect to P.

(1) The first type is an abduction that assumes existence of the rules in a given background theory. In this type, for a surprising fact C, we select a rule $C \leftarrow A$ in a background theory P, and propose a hypothesis A in P such that C is explained by the selected rule $C \leftarrow A$ and the hypothesis A. We call this type of abduction *rule-selecting abduction*. An inference schema of rule-selecting abduction is depicted by the following syllogism.

$$\frac{\begin{array}{l} C\text{: } \textit{surprising fact} \text{ wrt } P \\ \textit{Select} \text{ a rule } C \leftarrow A \text{ in } P \end{array}}{\textit{Propose} \text{ a hypothesis } A \text{ in } P}$$

(2) The second type is an abduction that assumes existence of the rules in a background theory other than a given one. In this type, we assume that the set of background theories is given in advance. Then, for a surprising fact C, we find a rule $C \leftarrow A$ in a background theory P', possibly not P, and propose a hypothesis A. We call this type *rule-finding abduction*. An inference schema of rule-finding abduction is depicted by the following syllogism.

$$\frac{\begin{array}{l} C\text{: } \textit{surprising fact} \text{ wrt } P \\ \textit{Find} \text{ a rule } C \leftarrow A \text{ in } P' \ (\neq P) \end{array}}{\textit{Propose} \text{ a hypothesis } A \text{ in } P}$$

(3) The third type is an abduction that cannot assume existence of the rules in any background theory. In this type, for a surprising fact C, we newly generate a rule $C \leftarrow A$ in a background theory P, and propose a hypothesis A in P such that C is explained by the generated rule $C \leftarrow A$ and the hypothesis A. We call this type *rule-generating abduction*. An inference schema of rule-generating abduction is depicted by the following syllogism.

$$\frac{C\text{: } \textit{surprising fact} \text{ wrt } P}{\begin{array}{l} \textit{Generate} \text{ a rule } C \leftarrow A \text{ in } P \\ \textit{Propose} \text{ a hypothesis } A \text{ in } P \end{array}}$$

If we apply the above three types to abduction for logic programming, then the syllogisms of rule-based abduction are illustrated as in Figure 3.1, where *sf* stands for a surprising fact. In rule-based abduction for logic programming, a surprising fact C

(1) rule-selecting	$\dfrac{P \not\vdash C \ (C : sf\ wrt\ P) \quad C \leftarrow A\ in\ P}{A\ (set\ of\ atoms)}$
(2) rule-finding	$\dfrac{P \not\vdash C \ (C : sf\ wrt\ P) \quad C \leftarrow A\ in\ P'}{A\ (set\ of\ atoms)}$
(3) rule-generating	$\dfrac{P \not\vdash C \ (C : sf\ wrt\ P)}{C \leftarrow A\ in\ P \quad A\ (set\ of\ atoms)}$

Figure 3.1: Rule-based abduction for logic programming

with respect to a program P is regarded as a ground atom such that $P \not\vdash C$. In other words, C is explained by P if C is provable in P. Note that, in the syllogisms of rule-selecting and rule-generating abduction, $P \cup A \vdash C$ holds for a proposed hypothesis A and a program P, by regarding A as the set $\{A\}$ of atoms.

For theory-based abduction, it is our purpose to obtain a theory A to explain a surprising fact C. In order to capture the properties of theory-based abduction, we also apply three types of explanatory hypotheses to theory-based abduction.

Let B be a background theory and C be a surprising fact with respect to B.

(4) The first type is an abduction that assumes existence of the theory in a given set of background theories. Note here that the set of background theories are given in advance. In this type, we can select and propose a theory A which makes the surprising fact C true. We call this type of abduction *theory-selecting abduction.* An inference schema of theory-selecting abduction is depicted by the following syllogism.

$$\frac{\begin{array}{l} C\text{: } \textit{surprising fact} \text{ wrt } B \\ \textit{Select} \text{ a theory } A \text{ such that } A \text{ makes } C \text{ true} \end{array}}{\textit{Propose} \text{ a hypothesis } A}$$

(5) The second type of abduction which we could call *theory-finding abduction* is the same as the theory-selecting abduction above, because we must assume that there exists a set of background theories.

(6) The third type is an abduction that cannot assume existence of the theory in the set of background theories. In this type, we generate and propose a theory A which makes the surprising fact C true. We call this type *rule-generating abduction.* An

(4) theory-selecting	$\dfrac{B \not\vdash C \ (C : sf\ wrt\ B) \quad A \vdash C}{A\ (theory)}$
(6) theory-generating	$\dfrac{B \not\vdash C \ (C : sf\ wrt\ B)}{A \vdash C \quad A\ (theory)}$

Figure 3.2: Theory-based abduction for logic programming

inference schema of theory-generating abduction is depicted by the following syllogism.

$$\frac{C\text{: } \textit{surprising fact} \text{ wrt } B}{\begin{array}{l}\textit{Generate} \text{ a theory } A \text{ such that } A \text{ makes } C \text{ true} \\ \textit{Propose} \text{ a hypothesis } A\end{array}}$$

If we apply the above two types to abduction for logic programming, then the syllogisms of theory-based abduction are illustrated as Figure 3.2. In theory-based abduction for logic programming, a surprising fact C with respect to a program B is also regarded as a ground atom such that $B \not\vdash C$.

3.2 Application to Previous Researches

Now we examine the various researches on abduction so far developed and show that all of them can be placed in our classification.

(1) *Rule-selecting abduction*: Abductive logic programming [Dun91, EK89, KM90, KKT92] is a kind of rule-selecting abduction. It is different from Peirce's abduction in the following viewpoint: Peirce has asserted that abduction begins with an observation of a surprising fact [Pei65, Uey79, Yon82]. However, in their works on abductive logic programming, Eshghi and Kowalski [EK89], Kakas and Mancarella [KM90], Dung [Dun91], and Kakas *et al.* [KKT92] have asserted that a hypothesis to explain the observed fact can be formed in the abductive framework. Kakas and Mancarella [KM90] have also asserted that the abductive framework is vacuous and ill-defined if there exist no models to explain the observation. Therefore, they cannot deal with the surprising fact in the sense of Peirce's abduction.

Abduction for explanation-based generalization by Genest *et al.* [GMP90] is a kind of rule-selecting abduction. However, it depends on heuristics which makes the surprising fact *surprising*.

Abduction for natural language interpretation by Hobbs *et al.* [HSME88] and Stickel [Sti91] is a kind of rule-selecting abduction. In the formulation of Hobbs *et al.* [HSME88], they have dealt with first order formulas with costs as the logical forms of abduction. On the other hand, Stickel [Sti91] has dealt with function-free definite programs as the logical forms of abduction.

Concerning expert system, abduction for diagnosis problem by Cox and Pietrzykowski [CP87] is a kind of rule-selecting abduction. They have introduced the concept of a *cause*, and dealt with resolutions for computing fundamental causes. Furthermore, the research of Pirri and Pizzuti [PP90] can be regarded as the diagnosis problem in abductive logic programming.

(2) *Rule-finding abduction*: Duval's abduction [Duv91] is a kind of rule-finding abduction. Duval [Duv91] has dealt with the following abduction for explanation-based generalization: Let D be a domain theory, $A \leftarrow B \wedge C$ be a rule in D, and C be a surprising fact with respect to D. Then, his system finds $C' \in D$ which is analogous to C, and adds a rule $A \leftarrow B \wedge C'$ to D. He called such adding rule abduction.

Thagard's *analogical abduction* [Tha88] is also a kind of rule-finding abduction.

(3) *Rule-generating abduction*: The constructive operators such as V and W operators [Mug92, MB88, Lin89, LU89] in inductive logic programming are a kind of rule-generating abduction. Concretely, the constructive operators generate definite clauses from a finite surprising facts, called *examples*. Hence, examples are regarded as surprising facts in Peirce's sense.

(4) *Theory-selecting abduction*: Poole's Theorist [Poo88] and hypothesis-based reasoning [Kun87] are theory-selecting abduction, where the candidates of a hypothesis are given in advance. The main part of their researches is how to select a suitable hypothesis from the candidates.

As the extensions of Poole's research, there exists the research of abduction for a model of *belief* by Levesque [Lev89] and Selman and Levesque [SL90]. Their frameworks of abduction depend on a model of belief, which is a kind of modal logic. It is their purpose to construct a model of belief for abduction, not to find an explanation. However, we can regard their abduction as the extension of Poole's abduction [Ino92].

Konolige [Kon92] has investigated the relationship between abduction and the diagnosis problem by introducing a *causal theory*. We can regard it as the extension of Poole's abduction.

Bylander *et al.* [BATJ91] have introduce the another framework of abduction for propositional logic. They have extended the symbol "$\rightarrow$" of logical implication to the causal relation, and analyze the computational complexity of abduction and the diagnosis problem. We can also regard it as the extension of Poole's abduction for propositional logic.

(6) *Theory-generating abduction*: Shapiro's model inference system [Sha81] and inductive logic programming [Mug92, MB88, Lin89, LU89] are a kind of theory-generating abduction. Model inference system and inductive logic programming *inductively* make definite programs. By the above systems, the definite programs are constructed by surprising facts, if we regard examples as surprising facts.

It is the main purpose of rule-selecting and theory-selecting abduction to *find* a hypothesis to explain a surprising fact. Then, they are related to nonmonotonic logic, the diagnosis problem in expert system, and knowledge representation. On the other hand, it is the main purpose of rule-generating and theory-generating abduction to *obtain* a hypothesis to explain a surprising fact. Then, they are related to inductive logic programming, machine learning, and knowledge acquisition. It is the main purpose of rule-finding abduction to *find* a hypothesis in the given set of background theories. Then, it is related to analogical reasoning.

Theory-based abduction is considered as the extensions of rule-based abduction, and rule-based abduction is an essential abduction. Hence, in the following chapters, we investigate each types of rule-based abduction for logic programming.

Chapter 4

Rule-Selecting Abduction

"It is not really difficult to construct a series of inferences, each dependent upon its predecessor and each simple in itself."

— *'The Adventure of the Dancing Men'*

"The Return of Sherlock Holmes"

Let P be a definite program. Throughout this thesis, a *surprising fact* C with respect to P is regarded as a ground atom such that $P \not\vdash C$. Note here that P is given before C is given. The *rule-selecting abduction* is a type of abduction which selects a rule in P and proposes a hypothesis to explain the surprising fact C. An inference schema of rule-selecting abduction is described by the following three steps:

1. A surprising fact C is observed.

2. A rule $C \leftarrow A$ is *selected* in P.

3. A hypothesis A is proposed.

For a surprising fact C, we regard the above inference schema as the following one by identifying a hypothesis A with the set $\{A\}$ of atoms:

1. A ground atom C such that $P \not\vdash C$ is given.

2. A rule $C' \leftarrow A'_1, \cdots, A'_n$ is *selected* in P, where $C'\theta = C$ and $A'_i\theta = A_i$ $(1 \leq i \leq n)$.

3. A hypothesis $\{A_1, \cdots, A_n\}$ is proposed. Then, $P \cup \{A_1, \cdots, A_n\} \vdash C$.

Note that the above inference schema is similar to abductive framework [Dun91, EK89, KM90, Poo88]. However, we are not interested in how semantics is suggested in the abductive framework, but we are interested in how a hypothesis is proposed. Also we are interested in abduction for definite program.

In this chapter, we investigate rule-selecting abduction for logic programming. In Section 4.1, we discuss the termination of rule-selecting abduction. We introduce the *head-reducing* programs, and show that all the derivations for a head-reducing program and a surprising fact are finite. In Section 4.2, we formulate abduction for default logic. We show that if there exists a hypothesis which explains a surprising fact, then there also exists the extension of a given default theory, which includes the surprising fact. In Section 4.3, we extend the concept of head-reducingness to that of *breadth-first head-reducing* programs, and the rule-selecting abduction to the *breadth-first rule-selecting abduction*. We also show that there exists a finite derivation for a breadth-first head-reducing program and a surprising fact. In Section 4.4, we realize the above three types of rule-selecting abduction as Prolog programs.

This chapter is based on the papers [Hir93a, Hir93b].

4.1 Rule-Selecting Abduction for Logic Programming

Let us consider the following definite program P_1:

$$P_1 = \left\{ \begin{array}{l} C_1 : \ p(f(X), f(Y)) \leftarrow p(X,Y), q(X), r(Y) \\ C_2 : \ q(f(X)) \leftarrow q(X) \\ C_3 : \ r(f(X)) \leftarrow r(X) \\ C_4 : \ r(a) \end{array} \right\}.$$

The least Herbrand model $M(P_1)$ of P_1 is $\{r(a), r(f(a)), r(f^2(a)), \cdots\}$. There exists no atom α with the predicate symbol p in $M(P_1)$. Also a ground atom $p(f(a), f^2(b))$ is given as a surprising fact with respect to P_1, that is, $P_1 \not\vdash p(f(a), f^2(b))$. Then, rule-selecting abduction for P_1 is the following process:

1. If we select no rules, then we obtain the following hypothesis H_1 by rule-selecting abduction for P_1:

$$H_1 = \{p(f(a), f^2(b))\}.$$

2. If we select the rule C_1, then we obtain the following hypothesis H_2 by rule-selecting abduction for P_1:

$$H_2 = \{p(a, f(b)), q(a), r(f(b))\}.$$

3. If we select the rules C_1 and C_3, then we obtain the following hypothesis H_3 by rule-selecting abduction for P_1:

$$H_3 = \{p(a, f(b)), q(a), r(b)\}.$$

Note that, for each H_i $(1 \leq i \leq 3)$, $P_1 \cup H_i \vdash p(f(a), f^2(b))$.

Hence, for a surprising fact α, rule-selecting abduction for P is the proposal of hypotheses H such that $P \cup H \vdash \alpha$.

The rule-selecting abduction can be realized in the following Prolog program `rs_abd`, which is a variant of partial evaluation in van Harmelen and Bundy [vHB88].

```
rs_abd(Goal,Leaves) :- clause(Goal,Clause),rs_abd(Clause,Leaves).
rs_abd((Goal1,Goal2),(Leaf1,Leaf2)) :-
        !,rs_abd(Goal1,Leaf1),rs_abd(Goal2,Leaf2).
rs_abd(Leaf,Leaf) :- !.
```

Since abduction is the first stage of scientific inquiry, we should consider the process of abduction which terminates. If abduction terminates, then we can automatically propose some hypotheses. Hence, in this section, we discuss the termination of rule-selecting abduction. It is our purpose to identify the class of definite programs for which all the processes of rule-selecting abduction terminate.

First, we introduce the following definitions.

Definition 4.1 *Let P be a definite program and p be a predicate symbol. Then, a recursive definition of p for P, denoted by $rec(P, p)$, is a definition clause of p constructed by the following procedure:*

1. *Select a clause in P whose head has the predicate p, and let it be $rec(P, p)$.*

2. *For* $rec(P,p) = A \leftarrow B_1, \cdots, B_l, \cdots, B_n$*, if there exists a clause* $E \leftarrow F_1, \cdots, F_m$ *such that* $B_l\theta = E\theta$ *for a substitution* θ*, and* $pred(B_l)(= pred(E)) \neq p$*, then eliminate the clause* $E \leftarrow F_1, \cdots, F_m$ *from* P*, and put*

$$rec(P,p){=}(A \leftarrow B_1, \cdots, B_{l-1}, F_1, \cdots, F_m, B_{l+1}, \cdots, B_n)\theta.$$

3. *Repeat 2 until it cannot be applied.*

A recursive program of p *for* P*, denoted by* $RP(P,p)$*, is a program consisting of a recursive definition* $rec(P,p)$ *and the applied clauses in constructing* $rec(P,p)$*.*

For a definite program P and a predicate symbol p, $rec(P,p)$ and $RP(P,p)$ are not unique in general.

Example 4.1 *Let* P_2, P_3 *and* P_4 *be the following definite programs:*

$$P_2 = \left\{ \begin{array}{l} p(f(X)) \leftarrow p(X), q(X,Y) \\ q(f(X),f(Y)) \leftarrow q(f(X),Y) \end{array} \right\},$$

$$P_3 = \left\{ \begin{array}{l} p(f(X)) \leftarrow p(X), q(X,Y) \\ q(f(X),f(Y)) \leftarrow q(X,f(Y)) \end{array} \right\},$$

$$P_4 = \left\{ \begin{array}{l} p(f(X)) \leftarrow p(f^2(X)), q(X,Y) \\ q(f(X),f(Y)) \leftarrow q(f(X),Y) \end{array} \right\}.$$

Then, the recursive definitions $rec(P_i,p)$ *and the recursive programs* $RP(P_i,p)$ $(2 \leq i \leq 4)$ *are as follows:*

$$\begin{aligned} rec(P_2,p) &= p(f^2(X)) \leftarrow p(f(X)), q(f(X),Y), \\ rec(P_3,p) &= p(f^2(X)) \leftarrow p(f(X)), q(X,f(Y)), \\ rec(P_4,p) &= p(f^2(X)) \leftarrow p(f^3(X)), q(f(X),Y), \end{aligned}$$

$$RP(P_2,p) = \left\{ \begin{array}{l} p(f^2(X)) \leftarrow p(f(X)), q(f(X),Y) \\ q(f(X),f(Y)) \leftarrow q(f(X),Y) \end{array} \right\},$$

$$RP(P_3,p) = \left\{ \begin{array}{l} p(f^2(X)) \leftarrow p(f(X)), q(X,f(Y)) \\ q(f(X),f(Y)) \leftarrow q(X,f(Y)) \end{array} \right\},$$

$$RP(P_4,p) = \left\{ \begin{array}{l} p(f^2(X)) \leftarrow p(f^3(X)), q(f(X),Y) \\ q(f(X),f(Y)) \leftarrow q(f(X),Y) \end{array} \right\}.$$

On the other hand, let P_5 be the following definite program:

$$P_5 = \left\{ \begin{array}{l} p(f(X)) \leftarrow p(X), q(X,Y) \\ p(f(X)) \leftarrow p(f^2(X)), q(X,Y) \\ q(f(X), f(Y)) \leftarrow q(X,Y) \end{array} \right\}.$$

Then, there exist the following two recursive definitions $rec(P_5, p)$:

$$p(f^2(X)) \leftarrow p(f(X)), q(f(X), Y),$$

$$p(f^2(X)) \leftarrow p(f^3(X)), q(f(X), Y).$$

There also exist the following two recursive programs $RP(P_5, p)$ corresponding to the above recursive definitions $rec(P_5, p)$:

$$\left\{ \begin{array}{l} p(f^2(X)) \leftarrow p(f(X)), q(f(X), Y) \\ q(f(X), f(Y)) \leftarrow q(X,Y) \end{array} \right\},$$

$$\left\{ \begin{array}{l} p(f^2(X)) \leftarrow p(f^3(X)), q(f(X), Y) \\ q(f(X), f(Y)) \leftarrow q(X,Y) \end{array} \right\}.$$

A clause $p(t_1, \cdots, t_n) \leftarrow B_1, \cdots, B_m$ is said to be *p-reducing with respect to the i-th argument* if $|t_i\theta| > |s_i^l\theta|$ for any substitution θ and for any index l such that $pred(B_l) = p$, where s_i^l is the i-th argument's term of B_l. A p-reducing clause with respect to some argument is called a *p-reducing clause* simply. These definitions are the extensions of *reducing* and *weakly reducing* programs by Yamamoto [Yam92].

Example 4.2 *In Example 4.1, the definition clause of p in P_2 and P_3 are p-reducing. The recursive programs $rec(P_2, p)$ and $rec(P_3, p)$ are also p-reducing. On the other hand, the definition clause of p in P_4 and the recursive definition $rec(P_4, p)$ is not p-reducing.*

For a p-reducing clause, the following lemma holds.

Lemma 4.1 *Let C be a p-reducing clause $p(t_1, \cdots, t_n) \leftarrow B_1, \cdots, B_m$ and $p(s_1, \cdots, s_n)$ be a ground atom. Then, all the SLD-derivations of $\{C\} \cup \{\leftarrow p(s_1, \cdots, s_n)\}$ are finite.*

Proof. Suppose that C is p-reducing with respect to the i-th argument.

If $p(t_1, \cdots, t_n)$ and $p(s_1, \cdots, s_n)$ are not unifiable, then the derivation of $\{C\} \cup \{\leftarrow p(s_1, \cdots, s_n)\}$ is finitely failed.

Suppose that $p(t_1, \cdots, t_n)$ and $p(s_1, \cdots, s_n)$ are unifiable. Since $p(s_1, \cdots, s_n)$ is ground, there exists a unifier θ for $p(t_1, \cdots, t_n)$ and $p(s_1, \cdots, s_n)$ such that $p(t_1, \cdots, t_n)\theta = p(s_1, \cdots, s_n)$. If $B_j\theta$ is the selected atom of the goal $\leftarrow B_1\theta, \cdots, B_m\theta$, and $B_j\theta$ and $p(t_1, \cdots, t_n)$ are not unifiable, then the derivation of $\{C\} \cup \{\leftarrow B_1\theta, \cdots, B_m\theta\}$ is finitely failed. Otherwise, suppose that $B_l\theta$ is the selected atom of the goal $\leftarrow B_1\theta, \cdots, B_m\theta$. Also suppose that $B_l\theta$ and $p(t_1, \cdots, t_n)$ are unifiable. Note that $s_i^l\theta$ is ground, where s_i^l is the i-th argument's term in B_l. By the definition of a p-reducing clause,

$$|s_i^l\theta| < |t_i\theta| = |s_i|.$$

Furthermore, if $p(t_1, \cdots, t_n)$ and $B_l\theta$ are unifiable, then, for the derivation of $\{C\} \cup \{\leftarrow B_l\theta\}$, there exists a unifier σ for $B_l\theta$ and $p(t_1, \cdots, t_n)$ such that $B_l\theta\sigma = p(t_1, \cdots, t_n)\sigma$. Then,

$$|s_i^l\sigma| < |t_i\sigma| = |s_i^l\theta\sigma| = |s_i^l\theta| < |s_i|.$$

Hence, the longest derivation of $\{C\} \cup \{\leftarrow B_1\theta, \cdots, B_m\theta\}$ is constructed in the following way: Let G_0 be the initial goal $\leftarrow B_1\theta, \cdots, B_m\theta$, and G_i be the i-th resolvent. Then, by selecting each atom $B_l\theta$ in the derivation, we can obtain the following resolvent G_m of the derivation:

$$\leftarrow (B_1\theta_1, \cdots, B_m\theta_1), (B_1\theta_2, \cdots, B_m\theta_2), \cdots, (B_1\theta_m, \cdots, B_m\theta_m).$$

For any $B_l\theta_k (1 \leq l, k \leq m)$, $|s_i^l\theta_k| < |s_i|$. Furthermore, by selecting each atom $B_l\theta_k$ in the derivation, we can also obtain the following resolvent G_{m+m^2} of the derivation:

$$\leftarrow ((B_1\theta'_1, \cdots, B_m\theta'_1), \cdots, (B_1\theta'_m, \cdots, B_m\theta'_m)), \cdots, (\cdots, (B_1\theta'_{m^2}, \cdots, B_m\theta'_{m^2})).$$

For any $B_l\theta'_k (1 \leq l \leq m, 1 \leq k \leq m^2)$, $|s_i^l\theta'_k| < |s_i| - 1$.

Hence, the length of the derivation of $\{C\} \cup \{\leftarrow B_1\theta, \cdots, B_m\theta\}$ is at most $\sum_{k=1}^{|s_i|} m^k$, and the length of the derivation of $\{C\} \cup \{\leftarrow p(s_1, \cdots, s_n)\}$ is at most $1 + \sum_{k=1}^{|s_i|} m^k$. ∎

Whether or not the process of rule-selecting abduction for a definite program terminates is characterized as the following concept of head-reducing.

Definition 4.2 *Let $rec(P,p)$ be a recursive definition $p(t_1,\cdots,t_n) \leftarrow B_1,\cdots,B_m$ of p for P. Then, a recursive program $RP(P,p)$ is called head-reducing if it satisfies the following conditions:*

1. *If there exists an index k such that $B_k = p(s_1^k,\cdots,s_n^k)$, then*
   - (a) *there exists an index j such that $|t_j\theta| > |s_j^k\theta|$ for any such k and for any substitution θ, and*
   - (b) *any atom B_l such that $B_l = q_l(u_1^l,\cdots,u_{n_l}^l)$ $(p \neq q_l)$ satisfies one of the following conditions:*
     - (b-i) *there exists the i-th argument's term u_i^l in B_l which is constructed by the variables appearing in t_j, and the definition clause of q_l is q_l-reducing with respect to the i-th argument, or*
     - (b-ii) *the definition clause of q_l is not included in $RP(P,p)$.*
2. *Otherwise, any $B_l = q_l(u_1^l,\cdots,u_{n_l}^l)$ satisfies one of the following conditions:*
   - (c) *there exists the i-th argument's term u_i^l in B_l which is constructed by the variables appearing in all arguments' terms $t_1,\cdots,t_n$ in $p(t_1,\cdots,t_n)$, and the definition clause of q_l is q_l-reducing with respect to i-th argument, or*
   - (d) *the definition clause of q_l is not included in $RP(P,p)$.*

Furthermore, P is head-reducing with respect to the predicate p if any recursive program $RP(P,p)$ of p for P is head-reducing.

Example 4.3 *In Example 4.1, P_3 is head-reducing with respect to p. On the other hand, P_2 is not head-reducing with respect to p, because the recursive program $RP(P_2,p)$ does not satisfy the condition (b-i) of Definition 4.2. Also P_4 is not head-reducing, because the recursive program $RP(P_4,p)$ does not satisfy the condition (a) of Definition 4.2.*

For P_5, the first recursive program is head-reducing, but the second recursive program is not head-reducing, because it does not satisfy the condition (a) of Definition 4.2. Then, P_5 is not head-reducing with respect to p.

Furthermore, the following typical Prolog programs [SS86, SS94] are head-reducing with respect to the predicate of the head.

$$\begin{aligned}&\{\ member(X,[W|Y]) \leftarrow member(X,Y)\ \},\\ &\{\ append([W|X],Y,[W|Z]) \leftarrow append(X,Y,Z)\ \},\\ &\{\ concat(X,[W|Y],[W|Z]) \leftarrow concat(X,Y,Z)\ \}.\end{aligned}$$

On the termination of rule-selecting abduction, the following theorem holds.

Theorem 4.1 *Let P be a definite program and p be a predicate symbol. If P is head-reducing with respect to p, then all the SLD-derivations of $P \cup \{\leftarrow p(s_1, \cdots, s_n)\}$ are finite.*

Proof. The result is proven by mathematical induction on the number of clauses in P. If the number is 1, then Lemma 4.1 implies the result.

Next suppose that all the derivations of $P \cup \{\leftarrow p(s_1, \cdots, s_n)\}$ are finite for $P = \{C_1, \cdots, C_k\}$, and let P' be $P \cup \{C_{k+1}\}$. Also suppose that any $RP(P', p)$ is head-reducing. Let C_{k+1} be the following clause:

$$C_{k+1}:\ p_{k+1}(t_1, \cdots, t_{n_{k+1}}) \leftarrow B_1, \cdots, B_l.$$

If the predicate symbol p_{k+1} does not occur in P, then all the input clauses of the derivation of $P \cup \{\leftarrow p(s_1, \cdots, s_n)\}$ do not include the clause C_{k+1}. Thus, the derivation of $P' \cup \{\leftarrow p(s_1, \cdots, s_n)\}$ is equal to one of $P \cup \{\leftarrow p(s_1, \cdots, s_n)\}$. Consequently, the derivation of $P' \cup \{\leftarrow p(s_1, \cdots, s_n)\}$ is finite by the induction hypothesis.

If the predicate symbol p_{k+1} occurs in the clause C_i of P, then there exists a clause C_i in P, and one of the following cases holds:

1. p_{k+1} occurs in the body of C_i, or

2. p_{k+1} occurs in the head of C_i.

If any $RP(P,p)$ does not include the clause C_i, then all the input clauses of the derivation of $P \cup \{\leftarrow p(s_1, \cdots, s_n)\}$ do not include C_i. Thus, the derivation of $P' \cup \{\leftarrow p(s_1, \cdots, s_n)\}$ is equal to one of $P \cup \{\leftarrow p(s_1, \cdots, s_n)\}$. In both cases, which are corresponding to the condition (b-ii) or (d) of Definition 4.2, the derivation of $P \cup \{\leftarrow p(s_1, \cdots, s_n)\}$ is finite by the induction hypothesis.

Thus, suppose that some $RP(P,p)$ includes C_i.

1. Suppose that p_{k+1} occurs in the body of C_i. Let C_i be the following clause:

$$C_i: \; p_i(u_1, \cdots, u_{n_i}) \leftarrow D_1, \cdots, D_j, \cdots, D_m,$$

   where $pred(D_j) = p_{k+1}$. By the induction hypothesis, if there exists an infinite derivation of $P' \cup \{\leftarrow p(s_1, \cdots, s_n)\}$, and $D_j\lambda$ and $p_{k+1}(t_1, \cdots, t_{n_{k+1}})$ are unifiable, where λ is a substitution, then the derivation of $P' \cup \{\leftarrow D_j\lambda\}$ is infinite. However, we can show that all the derivations of $P' \cup \{\leftarrow D_j\lambda\}$ are finite by the following discussion.

   Suppose that the derivation of $P' \cup \{\leftarrow D_j\lambda\}$ is infinite. Then, the derivation of $P' \cup \{\leftarrow B_1\sigma, \cdots, B_l\sigma\}$ is also infinite, where σ is a unifier of $D_j\lambda$ and $p_{k+1}(t_1, \cdots, t_{n_{k+1}})$. For any $B_i\sigma$, consider the predicate symbol $pred(B_i)$.

   (a) Suppose that for any i, $pred(B_i)$ does not occur in P or $pred(B_i)$ is p_{k+1}. Then, all the input clauses of the derivation of $P' \cup \{\leftarrow D_j\lambda\}$ include only the clause C_{k+1}. Since $RP(P,p)$ is head-reducing, then $D_j\lambda$ has the argument's term which is ground, and C_{k+1} is p_{k+1}-reducing with respect to this argument by the condition (b-i) or (c) of Definition 4.2. By Lemma 4.1, the derivation of $P' \cup \{\leftarrow D_j\lambda\}$ is finite.

   (b) Suppose that there exists an index i such that $pred(B_i)$ occurs in P and $pred(B_i) \neq p_{k+1}$. For any such i, one of the following two cases holds.

      i. Suppose that there exists a clause C_j such that $pred(B_i) = pred(head(C_j))$ and some $RP(P,p)$ includes C_j. If $head(C_j)$ and B_i are not unifiable, then the derivation of $P' \cup \{\leftarrow B_i\sigma\}$ is finite, because all the input clauses of the derivation of $P' \cup \{\leftarrow B_i\sigma\}$ do not include C_j. Otherwise, suppose that $head(C_j)$ and B_i are unifiable. Since any $RP(P', p)$ is head-reducing and includes C_j, C_j is $pred(B_i)$-reducing with respect to some argument. Note that this argument's term of $B_i\sigma$ is ground. By Lemma 4.1, the derivation of $P' \cup \{\leftarrow B_i\sigma\}$ is finite.

ii. Suppose that there exists a clause C_j such that $pred(B_i) = pred(head(C_j))$ and any $RP(P, p)$ does not include C_j. Then, all the input clauses of the derivation of $P' \cup \{\leftarrow B_i\sigma\}$ do not include any clause of any $RP(P, p)$. By the case (a), the SLD-derivation of $P' \cup \{\leftarrow B_i\sigma\}$ is finite.

By the cases (a) and (b), there exist no infinite SLD-derivations of $P' \cup \{\leftarrow B_1\sigma, \cdots, B_m\sigma\}$.

2. Suppose that p_{k+1} occurs in the head of C_i. Let C_i be the following clause:

$$A \leftarrow D_1, \cdots, D_m,$$

where $pred(A) = p_{k+1}$. Let $\leftarrow G_1\tau, \cdots, G_j\tau, \cdots, G_l\tau$ be the resolvent whose input clause is $A \leftarrow D_1, \cdots, D_m$ in the SLD-derivation of $P \cup \{\leftarrow p(s_1, \cdots, s_n)\}$. By the induction hypothesis, if there exists an infinite SLD-derivation of $P' \cup \{\leftarrow p(s_1, \cdots, s_n)\}$, and $G_j\tau$ and A are unifiable, then the SLD-derivation of $P' \cup \{\leftarrow G_j\tau\}$ is infinite. Then, the SLD-derivation of $P' \cup \{\leftarrow B_1\sigma, \cdots, B_l\sigma\}$ is infinite, where σ is a unifier of $G_j\tau$ and A. However, by the same proof as the case 1, there exist no infinite SLD-derivations of $P' \cup \{\leftarrow B_1\sigma, \cdots, B_l\sigma\}$.

Hence, all the derivations of $P' \cup \{\leftarrow D_j\lambda\}$ are finite. Therefore, all the SLD-derivations of $P' \cup \{\leftarrow p(s_1, \cdots, s_n)\}$ are also finite. ∎

4.2 Rule-Selecting Abduction for Default Logic

In general, abduction is deeply related to *nonmonotonic reasoning*, because both abduction and nonmonotonic reasoning are a kind of plausible inference. There exist various researches for nonmonotonic logic; *nonmonotonic modal logic* [McD82, MD80], *autoepistemic logic* [Moo85], *belief logic* [Lev89, SL90], *default logic* [Reit80, Poo88], and *circumscription* [McC80, McC86, Lif85, Hir92, Hir94a]. In this section, we compare rule-selecting abduction with Reiter's default logic [Reit80].

Poole [Poo88, Ino92] has defined an abductive framework and discussed the relationship between it and Reiter's default logic [Reit80]. We can describe the result of Poole [Poo88] in term of our abduction in the following way:

Theorem 4.2 (Poole [Poo88]) *Suppose that $P \not\vdash \alpha$. Then, there exists a hypothesis H such that $P \cup H \vdash \alpha$ if and only if there exists an extension E of default theory (D_H, P) such that $\alpha \in E$, where*

$$D_H = \left\{ \frac{: w(X)}{w(X)} \;\middle|\; w(X) \in H \right\}.$$

The above theorem means that there exists an extension which includes the explainable fact α, while it does not mean how a hypothesis is constructed when a surprising fact is observed. In other words, Poole's abduction is an *abduction for logic programming in term of default logic*, but not an *abduction for default logic itself*. Hence, we study abduction for default logic. Here, we deal with *function-free closed normal default theories whose conclusions are positive atoms*, because there exists an extension for such a default theory by Theorem 2.2. Also we deal with a definite program and an *integrity constraint* IC as negative information. The integrity constraint IC is of the form $IC = \bigvee_{i=1}^{n} (\leftarrow G_i)$, where $G_i = {G_1}^i \wedge \cdots \wedge {G_{n_i}}^i$ and ${G_j}^i$ is an atom.

The following lemma is shown by the definition of integrity constraints and the monotonicity of definite programs.

Lemma 4.2 *Let P be a definite program and IC be an integrity constraint. If $P \cup IC$ is consistent, then the least Herbrand model $M(P)$ of P is the model of $P \cup IC$ under the closed-world assumption [Llo87].*

In Section 4.1, a surprising fact is defined as a ground atom α such that $P \not\vdash \alpha$, and rule-selecting abduction is the proposal of a hypothesis H such that $P \cup H \vdash \alpha$. In default logic, a *surprising fact* is considered as a ground atom α which is not included in any extension of a given default theory (D, P). For the surprising fact α, it is our purpose to propose the new default theory $(D, P \cup H)$, instead of (D, P), such that there exists an extension which includes α. Then, we regard such an H as a *hypothesis* for rule-selecting abduction for default logic.

In order to define a surprising fact and a hypothesis for default logic, first we introduce the transformation from the following set of default rules D

$$D = \left\{ \frac{\alpha_i(\overline{X}) : w_i(\overline{X})}{w_i(\overline{X})} \;\middle|\; 1 \leq i \leq l \right\} \quad (\overline{X} : \text{tuple of variables})$$

to the following definite program P_D:

$$P_D = \left\{ w_i(\overline{X}) \leftarrow \alpha_i(\overline{X}) \;\middle|\; \frac{\alpha_i(\overline{X}) : w_i(\overline{X})}{w_i(\overline{X})} \in D, 1 \leq i \leq l \right\}.$$

For such P_D, the following lemma holds.

Lemma 4.3 *Let (D, P) be a closed default theory and α be a ground atom. If $P \cup P_D \not\models \alpha$, then there exists no extension of (D, P) which includes α.*

Proof. Let E be an extension of (D, P). By Theorem 2.1, E is constructed in the following way:

$$\begin{aligned} E_0 &= P, \\ E_{i+1} &= Th(E_i) \cup \left\{ w \;\middle|\; \frac{\beta : w}{w} \in D, \beta \in E_i, \neg w \notin E \right\}, \\ E &= \bigcup_{i \geq 0} E_i. \end{aligned}$$

Since $P_D = \left\{ w \leftarrow \beta \;\middle|\; \frac{\beta : w}{w} \in D \right\}$, $E_{i+1} \subseteq Th(E_i) \cup M(P_D)$ for any i. Then, $E \subseteq M(P \cup P_D)$. Hence, if $P \cup P_D \not\models \alpha$, that is, $\alpha \notin M(P \cup P_D)$, then $\alpha \notin E$. ∎

By the above lemma, we define *a surprising fact for a default theory* (D, P) as a ground atom α such that $P \cup P_D \not\models \alpha$.

Furthermore, a hypothesis for default logic is defined as follows:

Definition 4.3 *Let α be a surprising fact and $(D, P \cup IC)$ be a closed normal default theory, where P is a definite program and IC is an integrity constraint. Then, H is a hypothesis of α for a default theory $(D, P \cup IC)$ if it satisfies one of the following conditions:*

1. $P \cup H \vdash \alpha$ and $P \cup H \cup IC$ is consistent, or

2. $P \cup H \not\vdash \alpha$, $P \cup P_D \cup H \vdash \alpha$, and $P \cup P_D \cup H \cup IC$ is consistent.

Note that a hypothesis in a default logic is assumed to be minimal with respect to set inclusion.

Example 4.4 *Let (D, P) be the following closed normal default theory:*

$$D = \left\{ \frac{bird(X) : fly(X)}{fly(X)}, \frac{fish(X) : swim(X)}{swim(X)} \right\},$$

$$P = \left\{ \begin{array}{l} swim(X) \leftarrow penguin(X) \\ bird(X) \leftarrow penguin(X) \end{array} \right\}.$$

Then, the transformed definite program P_D from D is:

$$P_D = \left\{ \begin{array}{l} fly(X) \leftarrow bird(X) \\ swim(X) \leftarrow fish(X) \end{array} \right\}.$$

Let IC_1 be an integrity constraint $\leftarrow fly(X), swim(X)$.

1. *For a ground atom $fly(john)$, $P \cup P_D \not\vdash fly(john)$. Then, $fly(john)$ is a surprising fact for (D, P). The candidates for hypotheses are obtained as follows:*

   $$H_1 = \{fly(john)\},\ H_2 = \{bird(john)\},\ H_3 = \{penguin(john)\}.$$

   For each H_i $(1 \leq i \leq 3)$, H_1 satisfies the first condition, and H_2 satisfies the second condition of Definition 4.3. However, H_3 satisfies neither condition of Definition 4.3, because $P \cup H_3 \cup IC_1$ is inconsistent. Hence, H_1 and H_2 are the hypotheses of $fly(john)$ for $(D, P \cup IC_1)$.

2. *For a ground atom $swim(john)$, $P \cup P_D \not\vdash swim(john)$. Then, $swim(john)$ is a surprising fact for (D, P). The candidates for hypotheses are obtained as follows:*

   $$H_4 = \{swim(john)\},\ H_5 = \{fish(john)\},\ H_6 = \{penguin(john)\}.$$

   For each H_i $(4 \leq i \leq 6)$, H_4 and H_6 satisfy the first condition, and H_5 satisfies the second condition of Definition 4.3. Hence, all of H_4, H_5 and H_6 are the hypotheses of $swim(john)$ for $(D, P \cup IC_1)$.

Let IC_2 be an integrity constraint $\leftarrow bird(X), swim(X)$.

3. *For a surprising fact fly(john), we obtain the same candidates H_1, H_2, and H_3 of hypotheses just as the case 1. For each H_i $(1 \le i \le 3)$, H_1 satisfies the first condition, and H_2 satisfies the second condition of Definition 4.3. However, H_3 satisfies neither condition of Definition 4.3, because $P \cup H_3 \cup IC_2$ is inconsistent. Hence, H_1 and H_2 are the hypotheses of fly(john) for $(D, P \cup IC_2)$.*

4. *For a surprising fact swim(john), we obtain the same candidates H_4, H_5, and H_6 of hypotheses just as the case 2. For each H_i $(4 \le i \le 6)$, H_4 satisfies the first condition, and H_5 satisfies the second condition of Definition 4.3. However, H_6 satisfies neither condition of Definition 4.3, because $P \cup H_6 \cup IC_2$ is inconsistent. Hence, H_4 and H_5 are the hypotheses of swim(john) for $(D, P \cup IC_2)$.*

For a definite program P and a transformed program P_D from D, the following two lemmas hold:

Lemma 4.4 *Let (D, P) be a default theory. Then, the least Herbrand model $M(P \cup P_D)$ of $P \cup P_D$ is an extension of (D, P).*

Proof. Suppose that E is constructed in the following way:

$$\begin{aligned} E_0 &= \{f \mid f \leftarrow \in P\}, \\ E_{i+1} &= Th(E_i) \cup \left\{ w \;\middle|\; \frac{\alpha : w}{w} \in D, \alpha \in E_i \right\}, \\ E &= \bigcup_{i \ge 0} E_i. \end{aligned}$$

Since $\dfrac{\alpha : w}{w} \in D$ is equivalent to $w \leftarrow \alpha \in P_D$, $M(P \cup P_D) = E$. By the closed-world assumption, $\neg w \notin E$ for any w. By Definition 2.4, E is an extension of (D, P). ∎

Lemma 4.5 *Let (D, P) be a default theory. If $P \cup P_D \vdash \alpha$, then $M(P \cup P_D)$ is an extension of (D, P) which includes α.*

Proof. Since $P \cup P_D \vdash \alpha$, $\alpha \in M(P \cup P_D)$. By Lemma 4.4, $M(P \cup P_D)$ is an extension of (D, P). ∎

For an integrity constraint IC and a default theory (D, P), the following lemma holds:

Lemma 4.6 *Let $(D, P \cup IC)$ be a default theory, where P is a definite program and IC is an integrity constraint. If $(D, P \cup IC)$ satisfies one of the following conditions, then $M(P \cup P_D)$ is a consistent extension of $(D, P \cup IC)$ which includes α.*

1. *$P \vdash \alpha$, and $P \cup IC$ is consistent.*

2. *$P \not\vdash \alpha$, $P \cup P_D \vdash \alpha$, and $P \cup P_D \cup IC$ is consistent.*

Proof. Suppose $(D, P \cup IC)$ satisfies the above condition 1. By Lemma 4.2 and the consistency of $P \cup IC$, $M(P)$ is the model of $P \cup IC$. Since $P \vdash \alpha$, $\alpha \in M(P)$. For any β, $P \vdash \beta$ if and only if $P \cup IC \vdash \beta$. Then, E is an extension of (D, P) if and only if E is an extension of $(D, P \cup IC)$. Since $M(P \cup P_D)$ is an extension of (D, P), $M(P \cup P_D)$ is also an extension of $(D, P \cup IC)$.

Suppose $(D, P \cup IC)$ satisfies the above condition 2. By Lemma 4.2 and the consistency of $P \cup P_D \cup IC$, $M(P \cup P_D)$ is the model of $P \cup P_D \cup IC$. Since $P \cup P_D \vdash \alpha$, $\alpha \in M(P \cup P_D)$. If $P \cup IC$ is inconsistent, then $P \cup P_D \cup IC$ is also inconsistent, which contradicts the condition 2. Then, $P \cup IC$ is consistent. Hence, for any β, $P \vdash \beta$ if and only if $P \cup IC \vdash \beta$. Then, E is an extension of (D, P) if and only if E is an extension of $(D, P \cup IC)$. By Lemma 4.5, $E = M(P \cup P_D)$, and $\alpha \in E$. By Lemma 4.4, E is an extension of (D, P). Hence, $M(P \cup P_D)$ is an extension of $(D, P \cup IC)$.

By Corollary 2.1, if $P \cup IC$ is consistent, then an extension of $(D, P \cup IC)$ is also consistent. Hence, the extension of $(D, P \cup IC)$ which includes α is also consistent. ∎

For a closed normal default theory (D, P), the following theorem asserts that if there exists the hypothesis H satisfying one of the conditions of Definition 4.3, then there exists an extension of $(D, P \cup H)$. Hence, we can propose a default theory in which we believe a surprising fact, when it is observed.

Theorem 4.3 *Let (D, P) be a closed normal default theory and IC be an integrity constraint. Suppose that $P \cup P_D \not\vdash \alpha$. If there exists a hypothesis H of α for $(D, P \cup IC)$, then $M(P \cup P_D \cup H)$ is a consistent extension of $(D, P \cup H \cup IC)$ which includes α.*

Proof. By replacing P with $P \cup H$ in the proof of Lemma 4.6, we can obtain the result. ∎

Example 4.5 *Consider the default theory (D, P) in Example 4.4. For the integrity constraint IC_1, we obtain the following extensions E_i of $(D, P \cup H_i \cup IC_1)$ corresponding to the hypotheses H_i:*

$$E_1 = \{\mathit{fly}(\mathit{john})\},\ E_2 = \{\mathit{bird}(\mathit{john}), \mathit{fly}(\mathit{john})\},$$
$$E_4 = \{\mathit{swim}(\mathit{john})\},\ E_5 = \{\mathit{fish}(\mathit{john}), \mathit{swim}(\mathit{john})\},$$
$$E_6 = \{\mathit{penguin}(\mathit{john}), \mathit{swim}(\mathit{john}), \mathit{bird}(\mathit{john})\}.$$

On the other hand, for the integrity constraint IC_2, we also obtain the following extensions E_i of $(D, P \cup H_i \cup IC_2)$ corresponding to the hypotheses H_i:

$$E_1 = \{\mathit{fly}(\mathit{john})\},\ E_2 = \{\mathit{bird}(\mathit{john}), \mathit{fly}(\mathit{john})\},$$
$$E_4 = \{\mathit{swim}(\mathit{john})\},\ E_5 = \{\mathit{fish}(\mathit{john}), \mathit{swim}(\mathit{john})\}.$$

4.3 Breadth-First Rule-Selecting Abduction

In Section 4.1, we have introduced the subclass of definite programs, called head-reducing programs. In this class, all derivations are finite. However, this class is not so large. For example, let P_1 be the following program defining reversal of list:

$$P_1 = \left\{ \begin{array}{l} \mathit{reverse}([W|X], Y) \leftarrow \mathit{reverse}(X, Z), \mathit{concat}(W, Z, Y) \\ \mathit{concat}(X, [W|Y], [W|Z]) \leftarrow \mathit{concat}(X, Y, Z) \end{array} \right\}.$$

The recursive program $RP(P_1, \mathit{reverse})$ is not head-reducing, and P_1 is not head-reducing with respect to the predicate *reverse*. Hence, for a surprising fact α with the predicate *reverse*, rule-selecting abduction does not terminate for the program P and the goal $\leftarrow \alpha$. Then, in this section, we introduce new abduction, called *breadth-first rule-selecting abduction* for which termination is guaranteed in the above reverse programs.

The *breadth-first rule-selecting abduction* is a rule-selecting abduction which terminates by a fail branch in proof trees. Here, a fail branch is found by breadth-first search. For the above program P_1, suppose that a surprising fact $\mathit{reverse}([a, b], [b, a])$

is given. From the proof trees of $P_1 \cup \{\leftarrow reverse([a,b],[b,a])\}$ with the depth 0, 1, and 2, we obtain the following hypotheses H_0, H_1, and H_2:

$$\begin{aligned}
H_0 &= \{reverse([a,b],[b,a])\},\\
H_1 &= \{reverse([b],X), concat(a,X,[b,a])\},\\
H_2 &= \{reverse([\,],X), concat(b,X,[b|Y]), concat(a,Y,[a])\}.
\end{aligned}$$

In the proof tree of $P_1 \cup \{\leftarrow reverse([a,b],[b,a])\}$, since there exists a branch with depth 2, breadth-first rule-selecting abduction for P_1 terminates.

The termination of breadth-first rule-selecting abduction is reduced to the problem whether or not there exists a finite derivation. In order to characterize the termination of breadth-first rule-selecting abduction, we introduce the following concept of *breadth-first head-reducing* programs:

Definition 4.4 *Let $rec(P,p)$ be a recursive definition $p(t_1,\cdots,t_n) \leftarrow B_1,\cdots,B_m$ of p for P. Then, the recursive program $RP(P,p)$ is breadth-first head-reducing if it satisfies one of the following conditions:*

1. *there exist an atom B_i and an index l such that $pred(B_i) = p$ and $|t_l\theta| > |s_l\theta|$ for any substitution θ and the l-th argument's term s_l of B_i, or*
2. *for the atom such that $pred(B_i) \neq p$, one of the following conditions holds:*
   - (a) *the definition clause of $pred(B_i)$ is not included in $RP(P,p)$, or*
   - (b) *there exist terms t_l in $p(t_1,\cdots,t_n)$ and s_j^k in B_k such that $|t_l\theta| > |s_j^k\theta|$ for any substitution θ, and the definition clause of $pred(B_k)$ is $pred(B_k)$-reducing with respect to the j-th argument.*

Furthermore, P is breadth-first head-reducing with respect to the predicate p if any recursive program $RP(P,p)$ of p for P is breadth-first head-reducing.

Example 4.6 *For the above program P_1, a recursive definition $rec(P_1, reverse)$ is obtained uniquely as follows:*

$$rec(P_1, reverse) = reverse([W|X],[V|Y]) \leftarrow reverse(X,[V|Z]), concat(W,Z,Y).$$

Then, recursive program $RP(P_1, reverse)$ is also obtained as follows:

$$RP(P_1, reverse) = \left\{ \begin{array}{l} reverse([W|X],[V|Y]) \leftarrow reverse(X,[V|Z]), concat(W,Z,Y) \\ concat(X,[W|Y],[W|Z]) \leftarrow concat(X,Y,Z) \end{array} \right\}.$$

It is clear that $RP(P_1, reverse)$ satisfies the condition 1 of Definition 4.4. Hence, P is breadth-first head-reducing with respect to the predicate reverse. Furthermore, the following programs P_2 and P_3 are also breadth-first head-reducing with respect to the predicates isort and qsort respectively, where s is a successor function:

$$P_2 = \left\{ \begin{array}{l} isort([X|Xs],Ys) \leftarrow isort(Xs,Zs), insert(X,Zs,Ys) \\ insert(X,[Y|Ys],[Y|Zs]) \leftarrow greater(X,Y), insert(X,Ys,Zs) \\ insert(X,[Y|Ys],[Y|Zs]) \leftarrow less\_or\_eq(X,Y) \\ greater(s(X),Y) \leftarrow greater(X,Y) \\ less\_or\_eq(X,s(Y)) \leftarrow less\_or\_eq(X,Y) \end{array} \right\},$$

$$P_3 = \left\{ \begin{array}{l} qsort([X|Xs],Ys) \leftarrow partition(Xs,X,Littles,Bigs), \\ \qquad\qquad\qquad\qquad qsort(Littles,Ls), \\ \qquad\qquad\qquad\qquad qsort(Bigs,Bs), \\ \qquad\qquad\qquad\qquad append(Ls,[X|Bs],Ys) \\ partition([X|Xs],Y,[X|Ls],Bs) \leftarrow less\_or\_eq(X,Y), \\ \qquad\qquad\qquad\qquad\qquad\qquad partition(Xs,Y,Ls,Bs) \\ partition([X|Xs],Y,Ls,[X|Bs]) \leftarrow greater(X,Y), \\ \qquad\qquad\qquad\qquad\qquad\qquad partition(Xs,Y,Ls,Bs) \\ append([W|X],Y,[W|Z]) \leftarrow append(X,Y,Z) \\ greater(s(X),Y) \leftarrow greater(X,Y) \\ less\_or\_eq(X,s(Y)) \leftarrow less\_or\_eq(X,Y) \end{array} \right\}.$$

On the other hand, the following programs P_4, P_5, and P_6 are not breadth-first head-reducing with respect to the predicate p:

$$P_4 = \{p(X) \leftarrow p(X)\},$$
$$P_5 = \left\{ \begin{array}{l} p(X) \leftarrow q(X) \\ q(X) \leftarrow p(X) \end{array} \right\},$$
$$P_6 = \{p(X,Z) \leftarrow p(X,Y), p(Y,Z)\}.$$

Lemma 4.7 *Let C be a clause $p(t_1,\cdots,t_n) \leftarrow B_1,\cdots,B_m$ and $p(s_1,\cdots,s_n)$ be a ground atom. If there exists an atom B_i which satisfies one of the following condition, then there exists a finitely failed SLD-derivation of $\{C\} \cup \{\leftarrow p(s_1,\cdots,s_n)\}$:*

1. *$pred(B_i) \neq p$, or*

2. *pred*$(B_i) = p$, *and there exists an index* j *such that* $|t_j\theta| > |s^i_j\theta|$ *for any substitution* θ.

Proof. If $\{C\} \cup \{\leftarrow p(s_1, \cdots, s_n)\}$ holds the condition 1, then this derivation is finitely failed with a depth 1 by selecting an atom B_i.

Suppose that the condition 1 does not hold. Then, for any k, $pred(B_k) = p$. If the condition 2 holds, then, by selecting an atom B_i which satisfies the condition 2, the resolvent

$$\leftarrow B_1\theta, \cdots, B_i\theta, \cdots, B_m\theta$$

is obtained from the goal $\leftarrow p(s_1, \cdots, s_n)$ and clause C. Since $|t_j\theta| > |s^i_j\theta|$,

$$|s^i_j\theta| < |t_j\theta| = |s_j|.$$

Furthermore, by selecting an atom $B_i\theta$, the length of the j-th argument's term is decrease by 1 step by step for this derivation of $\{C\} \cup \{\leftarrow p(s_1, \cdots, s_n)\}$. Hence, this derivation is finitely failed with at most the depth $|s_j| \leq max\{|s_j| \mid 1 \leq j \leq n\}$. ∎

Then, we can show the following theorem for the termination of breadth-first rule-selecting abduction.

Theorem 4.4 *Let* P *be a definite program and* α *be a ground atom with a predicate* p. *If* P *is breadth-first head-reducing with respect to* p, *then there exists a finitely failed SLD-derivation of* $P \cup \{\leftarrow \alpha\}$.

Proof. Let $rec(P, p)$ be $p(t_1, \cdots, t_n) \leftarrow B_1, \cdots, B_m$. If P satisfies the condition 1 or 2 of Definition 4.4, then Lemma 4.7 implies the result.

Suppose that P does not satisfy the conditions 1 and 2, and satisfies the condition 3 in Definition 4.4. Let B_k be an atom $q(s^k_1, \cdots, s^k_h)$, and $|t_l\theta| > |s^k_j\theta|$ for any substitution θ. Let C be a definition clause $q(u_1, \cdots, u_h) \leftarrow A_1, \cdots, A_{l'}$ of q in $RP(P, p)$. Since C is q-reducing with respect to the j-th argument, then, for any i such that $pred(A_i) = q$, $|u_j\theta| > |v_j\theta|$, where $A_i = q(v_1, \cdots, v_h)$. By the definition of $rec(P, p)$, the following resolvent is obtained in the derivation of $P \cup \{\leftarrow \alpha\}$:

$$\leftarrow B_1\theta, \cdots, B_k\theta, \cdots, B_m\theta,$$

where θ is a unifier of α and $p(t_1, \cdots, t_n)$. Since $|s_j^k\theta| < |t_l\theta| = |s_l|$, and s_l is ground, $s_j^k\theta$ is also ground. By selecting an atom $B_k\theta$, the resolvent

$$\leftarrow B_1\theta\sigma, \cdots, B_{k-1}\theta\sigma, (A_1\sigma, \cdots, A_{l'}\sigma), B_{k+1}\theta\sigma, \cdots, B_m\theta\sigma$$

is obtained from the above goal and the clause C. For the j-th argument v_j in $A_i\sigma$,

$$|v_j\sigma| < |u_j\sigma| = |s_j^k\theta| < |t_l\theta| = |s_l|,$$

and $v_j\sigma$ is ground. Furthermore, we can select an atom $A_i\sigma$ applied to C. Consequently, this derivation is finitely failed with at most the depth $|s_l| < max\{|s_j| \mid 1 \leq j \leq n\}$. ∎

4.4 Prolog Implementation

As mentioned in Section 4.1, the rule-selecting abduction can be realized in the following Prolog program rs_abd.

```
rs_abd(Goal,Leaves) :- clause(Goal,Clause),rs_abd(Clause,Leaves).
rs_abd((Goal1,Goal2),(Leaf1,Leaf2)) :-
        !,rs_abd(Goal1,Leaf1),rs_abd(Goal2,Leaf2).
rs_abd(Leaf,Leaf) :- !.
```

Furthermore, we can improve the program rs_abd as the following program msrs_abd by using the concept of *most specific abduction* in Stickel [Sti91, Duv91, Ino92].

```
msrs_abd(Goal,Leaves) :- clause(Goal,Clause),msrs_abd(Clause,Leaves).
msrs_abd((Goal1,Goal2),(Leaf1,Leaf2)) :-
        !,msrs_abd(Goal1,Leaf1),msrs_abd(Goal2,Leaf2).
msrs_abd(Leaf,Leaf) :- (not clause(Leaf,X) -> true).
```

The program msrs_abd returns the combinations of the leaves for all proof trees as hypotheses, while the rs_abd program returns the combinations of the nodes for all proof trees. Furthermore, for the above programs, the following corollary of Theorem 4.1 holds:

Corollary 4.1 *Let P be a definite program and p be a predicate symbol. If P is head-reducing with respect to the predicate p, then the following goals*

?- rs_abd($p(s_1, \cdots, s_n)$,X),

?- msrs_abd($p(s_1, \cdots, s_n)$,X),

terminate for any ground atom $p(s_1, \cdots, s_n)$.

The breadth-first rule-selecting abduction can also be realized in the following Prolog program `bfrs_abd`. It returns the hypotheses as its second argument for a ground atom as its first argument and a depth as its third argument.

```
bfrs_abd(Goal,Leaves,Depth) :-
        Depth > 0,clause(Goal,Clause),Depth1 is Depth-1,
        bfrs_abd(Clause,Leaves,Depth1).
bfrs_abd((Goal1,Goal2),(Leaf1,Leaf2),Depth) :-
        !,bfrs_abd(Goal1,Leaf1,Depth),bfrs_abd(Goal2,Leaf2,Depth).
bfrs_abd(Leaf,Leaf,0) :- !.
```

Note that, for the program `bfrs_abd`, we must give a natural number as the depth in its third argument.

For example, for the following reverse program in Section 4.3

```
reverse([W|X],Y) :- reverse(X,Z),concat(W,Z,Y).
concat(X,[W|Y],[W|Z]) :- concat(X,Y,Z).
```

the results of the goal ?- `bfrs_abd(reverse([a,b],[b,a]),X,D)` are as follows:

```
: ?- bfrs_abd(reverse([a,b],[b,a]),X,0).     %%% depth = 0
X = reverse([a,b],[b,a]);
no
: ?- bfrs_abd(reverse([a,b],[b,a]),X,1).     %%% depth = 1
X = reverse([b],_260),concat(a,_260,[b,a]);
no
: ?- bfrs_abd(reverse([a,b],[b,a]),X,2).     %%% depth = 2
X = (reverse([],_376),concat(b,_376,[b|_514])),concat(a,_514,[a]);
no
: ?- bfrs_abd(reverse([a,b],[b,a]),X,3).     %%% depth = 3
no
```

For the program `bfrs_abd`, the following corollary of Theorem 4.4 holds:

```
hyp(Goal,Hyp,IC) :-
        rs_abd(Goal,Hyp),
        ((rep_assert(Hyp),consistent(Goal,Hyp,IC),rep_retract(Hyp))
                ->true;!,fail).
consistent(Goal,Hyp,IC) :-
        (call(Goal)->
               (call(ic(IC))->(write(': consistent'),nl) ;
               (write(': inconsistent'),nl)) ;
        (write(': inconsistent'),nl)).
ic(IC) :- (call(IC)->fail;true).
rep_assert((Atom1,Atom2)) :-
        (atom(Atom1)->assert(Atom1);rep_assert(Atom1)),
        (atom(Atom2)->assert(Atom2);rep_assert(Atom2)).
rep_assert(Atom) :- assert(Atom).
rep_retract((Atom1,Atom2)) :-
        (atom(Atom1)->retract(Atom1);rep_retract(Atom1)),
        (atom(Atom2)->retract(Atom2);rep_retract(Atom2)).
rep_retract(Atom) :- retract(Atom).
```

Figure 4.1: Program `hyp`

Corollary 4.2 *Let α be a ground atom with predicate p. If P is breadth-first head-reducing with respect to the predicate p, then there exists a depth* `d` *such that the goal*

`?- bfrs_abd(`α`,X,d)`

returns "`no`*".*

For the above example, the depth in Corollary 4.2 is 3.

The rule-selecting abduction for default logic is also realized as the Prolog program `hyp` in Figure 4.1. The program checks the consistency under an integrity constraint and outputs hypotheses. An integrity constraint is given as the form of disjunctions (expressed by ';') of conjunctions (expressed by ',') of atoms in the third argument of `hyp`. If there exists no refutation of a goal as conjunctions of the atoms in the integrity constraint, then the hypotheses are consistent with the given program and the integrity constraint.

The predicate `hyp` in Figure 4.1 works as follows: The predicate `rs_abd` returns a hypothesis as its second argument for a surprising fact as its first argument. The predicate rep_assert adds the hypothesis obtained by rs_abd to the original program. The predicate consistent checks consistency for the hypothesis and the integrity constraint, which is given as the third argument of hyp. The predicate rep_retract removes the hypothesis added by rep_assert from the original program. The predicate ic calls the integrity constraint, and returns 'true' (*resp.*, 'fail') if the integrity constraint fails (*resp.*, succeeds) on the original program.

The termination of the program hyp is also guaranteed by the following corollary of Theorem 4.1:

Corollary 4.3 *Let P be a definite program, IC be an integrity constraint, and p be a predicate symbol. For any predicate symbol q in IC, if P is head-reducing with respect to the predicate p and q, then the goal*

$$\texttt{?- hyp(}p(s_1,\cdots,s_n)\texttt{,X,IC)}$$

terminates for any ground atom $p(s_1,\cdots,s_n)$.

In order to apply default logic to the above program, we transform the set of default rules D to the following definite program P_D^+:

$$\left\{ w(\overline{X}) \leftarrow \alpha(\overline{X}), \mathit{default\_w}(\overline{X}) \;\middle|\; \frac{\alpha(\overline{X}) : w(\overline{X})}{w(\overline{X})} \in D \right\}.$$

Thus, we interpret $\mathit{default\_w}(\overline{t})$ appearing in a hypothesis as $w(\overline{t})$.

Example 4.7 *Consider $P \cup P_D^+$ which consists of the following clauses:*

```
fly(X) :- bird(X),default_fly(X).     %%% default
swim(X) :- fish(X),default_swim(X).   %%% default
swim(X) :- penguin(X).                %%% theory
bird(X) :- penguin(X).                %%% theory
```

Let IC_1 and IC_2 be integrity constraints $\leftarrow \mathit{fly}(Y), \mathit{swim}(Y)$ and $\leftarrow \mathit{bird}(Y), \mathit{swim}(Y)$, respectively. Since the above $P \cup P_D^+$ and IC_i $(i = 1, 2)$ satisfy the conditions of Corollary 4.3, the hyp program terminates and outputs the following hypotheses:

```
: ?- hyp(fly(john),X,(fly(Y),swim(Y))).              %%% IC1 %%%
penguin(john) , default_fly(john): inconsistent
bird(john) , default_fly(john): consistent
```

```
fly(john): consistent

: ?- hyp(swim(john),X,(fly(Y),swim(Y))).        %%% IC1 %%%
fish(john) , default_swim(john): consistent
penguin(john): consistent
swim(john): consistent

: ?- hyp(fly(john),X,(bird(Y),swim(Y))).        %%% IC2 %%%
penguin(john) , default_fly(john): inconsistent
bird(john) , default_fly(john): consistent
fly(john): consistent

: ?- hyp(swim(john),X,(bird(Y),swim(Y))).       %%% IC2 %%%
fish(john) , default_swim(john): consistent
penguin(john): inconsistent
swim(john): consistent
```

The predicate hyp is also applied to explanation-based generalization. *Explanation-based generalization* (for short, *EBG*) [Duv91, GMP90, HiA94a, MKKC86, vHB88] takes as inputs a *domain theory*, a *training example*, a *goal concept* and an *operationally criterion*. It constructs an explanation in term of the domain theory that proves how the training example satisfies the goal concept definition. Then, it determines a set of operationally sufficient conditions for the goal concept under which the explanation holds, and returns it as an output.

When we realize EBG as a Prolog program, we regard the domain theory and the training example as a definite program. Then, we construct a proof tree, which is called an *explanation tree* [HiA94a], and generalize it to obtain the general definition of goal concept. On the other hand, rule-selecting abduction is an inference from rules to facts. Then, it is corresponding to obtaining a training example from a domain theory in EBG.

Consider the *safe-to-stack problem* in Mitchell *et al.* [MKKC86]. A domain theory D of the safe-to-stack problem is the following definite program.

```
safe_to_stack(X,Y) :- lighter(X,Y).
lighter(X,Y) :- weight(X,W1),weight(Y,W2),less(W1,W2).
weight(X,500) :- isa(X,table).
weight(X,Y) :- volume(X,V),density(X,D),times(V,D,Y).
```

For the domain theory D, the goal

```
?- rs_abd(safe_to_stack(box1,table1),X)
```

terminates, because there exists exactly one recursive program $RP(D, \mathit{safe\_to\_stack})$ of the predicate *safe_to_stack* for D, and it is head-reducing. Furthermore, since D and an integrity constraint $\neg isa(box1, table) \vee \neg isa(table1, box)$ satisfy the condition of Corollary 4.3, the following goal

```
?- hyp(safe_to_stack(box1,table1),X,(isa(box1,table);isa(table1,box)))
```

terminates. Hence, we obtain the following hypotheses as the second argument:

```
isa(box1,table),isa(table1,table),less(500,500): inconsistent

isa(box1,table),
(volume(table1,_510),density(table1,_506),times(_510,_506,_310)),
less(500,_310)
: inconsistent

isa(box1,table),weight(table1,_310),less(500,_310): inconsistent

(volume(box1,_430),density(box1,_426),times(_430,_426,_314)),
isa(table1,table),less(_314,500)
: consistent

(volume(box1,_430),density(box1,_426),times(_430,_426,_314)),
(volume(table1,_608),density(table1,_604),times(_608,_604,_310)),
less(_314,_310)
: consistent

(volume(box1,_430),density(box1,_426),times(_430,_426,_314)),
weight(table1,_310),less(_314,_310)
: consistent

weight(box1,_314),isa(table1,table),less(_314,500): consistent

weight(box1,_314),
(volume(table1,_456),density(table1,_452),times(_456,_452,_310)),
less(_314,_310)
: consistent

weight(box1,_314),weight(table1,_310),less(_314,_310): consistent

lighter(box1,table1): consistent

safe_to_stack(box1,table1): consistent
```

Then, it is sufficient for EBG to give a training example as ground examples of a consistent hypotheses. Furthermore, if we consider that a hypothesis is the set of leaves in explanation tree, then, by replacing the predicate `rs_abd` with the predicate `msrs_abd` in the program `hyp`, we can obtain the first four outputs as hypotheses.

Chapter 5

Rule-Finding Abduction

"How absurdly simple!" I cried.

"Quite so!" said he, a little nettled. "Every problem becomes very childish when once it is explained to you."

— *'The Adventure of the Dancing Men'*
"The Return of Sherlock Holmes"

Let P and P' be definite programs and C be a surprising fact with respect to P. Note here that P and P' are given before α is given. The *rule-finding abduction* is a type of abduction which finds a rule in P' and proposes a hypothesis to explain the surprising fact C. An inference schema of rule-finding abduction is described by the following three steps:

1. A surprising fact C is observed.
2. A rule $C \leftarrow A$ is *found* in P'.
3. A hypothesis A is proposed.

For a surprising fact C, we regard the above inference schema as the following one by identifying A with the set $\{A\}$ of atoms:

1. A ground atom C such that $P \not\models C$ is given.
2. A rule $C' \leftarrow A'_1, \cdots, A'_n$ is *found* in P', where $C'\theta = C$ and $A'_i\theta = A_i$.
3. A hypothesis $\{A_1, \cdots, A_n\}$ is proposed in P.

In general, since we assume the set $\{P_1, \cdots, P_l\}$ of definite programs in rule-finding abduction, the above P' is equal to some P_i $(1 \leq i \leq l)$.

In this chapter, we investigate rule-finding abduction for logic programming. In Section 5.1, we give two examples, and explain what rule-finding abduction is. In Section 5.2, we investigate the concept of *abducible* in the abductive framework. In Section 5.3, we introduce the concept of *loop-pair*. It syntactically determines whether or not there exists an infinite process of rule-finding abduction. In Section 5.4, we also introduce the concept of *loop-elimination*, which is a transformation of programs for which rule-finding abduction terminates. In Section 5.5, we realize rule-finding abduction and loop-elimination as Prolog programs. In Section 5.6, we discuss analogical reasoning from the viewpoint of rule-finding abduction.

This chapter is based on the paper [Hir94d].

5.1 Rule-Finding Abduction for Logic Programming

Consider the following fossil-shell example. Let P_i $(1 \leq i \leq 3)$ be the following sets of clauses:

$$P_1 = \left\{ \ \mathit{find}(X, \mathit{fossil\_shell}, Y) \leftarrow \mathit{sea}(Y) \ \right\},$$

$$P_2 = \left\{ \begin{array}{l} \mathit{find}(X, \mathit{fossil\_shell}, Y) \leftarrow \mathit{used\_to\_be}(Y, \mathit{sea}) \\ \mathit{used\_to\_be}(X, Y) \leftarrow \mathit{be}(X, Y) \end{array} \right\},$$

$$P_3 = \left\{ \begin{array}{l} \mathit{find}(X, \mathit{fossil\_shell}, Y) \leftarrow \mathit{move}(\mathit{fossil\_shell}, \mathit{sea}, Y) \\ \mathit{move}(\mathit{fossil\_shell}, X, Y) \leftarrow \mathit{slow\_move}(\mathit{fossil\_shell}, X, Y) \\ \mathit{slow\_move}(X, Y, Z) \leftarrow \mathit{has\_not\_leg}(X) \\ \mathit{slow\_move}(X, Y, Z) \leftarrow \mathit{has\_not\_wing}(X) \end{array} \right\}.$$

Let α be a surprising fact $find(i, fossil\_shell, mountain)$. In rule-finding abduction, by selecting a program P_i from the set $\{P_1, P_2, P_3\}$ of programs, we find a rule and propose a hypothesis to explain the surprising fact α. We call such a selected program an *applied program of* α *for* $\{P_1, P_2, P_3\}$. Then, Figure 5.1 illustrates the applied programs and hypotheses H_i of α for $\{P_1, P_2, P_3\}$. For the applied program P_1, $P_1 \cup H_1 \vdash \alpha$. For the applied program P_2, $P_2 \cup H_2 \vdash \alpha$ and $P_2 \cup H_3 \vdash \alpha$. For the applied

applied programs	*hypotheses*
nothing	$H_0 = \{\mathit{find}(i, \mathit{fossil\_shell}, \mathit{mountain})\}$
P_1	$H_1 = \{\mathit{sea}(\mathit{mountain})\}$
P_2	$H_2 = \{\mathit{be}(\mathit{mountain}, \mathit{sea})\}$
	$H_3 = \{\mathit{used\_to\_be}(\mathit{mountain}, \mathit{sea})\}$
P_3	$H_4 = \{\mathit{has\_not\_leg}(\mathit{fossil\_shell})\}$
	$H_5 = \{\mathit{has\_not\_wing}(\mathit{fossil\_shell})\}$
	$H_6 = \{\mathit{slow\_move}(\mathit{fossil\_shell}, \mathit{sea}, \mathit{mountain})\}$
	$H_7 = \{\mathit{move}(\mathit{fossil\_shell}, \mathit{sea}, \mathit{mountain})\}$

Figure 5.1: Applied programs and hypotheses H_i of α for $\{P_1, P_2, P_3\}$

applied programs	*hypotheses*
nothing	$K_0 = \{p(f^3(a))\}$
P_4	$K_1 = \{p(f(a)), q(f^2(a))\}$
P_4, P_5	$K_2 = \{p(f(a)), q(a), r(a, f(a)), r(f(a), f(a))\}$
	$K_3 = \{p(f(a)), q(f(a)), r(f(a), f^2(a))\}$
P_4, P_5, P_6	$K_4 = \{p(f(a)), q(a), r(a, f(a)), r(a, f(a))\}$
	$K_5 = \{p(f(a)), q(f(a)), r(a, f(a))\}$

Figure 5.2: Applied programs and hypotheses K_i of β for $\{P_4, P_5, P_6\}$

program P_3, $P_3 \cup H_i \vdash \alpha$ $(4 \leq i \leq 7)$. The hypothesis H_0 is a trivial hypothesis, that is, $H_0 \vdash \alpha$.

Furthermore, we can give the example of programs with function symbols and recursion. Let P_i $(4 \leq i \leq 6)$ be the following sets of clauses:

$$P_4 = \left\{ \ p(f^2(X)) \leftarrow p(X), q(f(X)) \ \right\},$$

$$P_5 = \left\{ \ q(f(X)) \leftarrow q(X), r(X, f(X)) \ \right\},$$

$$P_6 = \left\{ \ r(f(X), f(Y)) \leftarrow r(X, Y) \ \right\}.$$

For a surprising fact $\beta = p(f^3(a))$, Figure 5.2 illustrates the applied programs and hypotheses of β for $\{P_4, P_5, P_6\}$. For the applied program P_4, $P_4 \cup K_1 \vdash \beta$. For the applied programs P_4 and P_5, $P_4 \cup P_5 \cup K_2 \vdash \beta$ and $P_4 \cup P_5 \cup K_3 \vdash \beta$. For the applied programs P_4, P_5, and P_6, $P_4 \cup P_5 \cup P_6 \cup K_4 \vdash \beta$ and $P_4 \cup P_5 \cup P_6 \cup K_5 \vdash \beta$. The hypothesis K_0 is a trivial hypothesis, that is, $K_0 \vdash \beta$.

Let P_i be a program for $1 \leq i \leq n$. If any program P_i is given before we apply to rule-finding abduction, then the termination of rule-finding abduction is reduced to one of rule-selecting abduction for the union $P_1 \cup \cdots \cup P_n$ of programs, and we have already discussed it in Chapter 4.

On the other hand, when we discuss the termination of abduction, we can adopt at least two strategies. One is the restriction of class of logic programs. The discussion in Chapter 4 gives an example of it. The other is the introduction of the criterion of termination. For example, in EBG, it is given as an *operationality criterion*. In the following sections, we discuss the later strategy for the termination of rule-finding abduction.

In rule-finding abduction, we should choose the programs for which rule-finding abduction terminates. Hence, it is our purpose in the following sections how to choose the programs to avoid an infinite process of rule-finding abduction.

5.2 Abducible Predicate

An *abducible predicate* (*abducible*, for short) is defined in an *abductive framework* [Dun91, EK89, KM90, Poo88]. First, we give the definition of the abductive framework as follows:

Definition 5.1 (Poole [Poo88]) *An abductive framework is defined as the triple (P, I, A), where P is a set of Horn clause, I is an integrity constraint, and A is a set of predicate symbols called abducible.*

In this chapter, we only deal with an abductive framework of definite programs. Then, an abductive framework is defined as the pair (P, A) without an integrity constraint, where P is a definite program. Here, an *abducible* means the set of predicate symbols of atoms which are assumed true or are hypotheses.

In an abductive framework, an *explanation* of α for (P, A) is defined as follows:

Definition 5.2 *Let α be a ground atom, and (P, A) be an abductive framework. Then, an explanation of α for (P, A) is a set H of atoms such that $P \cup H \vdash \alpha$ and $\Pi(H) \subseteq A$.*

Example 5.1 *Let P be the following program and $A = \{q\}$.*

$$P = \left\{ \begin{array}{l} p(X) \leftarrow q(X) \\ q(X) \leftarrow p(X) \end{array} \right\}.$$

Then, for a ground atom $\alpha = p(a)$, the set $H = \{q(a)\}$ is an explanation of α for (P, A). On the other hand, let $A' = \{r\}$. Then, there exist no explanations of α for (P, A').

An abducible is similar to an *operationality criterion*, which is introduced in EBG. Note that the purpose of abductive framework is different from that of EBG. An abductive framework is related to nonmonotonic reasoning or knowledge representation, while EBG is related to machine learning or knowledge acquisition.

For a ground atom α, the leaves of the proof tree of α given by EBG are elements of an operationality criterion, and we can regard them as abducible. Note that, in EBG, an operationality criterion is given before a proof tree is constructed. In other words, an operationality criterion is introduced in order to guarantee that the proof tree is finite.

Then, which of atoms is an abducible?

If a proof tree is finite, then the leaves of it are possible to be an abducible. Furthermore, for the set H of nodes in the proof tree, if any branch of the proof tree includes at least one element of H, then H can be regarded as an abducible. In Section 5.5, we realize the program whose outputs are such an H as Prolog program, if all proof trees are finite.

However, if the class of programs is not restricted, we cannot determine before the proof tree is constructed whether or not the branch of a proof tree is finite. Hence, in the next section, we investigate the syntactical characterization of programs whose proof trees have an infinite branch.

5.3 Loop-Pair

When we debug a Prolog program, we search for the proof trees of it, and check whether or not it correctly works according as our intention. If there exists an infinite branch of the proof trees, then this program is not designed with our intention. Hence, it is an important view for Prolog debugging to determine whether or not the branch of a proof tree is infinite. In order to solve this problem, we introduce the concept of a *loop-pair*. We deal with the loop-pair to syntactically characterize the termination of rule-finding abduction.

Definition 5.3 *Let s and t be terms. Then, a loop-pair $\langle\langle s,t\rangle\rangle$ is inductively defined as follows:*

1. *If s is a constant symbol a, then t is a term which includes the constant symbol a or a variable X as subterm.*
2. *If s is a variable X, then t is either a term which includes the variable X as subterm, or a variable Y.*
3. *If s is a term $f(s_1,\cdots,s_m)$, t is a term $f(t_1,\cdots,t_m)$, and $\langle\langle s_i,t_i\rangle\rangle$ is a loop-pair for any i $(1\leq i\leq m)$, then so is $\langle\langle s,t\rangle\rangle$.*

Example 5.2 *The following pairs are loop-pairs:*

$$\begin{gathered}\langle\langle a,a\rangle\rangle,\ \langle\langle a,f(a)\rangle\rangle,\ \langle\langle a,X\rangle\rangle,\ \langle\langle a,f(X)\rangle\rangle,\\ \langle\langle X,X\rangle\rangle,\ \langle\langle X,f(X)\rangle\rangle,\ \langle\langle X,Y\rangle\rangle,\\ \langle\langle f(a,b),f(X,b)\rangle\rangle,\ \langle\langle g(a,f(X)),g(X,f(Y))\rangle\rangle.\end{gathered}$$

Definition 5.4 *Let α and β be atoms $p(s_1,\cdots,s_n)$ and $p(t_1,\cdots,t_n)$, respectively. Then, $\langle\langle\alpha,\beta\rangle\rangle$ is a loop-pair if $\langle\langle s_i,t_i\rangle\rangle$ is a loop-pair for any i $(1\leq i\leq n)$.*

Lemma 5.1 *Let α and β be atoms $p(s_1,\cdots,s_n)$ and $\beta=p(t_1,\cdots,t_n)$, respectively. Let C be the following clause:*

$$C=p(u_1,\cdots,u_n)\leftarrow p(v_1,\cdots,v_n).$$

If $\langle\langle \alpha, \beta \rangle\rangle$ is a loop-pair, $\alpha\theta = p(u_1, \cdots, u_n)\theta$, and $\beta = p(v_1, \cdots, v_n)\theta$, then there exists an atom γ such that

1. *$\langle\langle \beta, \gamma \rangle\rangle$ is a loop-pair,*

2. *there exists a substitution σ such that $\beta\sigma = p(u_1, \cdots, u_n)\sigma$, and*

3. *$\gamma = p(v_1, \cdots, v_n)\sigma$.*

Proof. Let γ be an atom $p(w_1, \cdots, w_n)$. The result is proven by mathematical induction on the structure of t_i. Note that different capital letters represent different variables.

1. If t_i is a constant symbol a, then $s_i = a$ and $u_i = v_i = X$. Hence, $w_i = a$.

2. If t_i is a variable X, then the following three cases hold:

   (a) If s_i is a constant symbol a, then $u_i = U$ and $v_i = V$. Hence, $w_i = W$.

   (b) If s_i is the variable X, then $u_i = v_i = U$. Hence, $w_i = X$.

   (c) If s_i is a variable Y different from X, then $u_i = U$ and $v_i = V$. Hence, $w_i = W$.

3. If t_i is the form of $f(t'_1, \cdots, t'_n)$, then the following two cases hold:

   (a) If s_i is a subterm of t_i, then u_i is also a subterm of v_i. Hence, t_i is a subterm of w_i.

   (b) Otherwise, s_i is the form of $f(s'_1, \cdots, s'_n)$ and suppose that $\langle\langle s'_i, t'_i \rangle\rangle$ is a loop-pair for any i $(1 \leq i \leq n)$. Then, $s'_i\theta = u'_i\theta$, $t'_i = v'_i\theta$, $t'_i\sigma = u'_i\sigma$, and $v'_i = v'_i\sigma$. Hence, $s_i\sigma = u_i\sigma$ and $w_i = v_i\sigma$.

Hence, in each case, $\langle\langle t_i, w_i \rangle\rangle$ is a loop-pair, and $w_i = v_i\sigma$ for some substitution σ. Therefore, $\langle\langle \beta, \gamma \rangle\rangle$ is a loop-pair, $\beta\sigma = p(u_1, \cdots, u_n)\sigma$, and $\gamma = p(v_1, \cdots, v_n)\sigma$. ∎

Theorem 5.1 *Let P be a definite program, α and β be atoms, and $C_1, \cdots, C_n \in P$ be the applied clauses in the derivation from the goal $\leftarrow \alpha$ to the goal $\leftarrow \beta$ in P. If $\langle\langle \alpha, \beta \rangle\rangle$ is a loop-pair, then there exists an atom γ such that*

1. $\langle\langle \beta, \gamma \rangle\rangle$ *is a loop-pair, and*

2. *the goal* $\leftarrow \gamma$ *is derived from* $\leftarrow \beta$ *by applying the clauses* $C_1, \cdots, C_n \in P$.

Proof. For any C_i, by applying the selected atoms in the derivation from $\leftarrow \alpha$ to $\leftarrow \beta$, there exists an atom γ which satisfies the above condition 2. Then, we can reduce the result to Lemma 5.1. ∎

Let P be a definite program. If all predicate symbols in the heads of clauses in P are mutually distinct, then the input clauses in a derivation are determined uniquely for a goal. Hence, the following corollary holds:

Corollary 5.1 *Let P be a definite program and α be an atom. Suppose that all predicate symbols in the heads of clauses in P are mutually distinct. If a loop-pair appears in the branch of the proof tree of α on P, then this branch is infinite.*

By Corollary 5.1, we can select the programs which do not include such a branch, in order to avoid infinite branches of the proof tree.

5.4 Loop-Elimination

In rule-finding abduction, we can deal with the several programs given in advance. Then, in this section, we discuss the termination of rule-finding abduction by choosing programs.

It is a useful method for Prolog debugging to obtain and to analyze the transformed program whose termination is guaranteed, and to debug the original one. In this section, we discuss the termination of rule-finding abduction from this viewpoint. In Chapter 4, we have already captured the termination of rule-selecting abduction as head-reducing programs. Hence, this section also begins with head-reducing programs.

For a program P', if a program P is head-reducing with respect to the predicate p, is the union $P \cup P'$ head-reducing with respect to the predicate p?

Example 5.3 *Let P_1 and P_2 be programs $\{p(X) \leftarrow q(X)\}$ and $\{q(X) \leftarrow p(X)\}$. Clearly P_1 and P_2 are head-reducing with respect to the predicate p. Then, the union $P_1 \cup P_2$ is the following program:*

$$P_1 \cup P_2 = \left\{ \begin{array}{l} p(X) \leftarrow q(X) \\ q(X) \leftarrow p(X) \end{array} \right\}.$$

Obviously, $P_1 \cup P_2$ is not head-reducing with respect to the predicate p.

In general, even if P and P' are head-reducing with respect to the some predicate, $P \cup P'$ is not always head-reducing with respect to the same predicate. Then, is there the choice of programs whose union is head-reducing? In particular, for a clause C and a head-reducing program P with respect to the predicate p, we consider the condition under which $P \cup \{C\}$ is head-reducing with respect to p. First, we define *reducing programs*, which are more restricted than head-reducing programs, introduced by Yamamoto [Yam92].

Definition 5.5 (Yamamoto [Yam92]) *A clause $A \leftarrow B_1, \cdots, B_n$ is reducing if $|A\theta| > |B_i\theta|$ $(1 \leq i \leq n)$ for any substitution θ. A program P is a reducing program if all clauses in P are reducing.*

By Definition 5.5, any reducing program is also head-reducing with respect to any predicate. Furthermore, if P is a reducing program and C is reducing, then $P \cup \{C\}$ is also a reducing program. Then, $P \cup \{C\}$ is head-reducing with respect to any predicate. However, the following cases 1 and 2 hold:

1. Let P_3 be a program $\{p(f^2(X)) \leftarrow p(X), q(f(X))\}$, and C_3 be a clause $q(X) \leftarrow p(f^3(X))$. Then, $P_3 \cup \{C_3\}$ is not head-reducing with respect to the predicate p. Hence, even if P is a reducing program and C is p-reducing with respect to all arguments, $P \cup \{C\}$ is not always head-reducing with respect to p.

2. Let P_4 be a program $\{p(X) \leftarrow q(Y)\}$ and C_4 be a clause $q(f(X)) \leftarrow p(X)$. Then, $P_4 \cup \{C_4\}$ is not head-reducing with respect to the predicate p. Hence, even if P is head-reducing with respect to the predicate p and C is reducing, $P \cup \{C\}$ is not always head-reducing with respect to p.

Hence, if we extend a reducing program P to a head-reducing program with respect to the predicate p, or extend a reducing clause C to a p-reducing clause with respect to all arguments, then $P \cup \{C\}$ is not always head-reducing with respect to the predicate p.

Let P be a head-reducing program with respect to the predicate p and C be a p-reducing clause with respect to all arguments. In the remainder of this section, we consider the method to combine P with C. Then, it is our purpose to eliminate the infinite branches of proof trees of $P \cup \{C\}$.

First, we introduce the following transformation of a clause C for a program P.

Definition 5.6 *Let P be a program and C be a clause $A \leftarrow B_1, \cdots, B_l$. Then, loop-elimination of C for P, denoted by $le(C, P)$, is a clause which is replaced the predicate symbol q in B_i $(1 \leq i \leq l)$ appearing in some head of P by the predicate true_q.*

Then, the following lemma holds.

Lemma 5.2 *Let P be a program $\{p(t_1, \cdots, t_n) \leftarrow D_1, \cdots, D_m\}$ and C be a clause. If P is head-reducing with respect to the predicate p and C is $pred(head(C))$-reducing with respect to all arguments, then $P \cup \{le(C, P)\}$ is head-reducing with respect to the predicate symbol p.*

Proof. Suppose that C is a clause $A \leftarrow B_1, \cdots, B_l$.

If any D_j and A are not unifiable, then the result trivially holds. Suppose $D_j\theta = A\theta$. Let $le(C, P)$ be loop-elimination $A \leftarrow B'_1, \cdots, B'_l$ of C for P. Then, $rec(P \cup \{le(C, P)\}, p)$ is constructed in the following way:

$$p(t_1, \cdots, t_n)\theta \leftarrow D_1\theta, \cdots, D_{j-1}\theta, (B'_1\theta, \cdots, B'_l\theta), D_{j+1}\theta, \cdots, D_m\theta.$$

Then, $RP(P \cup \{le(C, P)\}, p) = \{rec(P \cup \{le(C, P)\}, p), le(C, P)\}$. By Definition 5.6, the predicate p appearing in the bodies of $le(C, P)$ is replaced by *true_p*. Then, the predicate p does not appear in the part $(B'_1\theta, \cdots, B'_l\theta)$ in the body of $rec(P \cup \{le(C, P)\}, p)$. Furthermore, since C is $pred(A)$-reducing with respect to all arguments, for B_i such that $pred(A) = pred(B'_i)$, $|t_k\theta| > |s_k\theta|$ for any argument's term t_k and s_k $(1 \leq k \leq n)$ of A and B'_i. For B'_i such that $pred(A) \neq pred(B'_i)$, any definition clauses of $pred(B'_i)$ does not appear in $RP(P \cup \{le(C,P)\},p)$. Hence, $RP(P \cup \{le(C,P)\},p)$ is head-reducing with respect to p. ∎

If a clause C is not $pred(A)$-reducing with respect to all arguments, then there exists the following counterexample of Lemma 5.2.

Let P_5 be a program $\{p(f(X)) \leftarrow q(X,Y)\}$ and C_5 be a clause $q(X,f(Y)) \leftarrow q(f(X),Y)$. Then, P_5 is head-reducing with respect to the predicate p, but C_5 is not q-reducing with respect to the first argument. Note that $le(C_5,P_5)$ is equal to C_5 itself. Then, for the goal $\leftarrow p(f^2(a))$, the derivations of $P_5 \cup \{C_5\} \cup \{\leftarrow p(f^2(a))\}$ are infinite.

The next theorem claims that, by loop-elimination, we can choose the several programs whose union is head-reducing. In other words, rule-finding abduction for $P \cup \{le(C,P)\}$ terminates.

Theorem 5.2 *Let P be a program and C be a clause. If P is head-reducing with respect to the predicate p, P includes the definition clause of p, and C is $pred(head(C))$-reducing with respect to all arguments, then $P \cup \{le(C,P)\}$ is also head-reducing with respect to the predicate symbol p.*

Proof. The result is proven by mathematical induction on the number $|P|$ of clauses in P. If the number is 1, that is, $|P| = 1$, then Lemma 5.2 implies the result.

Suppose that the result is true for $|P| = n$, and consider the result for $|P| = n+1$. Let P be a program $\{C_1, \cdots, C_n\}$ and P' be a program $P \cup \{C_{n+1}\}$. Let $C, le(C,P)$ and $le(C,P')$ be the following clauses:

$$C = A \leftarrow B_1, \cdots, B_l,$$
$$le(C,P) = A \leftarrow B'_1, \cdots, B'_l,$$
$$le(C,P') = A \leftarrow B''_1, \cdots, B''_l.$$

By the induction hypothesis, all of programs $P, P \cup \{le(C,P)\}$, and P' are head-reducing with respect to p. If C_{n+1} is not applied to the construction of $rec(P',p)$, then $rec(P', p) = rec(P, p)$, and the result holds by the induction hypothesis.

Suppose that C_{n+1} is applied to the construction of $rec(P', p)$. Then, there exists an index i such that $head(C_{n+1})$ is unifiable with an atom in the body of C_i. Let C_i and C_{n+1} be the following clauses:

$$C_i = D \leftarrow E_1, \cdots, E_j, \cdots, E_m,$$
$$C_{n+1} = F \leftarrow G_1, \cdots, G_k, \cdots, G_f.$$

Then, the recursive definition $rec(P' \cup \{le(C, P')\}, p)$ is constructed in the following way: Suppose that the following clause is an intermediate clause in constructing the recursive definition:

$$p(t_1, \cdots, t_h) \leftarrow A_1, \cdots, D', \cdots, A_g.$$

Since C_{n+1} is applied to the construction of $rec(P', p)$, suppose that $D'\sigma = D\sigma$. Then, by application of C_i, we obtain the following clause:

$$p(t_1, \cdots, t_h)\sigma \leftarrow A_1\sigma, \cdots, (E_1\sigma, \cdots, E_j\sigma, \cdots, E_m\sigma), \cdots, A_g\sigma.$$

Since $head(C_{n+1})$ is unifiable with an atom in the body of C_i, suppose that $E_j\sigma\theta = F\theta$. Then, by application of C_{n+1}, we also obtain the following clause:

$$\begin{aligned} & p(t_1, \cdots, t_h)\sigma\theta \leftarrow \\ & \quad A_1\sigma\theta, \cdots, (E_1\sigma\theta, \cdots, (G_1\theta, \cdots, G_k\theta, \cdots, G_f\theta), \cdots, E_m\sigma\theta), \cdots, A_g\sigma\theta. \end{aligned}$$

For any index k $(1 \leq k \leq f)$, if $G_k\theta$ and A are not unifiable, then, by the induction hypothesis, $P \cup \{le(C, P)\}$ is head-reducing with respect to p. Hence, $P' \cup \{le(C, P')\}$ is also head-reducing with respect to p.

Otherwise, suppose that there exists a unifier λ for $G_k\theta$ and A. Then, $G_k\theta\lambda = A\lambda$. The recursive definition $rec(P' \cup \{le(C, P')\}, p)$ is also constructed in the following way:

$$\begin{aligned} & p(t_1, \cdots, t_h)\sigma\theta\lambda \leftarrow \\ & \quad A_1\sigma\theta\lambda, \cdots, (E_1\sigma\theta\lambda, \cdots, \\ & \qquad (G_1\theta\lambda, \cdots, (B''_1\lambda, \cdots, B''_l\lambda), \cdots, G_f\theta\lambda), \\ & \qquad\quad \cdots, E_m\sigma\theta\lambda), \cdots, A_g\sigma\theta. \end{aligned}$$

By the definition of $le(C, P')$ and by the construction of $rec(P' \cup \{le(C, P')\}, p)$, B_i'' and any head of the clauses in P' are not unifiable. Consequently, for some substitution μ, the recursive definition $rec(P' \cup \{le(C, P')\}, p)$ is constructed as follows:

$$p(t_1, \cdots, t_h)\mu \leftarrow H_1\mu, \cdots, (B_1''\mu, \cdots, B_l''\mu), \cdots, H_s\mu.$$

Since P' is head-reducing with respect to p by the induction hypothesis, an atom $H_r\mu$ except $B_i''\mu$ $(1 \leq i \leq l)$ satisfies the conditions that $RP(P' \cup \{le(C, P')\}, p)$ is head-reducing with respect to p. Furthermore, for any atom $B_i''\mu$ $(1 \leq i \leq l)$, the definition clause of $pred(B_i'')$ does not appear in P' by the definition of $le(C, P'')$. On the other hand, C, so $le(C, P'')$, is $pred(A)$-reducing with respect to all arguments. Hence, $RP(P' \cup \{le(C, P')\}, p)$, which includes $rec(P' \cup \{le(C, P')\}, p)$, is head-reducing with respect to p.

Therefore, $P' \cup \{le(C, P')\}$ is head-reducing with respect to p. ∎

The *loop-elimination of P' for P*, denoted by $le(P', P)$, is the set of $le(C, P)$, where C is a clause in P'. In other words,

$$le(P', P) = \{le(C, P) \mid C \in P'\}.$$

By Theorem 5.2, when the programs P and P' are given, rule-finding abduction for $P \cup le(P', P)$ also terminates.

Example 5.4 *Let P_6 and P_7 be the following programs:*

$$P_6 = \left\{ \begin{array}{l} p(f(X)) \leftarrow q(X) \\ q(X) \leftarrow p(X) \end{array} \right\},$$

$$P_7 = \left\{ \begin{array}{l} p(X) \leftarrow q(X) \\ q(X) \leftarrow r(X), s(X). \end{array} \right\}.$$

Since the union $P_6 \cup P_7$ is not head-reducing with respect to the predicate p nor q, this program falls into an infinite loop for any ground goal with the predicates p and q.

On the other hand, after transforming P_7 to loop-elimination $le(P_7, P_6)$ of P_7 for P_6, then we obtain the following set of clauses:

$$le(P_7, P_6) = \left\{ \begin{array}{l} p(f(X)) \leftarrow q(X) \\ q(X) \leftarrow p(X) \\ p(X) \leftarrow true\_q(X) \\ q(X) \leftarrow r(X), s(X) \end{array} \right\}.$$

Furthermore, after transforming P_6 to loop-elimination $le(P_6, P_7)$ of P_6 for P_7, we also obtain the following set of clauses:

$$le(P_6, P_7) = \left\{ \begin{array}{l} p(f(X)) \leftarrow q(X) \\ q(X) \leftarrow true\_p(X) \\ p(X) \leftarrow q(X) \\ q(X) \leftarrow r(X), s(X) \end{array} \right\}.$$

By Theorem 5.2, rule-finding abduction for both $le(P_7, P_6)$ and $le(P_6, P_7)$ terminate for any ground goal.

5.5 Prolog Implementation

In rule-finding abduction, since we consider the several programs, the clause is given in the following form:

```
fact(World,clause(Head,Bodies)),
```

where the first argument `World` represents a program. Here, we give the programs on $w_1, w_2, \cdots$. A Prolog clause `fact(wi,clause(Head,Body))` means that the clause `Head`$\leftarrow$`Body` is an element of P_i.

The rule-finding abduction is realized by improving rule-selecting abduction as the following `rf_abd` program. Note that the third argument of `rf_abd` represents the set of programs and is returned as a list.

```
rf_abd(Goal,Leaves,World) :-
        fact(AnyWorld,clause(Goal,Clause)),member(AnyWorld,World),
        rf_abd(Clause,Leaves,World).
rf_abd((Goal1,Goal2),(Leaf1,Leaf2),World) :-
        !,rf_abd(Goal1,Leaf1,World),rf_abd(Goal2,Leaf2,World).
rf_abd(Leaf,Leaf,World) :- !.

member(X,[X|Y]) :- !.
member(X,[Y|Z]) :- member(X,Z) :- !.
```

We can apply the above program to examples in Section 5.1. For the first example, each clause of P_i ($1 \leq i \leq 3$) is given in the following forms:

```
fact(w1,clause(find(X,fossil_shell,Y),sea(Y))).
fact(w2,clause(find(X,fossil_shell,Y),used_to_be(Y,sea))).
fact(w2,clause(used_to_be(X,Y),be(X,Y))).
fact(w3,clause(find(X,fossil_shell,Y),move(fossil_shell,sea,Y))).
fact(w3,clause(move(fossil_shell,X,Y),slow_move(fossil_shell,X,Y))).
fact(w3,clause(slow_move(X,Y,Z),has_not_leg(X))).
fact(w3,clause(slow_move(X,Y,Z),has_not_wing(X))).
```

Then, for the surprising fact *find(i,fossil_shell,mountain)*, the results of rule-finding abduction are obtained as follows:

```
: ?- rf_abd(find(i,fossil_shell,mountain),X,W).
X = sea(mountain),                              %%% H1 %%%
W = [w1|_252] ;
X = be(mountain,sea),                           %%% H2 %%%
W = [w2|_254] ;
X = used_to_be(mountain,sea),                   %%% H3 %%%
W = [w2|_254] ;
X = has_not_leg(fossil_shell),                  %%% H4 %%%
W = [w3|_256] ;
X = has_not_wing(fossil_shell),                 %%% H5 %%%
W = [w3|_256] ;
X = slow_move(fossil_shell,sea,mountain),       %%% H6 %%%
W = [w3|_256] ;
X = move(fossil_shell,sea,mountain),            %%% H7 %%%
W = [w3|_256] ;
X = find(i,fossil_shell,mountain),              %%% H0 %%%
W = W ;
no
```

For the second example in Section 5.1, each clause of P_i $(4 \leq i \leq 6)$ is also given in the following forms:

```
fact(w4,clause(p(f(f(X))),(p(X),q(f(X))))).
fact(w5,clause(q(f(X)),(q(X),r(X,f(X))))).
fact(w6,clause(r(f(X),f(Y)),r(X,Y))).
```

Then, for the surprising fact $p(f^3(a))$, the results of rule-finding abduction are obtained as follows:

```
: ?- rf_abd(p(f(f(f(a)))),X,W).
X = p(f(a)),(q(a),r(a,f(a))),r(a,f(a)),         %%% K4 %%%
W = [w4,w5,w6|_566] ;
X = p(f(a)),(q(a),r(a,f(a))),r(f(a),f(f(a))),   %%% K2 %%%
W = [w4,w5|_386] ;
X = p(f(a)),q(f(a)),r(a,f(a)),                  %%% K5 %%%
```

```
W = [w4,w5,w6|_480] ;
X = p(f(a)),q(f(a)),r(f(a),f(f(a))),          %%% K3 %%%
W = [w4,w5|_386] ;
X = p(f(a)),q(f(f(a))),                       %%% K1 %%%
W = [w4|_282] ;
X = p(f(f(f(a)))),                            %%% K0 %%%
W = W ;
no
```

Furthermore, we can realize *breadth-first rule-finding* abduction in the following program `bfrf_abd` just as in Section 4.3:

```
bfrf_abd(Goal,Leaves,World,Depth) :-
        Depth > 0,
        fact(AnyWorld,clause(Goal,Clause)),
        member(AnyWorld,World),
        Depth1 is Depth-1,
        bfrf_abd(Clause,Leaves,World,Depth1).
bfrf_abd((Goal1,Goal2),(Leaf1,Leaf2),World,Depth) :-
        !,
        bfrf_abd(Goal1,Leaf1,World,Depth),
        bfrf_abd(Goal2,Leaf2,World,Depth).
bfrf_abd(Leaf,Leaf,World,0).
```

In Section 5.2, we discuss an abducible, which is similar to an operationality criterion. We can realize the abducible in the following program `rf_abd_prd`. The predicate `rf_abd_prd` returns the choice of programs as the third argument and the abducible as the fourth argument, if all the proof trees of a program for a goal are finite.

```
rf_abd_prd(Goal,Leaves,World,OC) :-
        fact(AnyWorld,clause(Goal,Clause)),
        member(AnyWorld,World),
        rf_abd_prd(Clause,Leaves,World,OC).
rf_abd_prd((Goal1,Goal2),(Leaf1,Leaf2),World,OC) :-
        !,
        rf_abd_prd(Goal1,Leaf1,World,OC),
        rf_abd_prd(Goal2,Leaf2,World,OC).
rf_abd_prd(Goal,Goal,World,OC) :-
        functor(Goal,Pred,_),
        Pred \= (,),
        member(Pred,OC).
```

We can also realize the loop-elimination in the following programs. The predicate `le_main` returns a loop-eliminated clause for a given program as the second argument. (Full Prolog version will be described in Appendix of this thesis.)

```
le_main(fact(World,clause(Head,Bodies)),fact(World,clause(Head,NewBody))) :-
        setof(Y,le(fact(World,clause(Head,Bodies)),Y),List),
        isort(List,List2),
        List2 = [MaxFact|List3],
        MaxFact = fact(World,clause(Head,NewBody)).
```

By Theorem 5.2, the program `rf_abd` terminates for any surprising fact of p and q. Incorporating loop-elimination with rule-finding abduction, we can give the following results.

Let P_1 and C be the following program and clause:

$$P_1 = \left\{ \begin{array}{l} p(f(X)) \leftarrow q(X) \\ q(f(X)) \leftarrow p(f(X)) \end{array} \right\}$$

$$C_1 = p(X) \leftarrow q(X), q(g(X))$$

Then, $P_1 \cup \{C_1\}$ is not head-reducing with respect to the predicate p nor q. On the other hand, we obtain the following loop-elimination $le(C_1, P_1)$ of C_1 in P_1:

$$le(C_1, P_1) = p(X) \leftarrow true\_q(X), q(g(X)).$$

Here, $P_1 \cup \{le(C_1, P_1)\}$ is head-reducing with respect to the predicate p.

In order to apply the program `le_main` to the above P_1 and C_1, let P_1 be the following clauses:

```
fact(w1,clause(p(f(X)),q(X))).
fact(w1,clause(q(f(X)),p(f(X)))).
```

When we give the clause `fact(w2,clause(p(X),(q(X),q(g(X)))))` in the first argument in the predicate `le_main`, then the predicate `le_main` returns a loop-elimination of the first argument's clause as the second argument. By the predicate `assert`, we add it to the program P_1. Finally, we obtain the hypothesis of a surprising fact $p(f^3(a))$ as follows:

```
: ?- le_main(fact(w2,clause(p(X),(q(X),q(g(X))))),Y),
           assert(Y),rf_abd(p(f(f(f(a)))),H,W).
X = f(_594),
Y = fact(w2,clause(p(f(_594)),(true(q,f(_594)),q(g(f(_594)))))),
H = q(a),
W = [w1|_1300] ;
```

```
X = f(_594),
Y = fact(w2,clause(p(f(_594)),(true(q,f(_594)),q(g(f(_594)))))),
H = true(q,f(a)),q(g(f(a))),
W = [w1,w2|_1494] ;
X = f(_594),
Y = fact(w2,clause(p(f(_594)),(true(q,f(_594)),q(g(f(_594)))))),
H = p(f(a)),
W = [w1|_1300] ;
X = f(_594),
Y = fact(w2,clause(p(f(_594)),(true(q,f(_594)),q(g(f(_594)))))),
H = q(f(a)),
W = [w1|_1300] ;
X = f(_594),
Y = fact(w2,clause(p(f(_594)),(true(q,f(_594)),q(g(f(_594)))))),
H = true(q,f(f(a))),q(g(f(f(a)))),
W = [w1,w2|_1422] ;
X = f(_594),
Y = fact(w2,clause(p(f(_594)),(true(q,f(_594)),q(g(f(_594)))))),
H = p(f(f(a))),
W = [w1|_1300] ;
X = f(_594),
Y = fact(w2,clause(p(f(_594)),(true(q,f(_594)),q(g(f(_594)))))),
H = q(f(f(a))),
W = [w1|_1300] ;
X = f(_594),
Y = fact(w2,clause(p(f(_594)),(true(q,f(_594)),q(g(f(_594)))))),
H = true(q,f(f(f(a)))),q(g(f(f(f(a))))),
W = [w2|_1324] ;
X = f(_594),
Y = fact(w2,clause(p(f(_594)),(true(q,f(_594)),q(g(f(_594)))))),
H = p(f(f(f(a)))),
W = W ;
no
```

Note that, by loop-elimination, we obtain the atoms `true(q,a)`, `true(q,f(a))`, and `true(q,f(f((a))))` in the above hypotheses. Then, we can interpret them as $q(a)$, $q(f(a))$, and $q(f^2(a))$ respectively.

5.6 Rule-Finding Abduction with Analogy

In the previous sections, we have discussed rule-finding abduction. In these discussions, we assumed that the found rules are not ground. If all of the rules in programs are ground, then we cannot apply rule-finding abduction to them, because the application of rule-finding abduction is based on the unification. Consider the following example.

Example 5.5 *Let $p(a)$ be a surprising fact, and P_1 and P_2 be the following programs:*

$$P_1 = \phi,$$
$$P_2 = \{p(b) \leftarrow q(b)\}.$$

By rule-finding abduction for $P_1 \cup P_2$, we can propose only a trivial hypothesis $\{p(a)\}$ of $p(a)$.

In Example 5.5, if we introduce the analogy such that a is analogous to b, then we can obtain a hypothesis $\{q(a)\}$. Hence, in this section, we discuss analogical reasoning from the viewpoint of rule-finding abduction.

Thargad [Tha88] and Duval [Duv91] have tried to discuss abduction and analogy in the same framework. Thagard [Tha88] has applied Kuhn's philosophy of science [Kuh70] to computer science, and dealt with *analogical abduction*, which is one of the methods of discovery. On the other hand, Duval [Duv91] has also dealt with abduction and analogy in the framework of explanation-based generalization. However, in such researches, the relationship between abduction and analogy are not clear, since their concepts of abduction and analogy are ambiguous.

In this thesis, we adopt the formulation of analogical reasoning by Haraguchi and Arikawa [Har85, HaA86, HiA94a, HiA94b]. It is based on the analogy between Herbrand universes of a base program P_b and that of a target program P_t.

Let P_b be a base program, P_t be a target program, and α be a ground atom. Then, a proof tree of α for P_b (*resp.*, P_t) is denoted by T^b_α (*resp.*, T^t_α). The leaves of T^b_α (*resp.*, T^b_α) is denoted by $leaves(T^b_\alpha)$ (*resp.*, $leaves(T^t_\alpha)$).

In the formulation of analogical reasoning [Har85, HaA86, HiA94a, HiA94b], it is natural to consider that a surprising fact is given in a target program P_t, not in a base program P_b. Hence, in the definitions in this section, a surprising fact is also given in a target program P_t.

In this section, it is our purpose to formulate rule-finding abduction incorporating with analogical reasoning. In other words, we deal with the concept of analogy in order to extend rule-finding abduction. Such the abduction is called *rule-finding abduction*

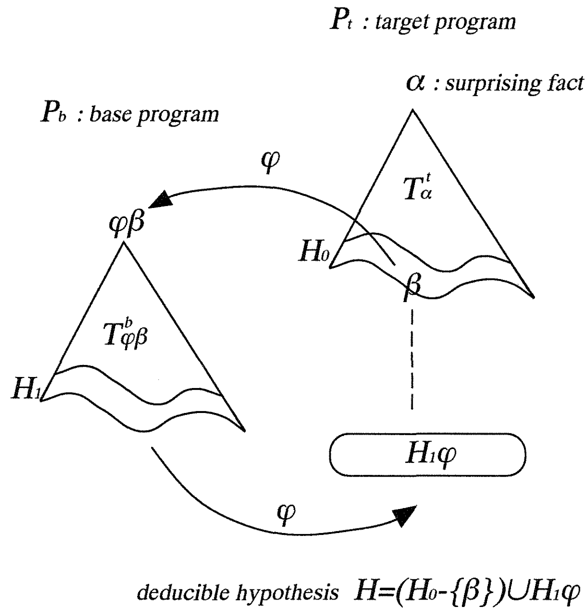

Figure 5.3: Deducible hypothesis, where $P_b \vdash \varphi\beta$

with analogy.

First, we formulate a simple hypothesis of rule-finding abduction with analogy as follows:

Definition 5.7 *Let $P_b = R_b \cup F_b$ and P_t be programs, and α be a surprising fact with respect to P_t. Let $\varphi \subseteq U(P_b) \times U(P_t \cup \{\alpha\})$ be a partial identity. Then, a set H of atoms is a* simple hypothesis of α for P_b and P_t with analogy φ *if H satisfies the following condition:*

$$R_b\varphi \cup P_t \cup H \vdash \alpha.$$

However, for the simple hypothesis, the variables in P_b or P_t are substituted by ground terms in $P_b \cup P_t$. In order to solve this problem, we introduce another hypothesis of rule-finding abduction with analogy, called a *deducible hypothesis*, as follows:

Definition 5.8 *Let α be a surprising fact, that is, $P_t \not\vdash \alpha$, and $\varphi \subseteq U(P_b) \times U(P_t \cup \{\alpha\})$ be a partial identity. Suppose that $P_t \cup H_0 \vdash \alpha$. For any $\beta \in H_0$, if $P_b \vdash \varphi\beta$, then $H = (H_0 - \{\beta\}) \cup H_1\varphi$, where H_1 is the set of nodes in $T^1_{\varphi\beta}$. Then, H is called a* deducible hypothesis of α for P_b and P_t with analogy φ.

deducible hypothesis H_{leaves} /* based on the leaves of the proof tree */
input P_b, P_t : programs,
α : a ground atom,
$\varphi \subseteq U(P_b) \times U(P_t \cup \{\alpha\})$: a partial identity
output H_{leaves} : a deducible hypothesis

$H := leaves(T^t_\alpha)$;
while there exists a ground atom $\beta \in H$ such that $P_b \vdash \varphi\beta$ **do**
$H_1 := leaves(T^b_{\varphi\beta})$;
$H' := H_1\varphi$;
$H'' := \phi$;
while $H' = \phi$ **do**
choose $\gamma \in H'$;
$H'' := H'' \cup leaves(T^t_\gamma)$;
$H' := H' - \{\gamma\}$;
end
$H_{leaves} := (H - \{\beta\}) \cup H''$;
end
output H_{leaves}
end

Figure 5.4: Algorithm to construct a deducible hypothesis H_{leaves}

Figure 5.3 illustrates the formulation of deducible hypotheses. It is clear that, if H is a deducible hypothesis, then it is a simple hypothesis. Furthermore, the variables in P_b (*resp.*, P_t) are substituted by only ground terms in P_b (*resp.*, P_t).

It arises a problem how to choose the deducible hypothesis H. For a proof tree, if we choose any combination of nodes in $T^b_{\varphi\beta}$, then it is very difficult to obtain all deducible hypotheses. Even if we choose two sets of nodes in $T^b_{\varphi\beta}$, after the above procedure to obtain deducible hypotheses in n times, the number of deducible hypotheses is at most 2^n. Hence, for a proof tree, we adopt the choice of only one hypothesis in $T^b_{\varphi\beta}$.

In order to construct a deducible hypothesis concretely, we introduce a deducible hypothesis H_{leaves} which is based on the leaves of a proof tree. A deducible hypothesis H_{leaves} is obtained by the algorithm in Figure 5.4. In this algorithm, H_0 and H_1 in Figure 5.3 are the leaves of T^t_α and $T^t_{\varphi\beta}$, respectively.

Let P_b and P_t be programs, and α be a ground atom. By P^b_α (*resp.*, P^t_α), we denote the set of clauses which are applied to a proof tree T^b_α (*resp.*, T^t_α) of α in P_b (*resp.*, P_t). Then, the following theorem holds:

Theorem 5.3 *Let P_b and P_t be programs, and α be a ground atom. Let $\varphi \subseteq U(P_b) \times U(P_t \cup \{\alpha\})$ be a partial identity and $leaves(T^t_\alpha)$ be a set $\{\beta_j \mid 1 \leq j \leq k\}$ of leaves in T^t_α. Suppose that $P_t \not\vdash \alpha$. If $P_b \vdash \varphi\beta_j$ $(1 \leq j \leq k)$, then $\{\bigcup_{j=1}^{k} P^b_{\varphi\beta_j}\varphi\} \cup P^t_\alpha \vdash \alpha$.*

Proof. For β_j, H^j_{leaves} denotes the deducible hypothesis of β_j. Then, $H^j_{leaves} \subseteq P^b_{\varphi\beta_j}\varphi$, and the clauses in $P^b_{\varphi\beta_j}\varphi$ are applied to P_t. Hence,

$$P^b_{\varphi\beta_j}\varphi \cup P^t_\alpha \cup \{leaves(T^t_\alpha) - \{\beta_j\}\} \vdash \alpha.$$

By applying the above consideration to β_j for $1 \leq j \leq k$, we can obtain the result. ∎

Since a deducible hypothesis is included in $\{\bigcup_{j=1}^{k} P^b_{\varphi\beta_j}\varphi\}$, Theorem 5.3 means that a deducible hypothesis is correct in the sense of analogical reasoning.

In this formulation, we assume that a partial identity φ is given in advance. This assumption is unreasonable. In analogical reasoning, an analogy φ is not given in advance, and it is a main problem to detect the φ. Then, in order to obtain an analogy φ while constructing H_{leaves}, we adopt the concept of *partially isomorphic generalizations*, which has been introduced by Hirowatari and Arikawa [HiA94b]. They have regarded an analogy as a partial function from $U(P_b)$ to $U(P_t \cup \{\alpha\})$, not a partial identity. They have also reduced the problem of the detection of partial identity to the unification of partially isomorphic generalization as Theorem 2.5. In rule-finding abduction with analogy, we also follow this consideration.

Consider the following examples.

Example 5.6 *Let P_3 be the following base program:*

$$P_3 = \left\{ \begin{array}{ll} C_1: & p(a, b) \\ C_2: & p(f(X), b) \leftarrow p(X, b) \end{array} \right\}.$$

Let α be a surprising fact $p(f^2(c), f^2(d))$ with respect to an empty target program.

1. *The partially isomorphic generalization PC_1 of C_1 is as follows:*

$$PC_1: \ p(X,Y).$$

Since $head(PC_1)$ and α are unifiable, and $p(a,b)$ is provable in P_b, we obtain the following deducible hypothesis H_1:

$$H_1 = \{p(f^2(c), f^2(d))\}.$$

There exist substitutions $\theta_1 = \{X := a, Y := b\}$ and $\theta_2 = \{X := f^2(c), Y := f^2(d)\}$ such that $head(PC_1)\theta_1 = p(a,b)$, and $head(PC_1)\theta_2 = \alpha$. Then, by Theorem 2.5, there exists the analogy $\varphi_1 \subseteq U(P_3) \times U(\{\alpha\})$ which is obtained by:

$$\varphi_1 = \{\langle t, s\rangle \mid X := t \in \theta_1, X := s \in \theta_2\}.$$

Hence, the analogy φ_1 for H_1 is a set $\{\langle a, f^2(c)\rangle, \langle b, f^2(d)\rangle\}$.

2. *The partially isomorphic generalization PC_2 of C_2 is as follows:*

$$PC_2: \ p(f(X), Y) \leftarrow p(X,Y).$$

Since $head(PC_2)$ and α are unifiable, we obtain the following candidate K_1 of deducible hypotheses:

$$K_1 = \{p(f(c), f^2(d))\}.$$

For an element $p(f(c), f^2(d))$ of K_1, we also continue the above discussion 1 and 2. Then, from K_1, we obtain the following deducible hypothesis H_2 and the analogy $\varphi_2 \subseteq U(P_3) \times U(\{\alpha\})$:

$$H_2 = \{p(f(c), f^2(d))\}, \ \varphi_2 = \{\langle a, f(c)\rangle, \langle b, f^2(d)\rangle\}.$$

Also we obtain the following candidate K_2 of deducible hypotheses:

$$K_2 = \{p(c, f^2(d))\}.$$

$$\begin{array}{ccc}
p(a,b) & p(f(a),b) & p(f^2(a),b) \\
H_1 & | & | \\
 & p(a,b) & p(f(a),b) \\
 & H_2 & | \\
 & & p(a,b) \\
 & & H_3
\end{array}$$

$$\begin{array}{ccc}
\varphi_1 & \varphi_2 & \varphi_3 \\
\langle a, f^2(c)\rangle & \langle a, f(c)\rangle & \langle a, c\rangle \\
\langle b, f^2(d)\rangle & \langle b, f^2(d)\rangle & \langle b, f^2(d)\rangle
\end{array}$$

Figure 5.5: Deducible hypotheses H_1, H_2, and H_3, and proof trees

For an element $p(c, f^2(d))$ of K_2, head(PC_1) and $p(c, f^2(d))$ are unifiable, while head(PC_2) and $p(c, f^2(d))$ are not. Furthermore, $p(a,b)$ is provable in P_t. Then, from K_2, we obtain the following deducible hypothesis H_3 and the analogy $\varphi_3 \subseteq U(P_3) \times U(\{\alpha\})$:

$$H_3 = \{p(c, f^2(d))\},\ \varphi_3 = \{\langle a, c\rangle, \langle b, f^2(d)\rangle\}.$$

Figure 5.5 illustrates three deducible hypotheses H_1, H_2, and H_3, and proof trees which are corresponding to H_i. For each proof tree, the root node is analogous to a surprising fact α under the analogy φ_i, and the atom which is analogous to leaf node under φ_i is corresponding to a deducible hypothesis H_i.

Example 5.7 *For P_3, if a surprising fact is a ground atom $p(f^2(c), c)$, then we obtain the following deducible hypotheses H_j and analogies φ_j $(4 \leq j \leq 6)$:*

$$\begin{aligned}
H_4 &= \{p(a,b)\}, & \varphi_4 &= \{\langle a, f^2(c)\rangle, \langle b, c\rangle\}, \\
H_5 &= \{p(f(a),b)\}, & \varphi_5 &= \{\langle a, f(c)\rangle, \langle b, c\rangle\}, \\
H_6 &= \{p(f^2(a),b)\}, & \varphi_6 &= \{\langle a, c\rangle, \langle b, c\rangle\}.
\end{aligned}$$

Note that φ_j $(4 \leq j \leq 6)$ is a function from $U(P_3)$ to $U(\{\alpha\})$.

On the other hand, let P_4 be the following base program:

$$P_4 = \left\{ \begin{array}{l} C_3 :\ p(a,a) \\ C_4 :\ p(f(X),a) \leftarrow p(X,a) \end{array} \right\}.$$

If either $p(f^2(c), f(d))$ or $p(f^2(c), d)$ is given as a surprising fact β, then there exist no deducible hypotheses. Because we regard an analogy φ as a function from $U(P_4)$ to $U(\{\beta\})$, and, for the above surprising facts and the base program P_4, there exist no such the analogies. On the other hand, let $p(f^2(c), c)$ be a surprising fact. Then, we obtain the following deducible hypothesis H_7 and the analogy φ_7:

$$H_7 = \{p(c, c)\},\ \varphi_7 = \{\langle a, c\rangle\}.$$

We can realize rule-finding abduction with analogy as the Prolog program in Figure 5.6. The program `ab_ana` computes a deducible hypothesis H_{leaves} and an analogy. Note that, this program assumes that a target program is empty, that is, only the first while-loop in Figure 5.4 is realized. (Full Prolog version will be described in Appendix of this thesis.)

The predicate `ab_ana` in Figure 5.6 is given a surprising fact as its first argument. Then, it returns a deducible hypothesis as its second argument, a pairing as its third argument, and a world as its fourth argument. The predicate `analogy` returns the pairing as the third argument between the base rule given as the second argument and the target rule given as the first argument. The predicate `provable` checks provability in a base program P_b, and, if so, then it proposes a deducible hypothesis as the third argument. The predicate `pig_rule` returns the partially isomorphic generalization `PG:-PGs` of the rule `BG:-BGs` as the second argument.

For a clause C, by $pig(C)$, we denote the partially isomorphic generalization of C. For a program P, by $pig(P)$, we denote the set $\{pig(C) \mid C \in P\}$. The termination of the Prolog program `ab_ana`, which is rule-finding abduction with analogy, is characterized in the following theorem as the corollary of Theorem 4.1.

Corollary 5.2 *Let $P_b = R_b \cup F_b$ and P_t be programs and p be a predicate symbol. If $pig(R_b) \cup P_t$ is head-reducing with respect to the predicate p, then all the derivations of $pig(R_b) \cup P_t \cup \{\leftarrow p(s_1, \cdots, s_n)\}$ are finite for any ground atom $p(s_1, \cdots, s_n)$.*

In the program `ab_ana`, a target program P_t is assumed an empty set. Hence, by Corollary 5.2 and Theorem 2.5, if $pig(R_b)$ is head-reducing with respect to the predicate

```
ab_ana(TG,TG,Pair,WorldTarget):-
        functor(TG,Pred,Arity),functor(BG,Pred,Arity),
        world(WorldBase,WorldTarget),
        fact(WorldBase,(BG:-true)),
        analogy((TG:-true),(BG:-true),Pair).

ab_ana(TG,TGs,Pair,WorldTarget):-
        functor(TG,Pred,Arity),functor(BG,Pred,Arity),
        provable(TG,BG,TGs,BGs,WorldTarget),
        not TG==TGs,
        analogy((TG:-TGs),(BG:-BGs),Pair).

provable(TG,BG,TL,BL,WorldTarget) :-
        rule(TG,BG,TGs,BGs,WorldTarget),
        provable(TGs,BGs,TL,BL,WorldTarget).

provable((TG,TGs),(BG,BGs),(TL,TLs),(BL,BLs),WorldTarget):-
        provable(TG,BG,TL,BL,WorldTarget),
        provable(TGs,BGs,TLs,BLs,WorldTarget).

provable(TG,BG,TG,BG,WorldTarget):-
        world(WorldBase,WorldTarget),fact(WorldBase,(BG:-true)),!.

rule(TG,BG,TGs,BGs,WorldTarget) :-
        world(WorldBase,WorldTarget),
        fact(WorldBase,(BG:-BGs)),
        not BGs=true,
        pig_rule((BG:-BGs),(PG:-PGs)),
        copy((PG:-PGs),(TG:-TGs)).
```

Figure 5.6: Program ab_ana

p, then, for any surprising fact $p(s_1, \cdots, s_n)$, the goal

$$\texttt{?- ab\_ana(}p(s_1, \cdots, s_n)\texttt{,X,P,W)}$$

terminates, and returns the deducible hypotheses as its second argument and the pairings as its third argument.

For the program in Figure 5.6, if the proof tree of a base program is obtained, then the computational complexity to obtain the deducible hypothesis is characterized as the following theorem:

Theorem 5.4 *Let P_b be a base program and α be a ground atom. For a given proof tree in P_b, a ground atom α' is a root and H' is the set of leaves. Suppose that $|\alpha| = k$ and $|H'| = l$. Then, a deducible hypothesis is computed in $O(k^3 l)$ time.*

Proof. By Theorem 2.4, for a root α', the partially isomorphic generalization β of α' is computed in $O(k^3)$. Since α is ground, whether or not β and α are unifiable is determined in $O(k)$. If β and α are unifiable, then, by Theorem 2.5, an analogy φ can be computed simultaneously. The time complexity to apply this φ to H' is $O(l)$. Hence, a deducible hypothesis $H'\varphi$ is computed in $O(k^3 l)$ time. ∎

The base program in Example 5.6 is represented as follows:

```
fact(w1,(p(a,b):-true)).
fact(w1,(p(f(X),b):-p(X,b))).
world(w1,w2).
```

Here, the atom `world(w1,w2)` represents that `w1` is a base program and `w2` is a target program. In this program, a target program `w2` is empty. Then, we obtain the following results:

```
: ?- ab_ana(p(f(f(c)),f(f(d))),X,P,W).
X = p(f(f(c)),f(f(d))),
P = [a--f(f(c)),b--f(f(d))],
W = w2 ;
X = p(c,f(f(d))),
P = [a--c,b--f(f(d))],
W = w2 ;
X = p(f(c),f(f(d))),
```

```
P = [a--f(c),b--f(f(d))],
W = w2 ;
no
: ?- ab_ana(p(c,d),X,P,W).
X = p(c,d),
P = [a--c,b--d],
W = w2 ;
no
```

Note that the solution `[a--f(f(c)),b--f(f(d))]` of the third argument for the program `ab_ana` means the pairing $\{\langle a, f^2(c)\rangle, \langle b, f^2(d)\rangle\}$.

On the other hand, the base program in Example 5.7 is represented as follows:

```
fact(w1,(p(a,a):-true)).
fact(w1,(p(f(X),a):-p(X,a))).
world(w1,w2).
```

Then, we obtain the following results:

```
: ?- ab_ana(p(f(f(c)),f(d)),X,P,W).
no
: ?- ab_ana(p(f(f(c)),d),X,P,W).
no
: ?- ab_ana(p(f(f(c)),c),X,P,W).
X = p(c,c),
P = [a--c],
W = w2 ;
no
: ?- ab_ana(p(f(f(c)),f(c)),X,P,W).
X = p(f(c),f(c)),
P = [a--f(c)],
W = w2 ;
no
```

Since we regard an analogy as a function, for the first two goals, we obtain no analogies. Then, we also obtain no deducible hypotheses.

Chapter 6

Rule-Generating Abduction

"The ideal reasoner", he remarked, "would, when he has once been shown a single fact in all its bearings, deduce from it not only all the chain of events which led up to it, but also all the results which would follow from it."

— *'The Five Orange Pips'*
"The Adventures of Sherlock Holmes"

Let P be a definite program and C be a surprising fact. The *rule-generating abduction* is a type of abduction which generates a rule in P and proposes a hypothesis to explain the surprising fact C. An inference schema of rule-generating abduction is described by the following three steps:

1. A surprising fact C is observed.
2. A rule $C \leftarrow A$ is *generated* in P.
3. A hypothesis A is proposed.

For a surprising fact C, we regard the above inference schema as the following one by identifying A with the set $\{A\}$ of atoms:

1. A ground atom C such that $P \not\vdash C$ is given.
2. A rule $C' \leftarrow A'_1, \cdots, A'_n$ is *generated* in P, where $C'\theta = C$ and $A'_i\theta = A_i$.
3. A hypothesis $\{A_1, \cdots, A_n\}$ is proposed. Then, $P \cup \{A_1, \cdots, A_n\} \vdash C$.

For the above inference schema, even in propositional logic, there exist infinitely many hypotheses and rules. In definite programs, if the class of programs is not restricted to some subclass, then there are also infinitely many meaningless hypotheses. Hence, we introduce the subclass of definite programs for rule-generating abduction. Furthermore, in rule-generating abduction, a surprising fact is given only once. Hence, we also need to generalize one ground atom.

In this chapter, we investigate rule-generating abduction for logic programming. In Section 6.1, we introduce the subclass of head-reducing programs in Chapter 4. In Section 6.2, we introduce the concept of a *safe generalization*, and discuss the properties of it. In Section 6.3, we investigate the number of hypotheses by rule-generating abduction. In Section 6.4, we design an efficient algorithm of rule-generating abduction for the introduced class by using safe generalization. In Section 6.6, we realize this algorithm as a Prolog program, and consider several examples.

This chapter is based on the papers [Hir94b, Hir94c].

6.1 Weakly 2-Reducing Programs

In this chapter, we deal with the following class of programs:

$$\left\{ \begin{array}{l} p(t_1, \cdots, t_n) \leftarrow p(s_1, \cdots, s_n) \\ p(u_1, \cdots, u_n) \end{array} \right\}.$$

In Chapter 4, we have introduced the class of head-reducing programs. As mentioned in Chapter 4, for a head-reducing program P and a ground atom α, all the derivations of $P \cup \{\leftarrow \alpha\}$ are finite. In this section, in order to characterize such programs as we mentioned above, we give the definitions as the special case of head-reducing programs.

In this chapter, we deal with lists as terms. Then, we assume that a list constructor $[\,|\,]$ and an empty list $[\,]$ are included in first order language $\mathcal{L}$. For a term t, $|t|$ denotes the length of t. In particular, for a list l, the length $|l|$ of l is defined as follows: $|l| = 1$, if l is an empty list $[\,]$. Otherwise $|l| = n + 1$, if t is a list $[a|list]$ and $|list| = n$.

For rule-generating abduction, we introduce some classes of definite programs. Let C be a definite clause $p(t_1, \cdots, t_n) \leftarrow p(s_1, \cdots, s_n)$. Suppose that a definite program has the form of $\{C\}$. By the discussion in Chapter 4, $\{C\}$ is head-reducing with respect to the predicate p if and only if C is p-reducing with respect to some argument. In this section, a clause C is called *head-reducing* if there exists an argument i such that $|t_i\theta| > |s_i\theta|$ for any substitution θ. Furthermore, we also introduce the following definitions:

Definition 6.1 (Hirata [Hir93a, Hir93b], Yamamoto [Yam92]) *Let C be a definite clause $p(t_1, \cdots, t_n) \leftarrow p(s_1, \cdots, s_n)$.*

1. *C is weakly reducing if $|t_i\theta| \geq |s_i\theta|$ for any substitution θ and for any i.*
2. *C is weakly head-reducing if it is head-reducing and weakly reducing.*

In other words, a clause $p(t_1, \cdots, t_n) \leftarrow p(s_1, \cdots, s_n)$ is weakly head-reducing if $|t_i\theta| \geq |s_i\theta|$ for any i, and $|t_k\theta| > |s_k\theta|$ for at least one argument k and for any ground substitution θ.

There are many Prolog clauses for list processing such that any argument of the head has the form of either X or $[W|X]$, and the body has the form of Y. Then, we restrict the form of clause as follows: A clause $p(t_1, \cdots, t_n) \leftarrow p(s_1, \cdots, s_n)$ is *2-reducing*, if it is head-reducing, and t_i has the form of either X_i or $[W_i|X_i]$ and s_i has the form of Y_i for any i. In 2-reducing programs, Y_i is not necessarily equal to X_i. A clause $p(t_1, \cdots, t_n) \leftarrow p(s_1, \cdots, s_n)$ is *weakly 2-reducing* if it is weakly reducing and 2-reducing. In other words, a clause $p(t_1, \cdots, t_n) \leftarrow p(s_1, \cdots, s_n)$ is weakly 2-reducing if t_i has the form of either X_i or $[W_i|X_i]$ and s_i has the form of X_i for any i.

Example 6.1 *The following Prolog clauses in Sterling and Shapiro [SS86, SS94] are weakly 2-reducing:*

$$\begin{aligned}
&\mathit{member}(X, [W|Y]) \leftarrow \mathit{member}(X, Y),\\
&\mathit{prefix}([W|X], [W|Y]) \leftarrow \mathit{prefix}(X, Y),\\
&\mathit{suffix}(X, [W|Y]) \leftarrow \mathit{suffix}(X, Y),\\
&\mathit{append}([W|X], Y, [W|Z]) \leftarrow \mathit{append}(X, Y, Z),\\
&\mathit{concat}(X, [W|Y], [W|Z]) \leftarrow \mathit{concat}(X, Y, Z).
\end{aligned}$$

Note that the clauses of member and suffix have the same forms. The first argument of member is a constant symbol, while that of suffix is a list.

For the above clauses, the following corollary of Theorem 4.1 holds.

Corollary 6.1 (Hirata [Hir93a, Hir93b], Yamamoto [Yam92]) *Let p be a predicate symbol, α be a ground atom with p, and C be a clause $p(t_1, \cdots, t_n) \leftarrow p(s_1, \cdots, s_n)$.*

1. *If C is head-reducing, in particular 2-reducing, then all the derivation of $\{C\} \cup \{\leftarrow \alpha\}$ are finite.*
2. *If C is weakly head-reducing, in particular weakly 2-reducing, then all the derivation of $\{C\} \cup \{\leftarrow \alpha\}$ are finite, and all the nodes of the derivation are ground.*

6.2 Safe Generalization

It is an important problem to avoid overgeneralization when we deal with generalizations. In general, whether or not a generalization β of α is overgeneral is determined by an intended model. For an atom α, suppose that $\beta\theta = \alpha$ and M is an intended model. Then, β is an *overgeneralization* of α if there exists a ground atom γ such that $\forall\beta \vdash \gamma$ and $M \not\models \gamma$ for M. However, the decision problem of whether or not there exists such the ground atom γ is undecidable. On the other hand, in rule-generating abduction, only one surprising fact is given, and it is hard to give in advance an intended model. To overcome these difficulties, in this section, we introduce the following syntactical generalization of one atom.

Let θ be a ground substitution, that is, $\theta = \cup_{i=1}^{n}\{X_i := t_i\}$ and every term t_i is ground. Let α be a ground atom and β be an atom such that $\beta\theta = \alpha$. Note that, throughout this chapter, if $\beta\theta = \alpha$, then a variable $X_i \in dom(\theta)$ appears in β and $t_i \neq [\,]$. A substitution θ is *well-defined* if, for any t_i, there exists no term t_j which is a subterm of t_i.

Example 6.2 *Let α be a ground atom $p([a, b], [b])$. Let β_i be the following atoms $(1 \leq i \leq 5)$:*

$$\beta_1 = p([a, X], [b]),\ \beta_2 = p([a|X], [b]),\ \beta_3 = p([X|Y], Y),$$
$$\beta_4 = p([a|X], [Y]),\ \beta_5 = p([a|X], Y).$$

For any β_i, there exist the following substitutions θ_i such that $\beta_i\theta_i = \alpha$ $(1 \leq i \leq 5)$:

$$\theta_1 = \{X := b\},\ \theta_2 = \{X := [b]\},\ \theta_3 = \{X := a, Y := [b]\},$$
$$\theta_4 = \{X := [b], Y := b\},\ \theta_5 = \{X := [b], Y := [b]\}.$$

Then, θ_1, θ_2, and θ_3 are well-defined, while θ_4 and θ_5 are not.

Let α be a ground atom and β be an atom such that $\beta\theta = \alpha$. If a substitution $\theta = \cup_{i=1}^{n}\{X_i := t_i\}$ is well-defined, then we can define a *reversal* $\theta^{-1} = \cup_{i=1}^{n}\{t_i := X_i\}$. Note that, if θ is well-defined, then, for any t_i and X_i, there exists no term t_j such that t_j is a subterm of t_i and no variable X_i such that $X_i = X_j (j \neq i)$. However, even if θ is well-defined, β is not always $\alpha\theta^{-1}$.

Example 6.3 *For β_1 and β_2 in Example 6.2,*

$$\alpha\theta_1^{-1} = p([a, b], [b])\{b := X\} = p([a, X], [X]) \neq \beta_1,$$
$$\alpha\theta_2^{-1} = p([a, b], [b])\{[b] := X\} = p([a|X], X) \neq \beta_2.$$

On the other hand,

$$\alpha\theta_3^{-1} = p([a, b], [b])\{a := X, [b] := Y\} = p([X|Y], Y) = \beta_3.$$

For the reversal θ^{-1}, the following lemma holds.

Lemma 6.1 *Let α be a ground atom and β be an atom such that $\beta\theta = \alpha$. Suppose that a substitution $\theta = \cup_{i=1}^{n}\{X_i := t_i\}$ is well-defined. Then, $\beta = \alpha\theta^{-1}$ if and only if no term t_i appears in β.*

Proof. Suppose that $\beta = \alpha\theta^{-1}$. By the definition of θ^{-1}, $\alpha\theta^{-1}$ is an atom which replaces all the terms t_i in α with the variable X_i. Then, no term t_i appears in $\alpha\theta^{-1} = \beta$.

For simplicity, suppose that $\theta = \{X := t\}$. If a term t appears once in α, that is, α has the form of $p(\cdots t \cdots)$, then $\alpha\theta^{-1} = p(\cdots t \cdots)\{t := X\} = p(\cdots X \cdots)$. Since α is ground and $\beta\theta = \alpha$, $\alpha\theta^{-1} = \beta$.

If a term t appears at least twice in α, that is, α has the form of $p(\cdots t \cdots t \cdots)$, then β has the form of $p(\cdots X \cdots Y \cdots)$. If $X \neq Y$, then, by $\beta\theta = \alpha$, θ has the form of $\{X := t, Y := t\}$. This θ is not well-defined. Hence, it is contradiction. Then, $X = Y$, $\theta = \{X := t\}$, and β has the form of $p(\cdots X \cdots X \cdots)$. Hence, $\beta = \alpha\theta^{-1}$. ∎

A ground term t ($\neq [\ ]$) is a *common term* in α if t appears at least twice in α. In particular, if a common term is a ground list, it is called a *common list*. Then, we formulate a *safe generalization* which is based on the syntax of one ground atom as follows:

Definition 6.2 *Let α be a ground atom, θ be a substitution $\cup_{i=1}^{n}\{X_i := t_i\}$, and β be an atom such that $\beta\theta = \alpha$. An atom β is a safe generalization of α if (β, θ) satisfies the following safeness conditions:*

1. *θ is well-defined,*
2. *$\beta = \alpha\theta^{-1}$, and*
3. *if there exist common terms in α, then there exists a ground term $t_j \in \cup_{i=1}^{n}\{t_i\}$ such that t_j is a common term in α.*

Let α be a ground atom and β be an atom such that $\beta\theta = \alpha$. Let t be some common term in α. If θ is well-defined and θ^{-1} has the form of $\{t := X\}$, then β is safe on α.

Example 6.4 *Let α be a ground atom $p([a, b], [b])$ and β_i be an atom such that $\beta_i\theta_i = \alpha$ $(6 \leq i \leq 8)$. Then, the common terms in α are $[b]$ and b.*

1. *Let β_6 be an atom $p(X, Y)$ and θ_6 be a substitution $\{X := [a, b], Y := [b]\}$. By the safeness condition 1, β_6 is not safe on α.*

2. *Let β_7 be an atom $p([X, b], [Y])$ and θ_7 be a substitution $\{X := a, Y := b\}$. By the safeness condition 2, β_7 is not safe on α.*

3. *Let β_8 be an atom $p(X, [b])$ and θ_8 be a substitution $\{X := [a, b]\}$. By the safeness condition 3, β_8 is not safe on α.*

For the above α, atoms $p([a|X], X), p([a, X], [X]), p([Y, X], [X])$, and $p([Y|X], X)$ are safe on α.

In general, an atom is regarded as a relation between its arguments. Thus, the syntactical generalization of one atom should be obtained by replacing common terms with common variables. The safe generalization is an example of such generalizations. On the other hand, in weakly 2-reducing programs, it suffices to consider only two types of terms, constant symbols and lists. Then, we also define the following two types of substitutions.

Let θ be a substitution $\cup_{i=1}^{n}\{X_i := t_i\}$. Then, θ is a *constant substitution* (*resp.*, a *list substitution*) if every t_i is a constant symbol (*resp.*, a ground list) without an empty list [ ].

In particular, a constant substitution is related to *partially isomorphic generalizations* which have been introduced by Hirowatari and Arikawa [HiA94b]. Note that, though a replaceable term includes an empty list [ ], the definition of a replaceable term is independent of the proof of Theorem 2.3. Let RT be the set of all replaceable terms of α and $T \subseteq RT$. Then, we can re-formulate a partially isomorphic generalization by using T instead of RT, and show that Theorem 2.3 also holds for the set T of replaceable terms. Let α be a ground atom, θ_c be a constant substitution, and β be an atom such that $\beta\theta_c = \alpha$. Thus, we assume that *β is a partially isomorphic generalization of α, whose replaceable terms are all constant symbols in α except an empty list* [ ].

In Section 6.4, we apply rule-generating abduction to a list substitution θ_l and a constant substitution θ_c in the following way: Let α be a ground atom, that is, a surprising fact. First, by using a list substitution, we obtain an atom β such that

$\beta\theta_l = \alpha$ and β is safe on α. Secondly, by using a constant substitution, we obtain an atom γ such that $\gamma\theta_c = \beta$ and γ is safe on β. By the above assumption, γ is also a partially isomorphic generalization of β.

Unfortunately, $\theta_c\theta_l$ is not always well-defined, and γ is not always safe on α. For example, let α be a ground atom $p([a, b, c], [b, c], [a, b, c])$. Then, there exist the following atoms β_i such that $\beta_i\theta_i = \alpha$ and θ_i is a well-defined list substitution $(9 \leq i \leq 11)$:

$$\begin{array}{ll}
\beta_9 = p([a, b|X], [b|X], [a, b|X]) & \theta_9 = \{X := [c]\}, \\
\beta_{10} = p([a|Y], Y, [a|Y]) & \theta_{10} = \{Y := [b, c]\}, \\
\beta_{11} = p(Z, [b, c], Z) & \theta_{11} = \{Z := [a, b, c]\}.
\end{array}$$

Then, there exist no generalization γ of β and substitution $\sigma(\neq \varepsilon)$ such that $\gamma\sigma = \beta_{11}$ and $\theta_{11}\sigma$ is well-defined. Note that there does not exist the greatest list generalization of α.

Let α be a ground atom. Let β and γ be atoms such that $\beta\theta_l = \alpha$ and $\gamma\theta_c = \beta$. Suppose that both (β, θ_l) and (γ, θ_c) satisfy the safeness conditions. Then, the following two theorems hold.

Theorem 6.1 *If $\theta_c\theta_l$ is well-defined, then γ is a safe generalization of α.*

Proof. Suppose that $\theta_c\theta_l$ is well-defined. Then, $(\gamma, \theta_c\theta_l)$ satisfies the safeness condition 1.

Since both (β, θ_l) and (γ, θ_c) satisfy the safeness conditions, $(\gamma, \theta_c\theta_l)$ satisfies the safeness condition 3.

By the supposition, $\beta = \alpha\theta_l^{-1}$ and $\gamma = \beta\theta_c^{-1}$. The list substitution θ_l replaces the common lists in α by variables. The constant substitution θ_c replaces the same constant symbols in β by the same variables and other constant symbols by other variables. Hence, the composition $\theta_c\theta_l$ replaces the common lists in α by variables, the same constant symbols in α except common lists by the same variables, and other constant symbols by other variables. By Lemma 6.1 and since $\theta_c\theta_l$ is well-defined, then $(\gamma, \theta_c\theta_l)$ satisfies the safeness condition 2. ∎

Theorem 6.2 *Suppose that any constant symbol appearing in common lists in α does not appear elsewhere in α except in the lists. If $\beta = \alpha\theta_l^{-1}$ and $\gamma = \beta\theta_c^{-1}$, then $\theta_c\theta_l$ is well-defined. Hence, γ is a safe generalization of α.*

Proof. Suppose that $\theta_l = \cup_{i=1}^{n}\{X_i := l_i\}$, where l_i is a common list in α. For any j-th argument's term t_j of α, if t_j includes l_i, then $t_j = [a_1^j, a_2^j, \cdots, a_{n_j}^j | l_i]$, and no constant symbols $a_1^j, a_2^j, \cdots, a_{n_j}^j$ appear in l_i. Then, θ_c does not include the binding $X := c$ such that c appears in l_i. Hence, $\theta_c\theta_l$ is well-defined. By Theorem 6.1, $(\gamma, \theta_c\theta_l)$ satisfies the safeness conditions. Then, for γ such that $\gamma\theta_c\theta_l = \alpha$, γ is a safe generalization of α. ∎

6.3 Number of Hypotheses

In Chapter 4, we have discussed the head-reducing programs for which all the derivations are finite. For a given ground atom $p(t_1, \cdots, t_n)$, the head-reducing rule

$$p(t_1', \cdots, t_n') \leftarrow p(s_1', \cdots, s_n')$$

is generated and the hypothesis $p(s_1, \cdots, s_n)$ is proposed by rule-generating abduction, where

$$p(t_1', \cdots, t_n')\theta = p(t_1, \cdots, t_n),$$
$$p(s_1', \cdots, s_n')\theta = p(s_1, \cdots, s_n).$$

An inference schema is depicted by the following syllogism:

$$\frac{p(t_1, \cdots, t_n)}{p(t_1', \cdots, t_n') \leftarrow p(s_1', \cdots, s_n') \qquad p(s_1, \cdots, s_n)}.$$

Unfortunately, even if the generated rule is 2-reducing or weakly 2-reducing, the number of hypotheses increases in exponential order with respect to the length of an atom as follows:

Theorem 6.3 *Let $p(t_1, \cdots, t_n)$ be a surprising fact.*

1. *The number of 2-reducing rule which satisfies the above syllogism is at most 6^n.*

2. *The number of weakly 2-reducing rule which satisfies the above syllogism is at most* 3^n.

Proof. Suppose that the generated rule is the form $p(u_1, \cdots, u_n) \leftarrow p(v_1, \cdots, v_n)$.

First, we show the case 1. Suppose that the generated rule is 2-reducing. The number of combinations such that u_i has the form of $[W_i|X_i]$ for j arguments in n arguments is at most ${}_nC_j$. In such j arguments, the number of combinations such that at least one v_i has the form of X_i is at most

$$\sum_{i=2}^{j} {}_jC_i,$$

where ${}_1C_2$ is assumed to be 1. In such j arguments, the number of combinations such that at least two W_l and W_k $(l \neq k)$ are the same is also at most

$$\sum_{i=2}^{j} {}_jC_i.$$

In the remained $(n-j)$ arguments, the number of combinations for the variables of body is at most 2^{n-j}.

Then, the total number of hypotheses is at most

$$\sum_{j=1}^{n} {}_nC_j \times \left(\sum_{i=2}^{j} {}_jC_i \right)^2 \times 2^{n-j}.$$

Hence, we obtain the following formulas:

$$\begin{aligned}
\sum_{j=1}^{n} {}_nC_j \times \left(\sum_{i=2}^{j} {}_jC_i \right)^2 \times 2^{n-j} &= \sum_{j=1}^{n} {}_nC_j \times \left(2^j - j - 1 \right)^2 \times 2^{n-j} \\
&\leq \sum_{j=1}^{n} {}_nC_j \times 2^{2j} \times 2^{n-j} \\
&= \sum_{j=1}^{n} {}_nC_j \times 2^{n+j} \\
&= \left(\sum_{j=1}^{n} {}_nC_j \times 2^j \right) \times 2^n \\
&= \{(1+2)^n - 1\} \times 2^n \\
&\leq 6^n.
\end{aligned}$$

Here, we have used the following formula:

$$\sum_{i=0}^{n} {}_nC_i \times x^i = (1+x)^n.$$

Now, we also show the case 2. Suppose that the generated rule is weakly 2-reducing. The number of combinations such that u_i has the form of $[W_i|X_i]$ for j arguments in n arguments is at most ${}_nC_j$. In such j arguments, the number of combinations such that at least one v_i has the form of X_i is at most

$$\sum_{i=2}^{j} {}_jC_i,$$

where ${}_1C_2$ is assumed to be 1.

Then, the total number of hypotheses is at most

$$\sum_{j=1}^{n} {}_nC_j \times \left(\sum_{i=2}^{j} {}_jC_i \right).$$

Hence, we obtain the following formulas:

$$\sum_{j=1}^{n} {}_nC_j \times \left(\sum_{i=2}^{j} {}_jC_i \right) \; = \; \sum_{j=1}^{n} {}_nC_j \times (2^j - j - 1) \; \leq \; \sum_{j=1}^{n} {}_nC_j \times 2^j \; \leq \; 3^n.$$

∎

On the other hand, by using safe generalizations in Section 6.2, we design an algorithm to generate weakly 2-reducing rules as follows: Suppose that a ground atom α is given. First, by generalizing α with a list substitution θ_l, we obtain an atom β such that $\beta\theta_l = \alpha$ and β is safe on α. We call such a β a *list generalization* of α. Secondly, by generalizing β with a constant substitution θ_c, we obtain an atom γ such that $\gamma\theta_c = \beta$ and γ is safe on β. We call such a γ a *constant generalization* of β. Note that γ is also assumed to be a *partially isomorphic generalization* of β. For this algorithm, the number of rules is at most the number of generalizations. Then, we investigate the number of generalizations, in particular, the number of maximal generalizations.

Let α be a ground atom $p(t_1, \cdots, t_n)$. For all common lists in α, we can classify them by the *sublist relation*. For example, let α be the following ground atom:

$$p([a,b,c,d],[c,d],[b,c],[c],[b,c,d]) \; (= p(t_1,t_2,t_3,t_4,t_5)),$$

and t_i be the i-th argument's term of α. Then, t_2, t_4, and t_5 are common lists in α. By the sublist relation, we classify them into $\{t_2, t_5\}$ and $\{t_4\}$.

The number of maximal generalizations is characterized as the following theorem.

Theorem 6.4 *Let l be the number of classes by the sublist relation. Then, the number of the maximal generalizations is at most*

$$\left(\frac{\left(\sqrt{2}\right)^n}{l}\right)^l.$$

Even in case $l = 1$, the number of the maximal list generalizations of α is at most $\left(\sqrt{2}\right)^n$.

Proof. Let α be a ground atom $p(t_1, \cdots, t_n)$ and K_n be the number of the maximal list generalizations of α. If $l = 1$, then we can find the upper bound of K_n in the following way.

For simplicity, suppose that the common lists in α are $t_1, t_2, \cdots, t_{n-1}$, and $|t_1| > |t_2| > \cdots > |t_{n-1}|$. We denote the generalization $\alpha\{t_{j_1} := X_{j_1}, \cdots, t_{j_f} := X_{j_f}\}$ by $\beta_{(j_1,\cdots,j_f)}$. Note that t_{j_i+1} is not a common list in $\beta_{(j_i)}$. Furthermore, for $\beta_{(j_i,j_i+a,j_i+a+b)}$ $(a, b = 2 \text{ or } 3)$, there exist substitutions θ_{j_i+a+b}, θ_{j_i+a}, and θ_{j_i} such that

$$\begin{aligned}
\beta_{(j_i,j_i+a)} &= \beta_{(j_i,j_i+a,j_i+a+b)}\theta_{j_i+a+b}, \\
\beta_{(j_i,j_i+a+b)} &= \beta_{(j_i,j_i+a,j_i+a+b)}\theta_{j_i+a}, \\
\beta_{(j_i+a,j_i+a+b)} &= \beta_{(j_i,j_i+a,j_i+a+b)}\theta_{j_i}.
\end{aligned}$$

Hence, $\beta_{(j_i,j_i+a)}$, $\beta_{(j_i,j_i+a+b)}$, and $\beta_{(j_i+a,j_i+a+b)}$ are not maximal list generalizations.

By using the indices of β, K_n is equal to the number of the sequences $(j_1, \cdots, j_f)$ which satisfy the following conditions:

1. $j_1 = 1$ or 2,
2. $j_f = n - 1$ or n, and
3. the adjacent number of j_i is either $j_i + 2$ or $j_i + 3$.

For example, if $n = 8$, then the following seven sequences

$$(1, 3, 5, 7), (1, 3, 6), (1, 4, 6), (1, 4, 7), (2, 4, 6), (2, 4, 7), (2, 5, 8)$$

satisfy the above conditions. For the sequence $(j_1, \cdots, j_f)$ which satisfies the above conditions, the number of sequences such that $j_1 = 1$ is greater than $j_1 = 2$. Let A_n be the set of the sequences $(j_1, \cdots, j_f)$ which satisfy the above conditions and $j_1 = 1$. Then, $K_n \leq 2|A_n|$. Furthermore, for $n \geq 6$, we can construct the set A_n in the following way:

1. if $(j_1, \cdots, j_f) \in A_{n-2}$, then $(j_1, \cdots, j_f, n) \in A_n$, and
2. if $(j'_1, \cdots, j'_f) \in A_{n-3}$, then $(j'_1, \cdots, j'_f, n-1) \in A_n$.

Hence, $|A_n|$ satisfies the following equations:

$$|A_3| = 1,\ |A_4| = 1,\ |A_5| = 2,\ |A_n| = |A_{n-2}| + |A_{n-3}|\ (n \geq 6).$$

By mathematical induction on n, we obtain the following formula:

$$\left(\sqrt[3]{2}\right)^{n-2} \leq |A_n| \leq \left(\sqrt{2}\right)^{n-2} \quad (n \geq 6).$$

Hence, the number K_n of the maximal list generalizations is characterized by the following formula:

$$K_n \leq 2\left(\sqrt{2}\right)^{n-2} = \left(\sqrt{2}\right)^{n}.$$

Note that this formula also holds for any $n \geq 1$.

Let l be the number of classes by the sublist relation and C_j be such a class for $1 \leq j \leq l$. For any C_j, the number of the sequences which satisfy the above conditions is at most $\left(\sqrt{2}\right)^{|C_j|}$. Then, the number K_n of the maximal generalizations is at most

$$\left(\sqrt{2}\right)^{|C_1|} \times \cdots \times \left(\sqrt{2}\right)^{|C_l|}.$$

Hence, K_n is also characterized as the following formula:

$$K_n \leq \left(\frac{\left(\sqrt{2}\right)^{n}}{l}\right)^{l}.$$

∎

By Theorem 6.4, the number of weakly 2-reducing rules, even if we adopt the maximal list generalizations, increases exponentially with respect to n. Hence, in the next section, we restrict the forms of generalizations, and design the algorithm for rule-generating abduction whose number of hypotheses is at most n.

6.4 Algorithm PROPOSE

In this section, we design an efficient algorithm for rule-generating abduction, which is called *PROPOSE*. In the algorithm *PROPOSE* illustrated in Figure 6.1, we restrict the reversal for list generalizations to the form of $\{t := X\}$, where t is both a common list in α and some argument's term of α. Obviously, the list generalization $\alpha\{X := t\}$ is safe on α. Then, the following lemma holds.

Lemma 6.2 *The number of weakly 2-reducing rules generated by the algorithm PROPOSE is at most n.*

Proof. The number of terms that are both a common term and some argument's term is at most n. Then, the number of elements of L is at most n. Hence, the number of hypotheses is also at most n. ∎

An important basis on the algorithm *PROPOSE* is that, *if the i-th argument's term is some common list in α, then the i-th argument's term of the head of the generated rule is a variable; otherwise, it is a list.* Furthermore, by the algorithm *PROPOSE*, the clauses in Example 6.1 are constructed from one ground atom.

In Figure 6.1, $rs\_abd(fact, head \leftarrow body, hyp)$, which is rule-selecting abduction, is a procedure to propose a hypothesis hyp such that $(head)\sigma = fact$ and $(body)\sigma = hyp$ for some substitution σ.

For the algorithm *PROPOSE*, the following two theorems hold.

Theorem 6.5 *Let α be a ground atom $p(t_1, \cdots, t_n)$ and $k = |t_1| + \cdots + |t_n|$. Then, the algorithm PROPOSE computes the rules and hypotheses in $O(k^3)$ time.*

Proof. For any t_i, it can be determined whether or not t_i is a sublist of t_j in $O(|t_i|)$. Then, for any i, it can be determined whether or not t_i is a sublist of any $t_j (j \neq i)$ in $O((n-1)|t_i|)$. Hence, the set L in the algorithm *PROPOSE* can be constructed in $O((n-1)k)$.

Algorithm $PROPOSE(\alpha, head \leftarrow body, hyp)$

  input $\alpha = p(t_1, \cdots, t_n)$: a fact, i.e., a ground atom

  output $head \leftarrow body$: a rule

        δ : a hypothesis

  $L := \{\beta \mid \beta = \alpha\{t_i := V_i\}, t_i \textit{ is a common list in } \alpha\} \cup \{\alpha\}$;

                                        /* β : safe on α */

  while $L \neq \phi$ **do**

    select $\beta \in L$;

    $\gamma :=$ *the greatest constant generalization* $p(s_1, \cdots, s_n)$ *of* β;

    for $i = 1$ **to** n

      if $s_i = [\,]$ **then** /* base step */

        output $\gamma \leftarrow true$ /* a rule */

        output $true$ /* a hypothesis */

        halt;

      else if s_i is a variable **then** /* induction step */

        $head\_arg_i := X_i$; /* X_i is a new variable */

        $body\_arg_i := X_i$;

      else /* $s_i = [W_1^i, \cdots]$ */

        $head\_arg_i := [W_1^i | X_i]$; /* X_i is a new variable */

      end

    end

    $head := p(head\_arg_1, \cdots, head\_arg_n)$; /* $head\_arg_i = [W_1^i | X_i]$ or X_i */

    $body := p(X_1, \cdots, X_n)$;

    output $head \leftarrow body$ /* a rule */

    $rs\_abd(\alpha, head \leftarrow body, \delta)$; /* rule-selecting abduction */

    output δ /* a hypothesis */

    $L := L - \{\beta\}$;

  end

Figure 6.1: Algorithm *PROPOSE*

For the selected β in L, the greatest constant generalization of β is also a partially isomorphic generalization of β. By Theorem 2.4, a partially isomorphic generalization γ of β can be computed in $O(k^2)$. Since the procedures in the for-loop can be computed in $O(n)$, the for-loop terminates in $O(n^2)$. Then, the procedures in while-loop can be computed in $O(k^2+n^2)$. Since the number of elements in L is at most n by Lemma 6.2, the while-loop terminates in $O(k^2n+n^3)$. Hence, the algorithm *PROPOSE* terminates in $O((n-1)k + k^2n + n^3)$.

Since $n \leq k$, the algorithm *PROPOSE* computes rules and hypotheses in $O(k^3)$ time. ∎

Theorem 6.6 *Let α be a ground atom $p(t_1, \cdots, t_n)$ and δ be the proposed hypothesis $p(s_1, \cdots, s_n)$ by PROPOSE. If there exists a selected common list l in α by the algorithm PROPOSE and l appears in t_i, then l also appears in s_i.*

Proof. Let l be the selected common list in α by *PROPOSE*. If the i-th argument's term t_i of α is l itself, then the i-th argument's terms of both the head and the body of the generated rules are variables X_i. Then, the i-th argument's term s_i of δ is also l.

Suppose that l appears in another argument's term of α, and t_i has the form of $[a_1^i, a_2^i, \cdots, a_{n_i}^i | l]$. By the algorithm *PROPOSE*, the i-th argument's term of the head of the generated rule is a list $[Y_1^i | X_i]$, where Y_1^i is a variable corresponding to a_1^i, while one of the body is a variable X_i. Then, for the hypothesis δ, the i-th argument's term s_i of δ has the form of $[a_2^i, \cdots, a_{n_i}^i | l]$.

Hence, the selected common list l also appears in the i-th argument's term s_i of δ. ∎

Theorem 6.6 claims that, if a given ground atom satisfies the relation on common lists, then the proposed hypothesis by the algorithm *PROPOSE* also satisfies it.

6.5 Examples

In this section, we discuss the several examples for the algorithm *PROPOSE*.

Example 6.5 *Let α be a ground atom $p([a,b],[c,d],[a,b,c,d])$. The list $[c,d]$ is both a common list in α and the second argument's term of α. By the construction of L, $\beta = p([a,b],V_2,[a,b|V_2])$ is a safe list generalization of α, and $\gamma = p([X,Y],V_2,[X,Y|V_2])$ is the greatest constant generalization of β. The first argument's term of γ is a list which begins with X, the second argument's term is a variable V_2, and the third argument's term is also a list which begins with X. By the for-loop in PROPOSE, we obtain a head $p([X|X_1],X_2,[X|X_3])$ and a body $p(X_1,X_2,X_3)$. Hence, PROPOSE generates a rule*

$$p([X|X_1],X_2,[X|X_3]) \leftarrow p(X_1,X_2,X_3),$$

and proposes a hypothesis $p([b],[c,d],[b,c,d])$. Note that the predicate p means the append in Example 6.1.

Since L includes α, then $\beta = \alpha$, and $\gamma = p([X,Y],[Z,W],[X,Y,Z,W])$. The first argument's term of γ is a list which begins with X, the second argument's term is a list which begins with Z, and the third argument's term is also a list which begins with X. By the for-loop in PROPOSE, we obtain a head $p([X|X_1],[Z|X_2],[X|X_3])$ and a body $p(X_1,X_2,X_3)$. Hence, PROPOSE generates a rule

$$p([X|X_1],[Z|X_2],[X|X_3]) \leftarrow p(X_1,X_2,X_3),$$

and proposes a hypothesis $p([b],[d],[b,c,d])$.

Furthermore, each of the rules and hypotheses by PROPOSE respectively satisfies the following syllogisms:

$$\frac{p([a,b],[c,d],[a,b,c,d])}{p([X|X_1],X_2,[X|X_3]) \leftarrow p(X_1,X_2,X_3) \qquad p([b],[c,d],[b,c,d])},$$

$$\frac{p([a,b],[c,d],[a,b,c,d])}{p([X|X_1],[Z|X_2],[X|X_3]) \leftarrow p(X_1,X_2,X_3) \qquad p([b],[d],[b,c,d])}.$$

Example 6.6 *Let α be a ground atom $p(a,[a,b])$. Since there exist no common lists in α, then $\beta = p(a,[a,b])$, and $\gamma = p(X,[X,Y])$. The first argument's term of γ is a variable X, and the second argument's term is a list which begins with X. By the for-loop in PROPOSE, we obtain a head $p(X_1,[X|X_2])$ and a body $p(X_1,X_2)$. Hence, PROPOSE generates a rule*

$$p(X_1, [X|X_2]) \leftarrow p(X_1, X_2),$$

and proposes a hypothesis $p(a, [b])$. Note that the predicate p means the member in Example 6.1.

Let α be a ground atom $p([a], [a, b])$. Since there exist no common lists in α, $\beta = p([a], [a, b])$ and $\gamma = p([X], [X, Y])$. The first argument's term of γ is a list which begins with X, and the second argument's term is also a list which begins with X. By the for-loop in PROPOSE, we obtain a head $p([X|X_1], [X|X_2])$ and a body $p(X_1, X_2)$. Hence, PROPOSE generates a rule

$$p([X|X_1], [X|X_2]) \leftarrow p(X_1, X_2),$$

and proposes a hypothesis $p([\ ], [b])$. Note that the predicate p means the prefix in Example 6.1.

Let α be a ground atom $p([b], [a, b])$. The list $[b]$ is both a common list in α and the first argument's term of α. Then, $\beta = p(V_1, [a|V_1])$ and $\gamma = p(V_1, [X|V_1])$. The first argument's term of γ is a variable V_1, and the second argument's term is a list which begins with X. By the for-loop in PROPOSE, we obtain a head $p(X_1, [X|X_2])$ and a body $p(X_1, X_2)$. Hence, PROPOSE generates a rule

$$p(X_1, [X|X_2]) \leftarrow p(X_1, X_2),$$

and proposes a hypothesis $p([b], [b])$. Note that the predicate p means the suffix in Example 6.1. On the other hand, since L includes $\alpha = p([b], [a, b])$, then $\beta = \alpha$, and $\gamma = p([X], [Y, X])$. The first argument's term of γ is a list which begins with X, and the second argument's term is a list which begins with Y. By the for-loop in PROPOSE, we obtain a head $p([X|X_1], [Y|X_2])$ and a body $p(X_1, X_2)$. Hence, PROPOSE generates a rule

$$p([X|X_1], [Y|X_2]) \leftarrow p(X_1, X_2),$$

and proposes a hypothesis $p([\ ], [b])$. Note that the predicate p means the defining lists, that is, all arguments' terms are lists.

If $\alpha = p([a,b],[c,d],[a,b,c,d])$, then *PROPOSE* generates a rule

$$p([X|X_1], X_2, [X|X_3]) \leftarrow p(X_1, X_2, X_3),$$

and proposes a hypothesis $p([b],[c,d],[b,c,d])$. If $\alpha = p([a,b],[a,b,c,d],[c,d])$, then *PROPOSE* also generates a rule

$$p([X|X_1], [X|X_2], X_3) \leftarrow p(X_1, X_2, X_3),$$

and proposes a hypothesis $p([b],[b,c,d],[c,d])$. Hence, the algorithm *PROPOSE* is independent of the order of arguments.

Furthermore, by the construction of *member* and *suffix* in Example 6.6, the algorithm *PROPOSE* is also independent of the types of argument. In other words, *PROPOSE* needs no types of arguments.

6.6 Prolog Implementation

We can realize the algorithm *PROPOSE* in a Prolog program as in Figure 6.2. The predicate `propose` returns a generated rule as the third argument. It also returns a hypothesis proposed by the generated rule as the second argument. (Full Prolog version will be described in Appendix of this thesis.)

The predicate `propose` in Figure 6.2 returns a proposed hypothesis as its second argument and a generated rule as its third argument for a surprising fact as its first argument. The predicate `while_loop` means the while-loop in the algorithm *PROPOSE*. The remainder which follows the predicate `while_loop` in the predicate `propose` means the construction of base step in the algorithm *PROPOSE*. The predicate `list_gen` and `const_gen` mean list and constant generalizations, respectively.

For the following five surprising facts

$$p([a,b],[c,d],[a,b,c,d]),\ p(a,[a,b]),\ p([a],[a,b]),$$
$$p([b],[a,b]),\ p([a,b,c],[b,c],[a,b,c]),$$

```
propose(Fact,Atoms,(NewHead:-NewBody)) :-
        !,
        while_loop(Fact,(NewHead:-TmpNewBody)),
        (NewHead == TmpNewBody ->
            NewBody = true,
            const_gen(Fact,TmpHead1),
            list_gen_main(TmpHead1,TmpHead2),
            replace_term(TmpHead2,list,_,NewHead),!;
            NewBody = TmpNewBody,
            assert((NewHead:-NewBody)),
            rs_abd(Fact,Atoms),
            retract((NewHead:-NewBody))).

while_loop(X,(NewHead:-NewBody)) :-
        list_gen(X,List),
        member_of(W,List),
        common_list_var(W,NW),
        const_gen(NW,Head),
        create(Head,Body),
        create2(Head,Body,NewHead,NewBody).
```

Figure 6.2: Program propose

the results of propose are as follows:

```
: ?- propose(p([a,b],[c,d],[a,b,c,d]),X,Y).
X = p([b],[d],[b,c,d]),
Y = p([_2046|_2602],[_1378|_2600],[_2046|_2598]):-p(_2602,_2600,_2598) ;
X = p([b],[c,d],[b,c,d]),
Y = p([_1578|_2070],_2058,[_1578|_2066]):-p(_2070,_2058,_2066) ;
no

: ?- propose(p(a,[a,b]),X,Y).
X = p(a,[b]),
Y = p(_1334,[_972|_1340]):-p(_1334,_1340) ;
no

: ?- propose(p([a],[a,b]),X,Y).
X = p([],[b]),
Y = p([_1016|_1418],[_1016|_1416]):-p(_1418,_1416) ;
no

: ?- propose(p([b],[a,b]),X,Y).
X = p([],[b]),
Y = p([_1158|_1560],[_904|_1558]):-p(_1560,_1558) ;
X = p([b],[b]),
Y = p(_1220,[_872|_1226]):-p(_1220,_1226) ;
no
```

```
: ?- propose(p([a,b,c],[b,c],[a,b,c]),X,Y).
X = p([b,c],[c],[b,c]),
Y = p([_2066|_2622],[_1738|_2620],[_2066|_2618]):-p(_2622,_2620,_2618) ;
X = p([b,c],[b,c],[b,c]),
Y = p([_1638|_2102],_2090,[_1638|_2098]):-p(_2102,_2090,_2098) ;
X = p([a,b,c],[c],[a,b,c]),
Y = p(_2360,[_1670|_2368],_2356):-p(_2360,_2368,_2356) ;
no
```

Note that, in the last example, two atoms $\beta_1 = p([a|V_2], V_2, [a|V_2])$ and $\beta_2 = p(V_1, [b, c], V_1)$ are the safe list generalizations of $p([a, b, c], [b, c], [a, b, c])$ obtaining by the algorithm *PROPOSE*. Hence, for a ground atom $p([a, b, c], [b, c], [a, b, c])$, there are three hypotheses and rules. The first rule obtained by *PROPOSE* means that all arguments' terms are lists, and the first and the third arguments' terms begin with the same constant symbols. The second rule means that the second argument's term is the sublist of the first and the third arguments' terms. The third rule means that the first argument's term is equal to the third argument.

On the other hand, for a ground atom $p([a, b, c], [d, e], [a, b, c])$, the predicate `propose` returns the following two rules and hypotheses:

```
: ?- propose(p([a,b,c],[d,e],[a,b,c]),X,Y).
X = p([b,c],[e],[b,c]),
Y = p([_2510|_3066],[_1490|_3064],[_2510|_3062]):-p(_3066,_3064,_3062) ;
X = p([a,b,c],[e],[a,b,c]),
Y = p(_2056,[_1366|_2064],_2052):-p(_2056,_2064,_2052) ;
no
```

From this surprising fact, the rule whose second argument's term is the sublist of the first and the third arguments' terms is not generated by *PROPOSE*.

By the algorithm *PROPOSE*, we can obtain a rule. If an intended model M is given, and an oracle to determine whether or not $M' \subseteq M$ is also given, then we can design the algorithm as Figure 6.3.

Consider Example 6.5. Let $p([a, b], [c, d], [a, b, c, d])$ be a surprising fact. we obtain the following two rules R_i and hypotheses H_i by *PROPOSE*:

$$\left\{ \begin{array}{l} R_1 : p([X|X_1], X_2, [X|X_3]) \leftarrow p(X_1, X_2, X_3) \\ H_1 : p([b], [c, d], [b, c, d]) \end{array} \right\},$$

input $\alpha = p(t_1, \cdots, t_n)$: a fact, i.e., a ground atom
M: the least Herbrand model of P
output P' : a 2-reducing program

select $\alpha \in M$;
$PROPOSE(\alpha, rule, hyp)$;
$M' :=$ *the least Herbrand model of* $\{rule, hyp\}$;
if $M' \subseteq M$ **then** /* oracle */
  output $P' := \{rule, hyp\}$
halt

Figure 6.3: *PROPOSE* and oracle

$$\left\{ \begin{array}{l} R_2 : p([X|X_1], [Z|X_2], [X|X_3]) \leftarrow p(X_1, X_2, X_3) \\ H_2 : p([b], [d], [b, c, d]) \end{array} \right\}.$$

Suppose that an intended model is given as the least Herbrand model M_1 of the following program *append*:

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

Then, the least Herbrand model of $\{R_1, H_1\}$ is a subset of M_1, while one of $\{R_2, H_2\}$ is not a subset of M_1. Hence, the program $\{R_1, H_1\}$ is obtained by the algorithm in Figure 6.3. The program *append* means that the third argument's list is the result of concatenating the first and the second arguments' lists.

On the other hand, suppose that an intended model is given as the least Herbrand model M_2 of the following program *list*:

$$\left\{ \begin{array}{l} list([\,], [\,], [\,]) \\ list([W_1|X], [W_2|Y], [W_3|Z]) \leftarrow list(X, Y, Z) \end{array} \right\}.$$

Then, both the least Herbrand models of $\{R_1, H_1\}$ and of $\{R_2, H_2\}$ are subsets of M_2. Hence, the programs $\{R_1, H_1\}$ and $\{R_2, H_2\}$ are obtained by the algorithm in Figure 6.3. The program *list* means that all arguments' terms are lists.

Chapter 7

Conclusion

> *"The case has been an interesting one," remarked Holmes, when our visitors had left, "because it serves to show very clearly how simple the explanation may be of an affair which at first sight seems to be almost inexplicable."* — *'The Adventure of the Noble Bachelor'*
> *"The Adventures of Sherlock Holmes"*

This thesis has discussed abduction for logic programming.

In Chapter 3, we have classified abduction in computer science into five types: *rule-selecting abduction*, *rule-finding abduction*, *rule-generating abduction*, *theory-selecting abduction*, and *theory-generating abduction*. This classification is based on the interpretation of syllogism and the definition of hypothesis. Furthermore, we have examined various researches on abduction in computer science so far developed, and shown that most of them can be placed in our classification.

In Chapter 4, we have investigated rule-selecting abduction for logic programming. From the philosophical viewpoint, abduction is the first stage of scientific inquiry. Then, we should consider the process of abduction which terminates. In order to characterize the termination, we have introduced the concept of *head-reducing* programs. We have shown that all the derivations for a head-reducing program and a surprising fact are finite. Hence, all the processes of rule-selecting abduction for a head-reducing program are finite.

Furthermore, we have compared rule-selecting abduction with default logic. We have formulated a surprising fact and a hypothesis for default logic, and shown that, if there exists a hypothesis which explains a surprising fact, then there also exists an extension of a given default theory, which includes the surprising fact. This extension is corresponding to the least Herbrand model of the definite program obtaining from the default theory. This result is an extension of Poole's theory [Poo88].

Since the class of head-reducing programs is not so large, we have extended the concept of head-reducingness to that of *breadth-first head-reducing* programs, and the rule-selecting abduction to the *breadth-first rule-selecting abduction*. We have shown that there exists a finite derivation for a breadth-first head-reducing program and a surprising fact. Hence, the process of breadth-first rule-selecting abduction for a breadth-first head-reducing program is finite.

Finally, rule-selecting abduction for logic programming and for default logic, and breadth-first rule-selecting abduction have been implemented by Prolog programs.

In Chapter 5, we have investigated rule-finding abduction for logic programming. We have introduced two concepts of *loop-pair* and *loop-elimination*. The loop-pair syntactically determines whether or not there exists an infinite process of rule-finding abduction. On the other hand, the loop-elimination is a transformation of programs. By using loop-elimination, we can choose the programs for which the process of rule-finding abduction terminates. We have shown that if a loop-pair appears in a derivation, then the derivation becomes infinite. We have also shown that, for given two programs, if we transform one program by loop-elimination, then all the derivations for union of the transformed program and the rest are finite.

Furthermore, we have formulated *rule-finding abduction with analogy*, which is an extension of rule-finding abduction. We have introduced the concept of *deducible hypotheses*, which are hypotheses for rule-finding abduction and are guaranteed correct in the sense of analogical reasoning. In this formulation, in order to obtain an analogy while constructing a deducible hypothesis, we have adopted *partially isomorphic generalizations*. By using these concepts, we have designed an algorithm of rule-finding abduction with analogy, and implemented it by a Prolog program.

In Chapter 6, we have investigated rule-generating abduction for logic programming. We have introduced *weakly 2-reducing* programs for rule-generating abduction. We have also discussed a *safe generalization*, which is a generalization of one atom whose common terms are replaced by common variables. We have given some properties of safe generalizations.

In rule-generating abduction for weakly 2-reducing programs, we have shown that the number of hypotheses increases in exponential order with respect to the length of a surprising fact. On the other hand, by using safe generalizations, we have designed an algorithm *PROPOSE* to construct weakly 2-reducing rules. The number of hypotheses by *PROPOSE* is at most the length of a surprising fact. We have shown that the algorithm *PROPOSE* generates rules and proposes hypotheses in polynomial time with respect to the length of a surprising fact. Also we have shown that the selected common list in some argument of a surprising fact appears in the same argument of the hypothesis proposed by *PROPOSE*.

We have left several problems as future works.

In this thesis, a surprising fact has been defined by a ground atom α such that $P \not\vdash \alpha$ for a background theory P. However, the definition of a surprising fact may possibly be extended by introducing probability, cost [HSME88, Sti91], modality [Lev89, SL90], or causal weight [BATJ91]. It is a future work to investigate what a surprising fact is in these new settings.

In this thesis, we have dealt with *rule-based abduction*, but not *theory-based abduction*. Many systems which are related to theory-based abduction have already designed and realized. For example, there are Shapiro's model inference system [Sha81], inductive logic programming [Mug92, MB88, Lin89, LU89], Poole's Theorist [Poo88], and hypothesis-based reasoning [Kun87]. Concerning these systems, we have left the problems to find the abduction, to characterize the class of objects, and to investigate the theoretical properties such as termination and computational complexity.

We have discussed abduction for *definite programs*. On the other hand, rule-selecting abduction is related to nonmonotonic logic. When we discuss the relationship between nonmonotonic logic and logic programming, we can also deal with such extensions of definite programs as *normal program*, *general program*, and *disjunctive program* [Dun91, EK89, KM90, KKT92, Llo87]. Abductive logic programming is a kind of rule-selecting abduction for such programs, while the surprising fact in this setting is not surprising in our sense. Hence, we need to extend the class of programs for abduction and investigate the property of their abduction in our sense.

This thesis has also discussed rule-finding abduction and analogical reasoning in the same framework. This is a certain step toward acquiring the knowledge from abductive and analogical viewpoints, although we have just shown a few theoretical results. We need to formulate so called *analogy by abduction* other than *abduction with analogy*. We also need to solve the problem of incorporating rule-finding abduction with rule-generating abduction by using analogy.

Abduction is an inference to propose hypotheses which should be used before deduction and induction are applied. We have left the problem to combine abduction, in particular rule-generating abduction, and inductive logic programming. Ling [Lin89, LU89] has introduced the *constructive* inductive logic programming. There may exist a relationship between such works and ours.

Furthermore, as to the inductive logic programming, we have left the problem of predicate invention. In general, the number of rules and hypotheses proposed in predicate invention becomes exponentially large. Hence, we need to introduce some heuristics such as on the number of local variables, invented predicate symbols and rules, a distinction between necessary and useful intermediate terms, and so on.

All researches on abduction in computer science are based on computational logic, and the abduction beyond the scope of computational logic such as abduction for formal language or numerical data has not yet been studied. On the other hand, abduction begins with a surprising fact. In machine learning, a surprising fact is considered as a good example. Hence, abduction is regarded as *learning from good examples*. We need to study the abduction for various frameworks together with machine learning. This should be one of the most important future works.

References

[ASO91] Arimura, H., Shinohara, T. and Otsuki, S.: *A polynomial time algorithm for finite unions of tree pattern languages*, Proceedings of the 2nd Workshop on Algorithmic Learning Theory, 105–114, 1991.

[BATJ91] Bylander, T., Tanner, M. C., Allemang, D. and Josephson, J. R.: *The computational complexity of abduction*, Artificial Intelligence **49**, 25–60, 1991.

[Bro77] Brown, H. I.: *Perception, theory and commitment*, Precedent Publishing, 1977.

[Cha79] Chalmers, A. F.: *What is this thing called science?*, University of Qeensland Press, 1979.

[CL73] Chang, C. and Lee, R.: *Symbolic logic and mechanical theorem proving*, Academic press, 1973.

[CP87] Cox, P. T. and Pietrzykowski, T.: *General diagnosis by abductive inference*, Proceedings of the 1987 Symposium on Logic Programming, 183–189, 1987.

[Dun91] Dung, P. M.: *Negation as hypothesis: an abductive foundation for logic programming*, Proceedings of the 8th International Conference on Logic Programming, 3–17, 1991.

[Duv91] Duval, B.: *Abduction for explanation-based learning*, Proceedings of European Working Session on Learning (1991), Lecture Notes in Artificial Intelligence **482**, 348–360, 1991.

[EK89] Eshghi, K. and Kowalski, R. A.: *Abduction compared with negation by failure*, Proceedings of the 6th International Conference on Logic Programming, 234–254, 1989.

[Fla94] Flach, P.: *Simply logical*, John Wiley & Sons, 1994.

[GMP90] Genest, J., Matwin, S. and Plante, B.: *Explanation-based learning with incomplete theories: a three-step approach*, Proceedings of the 7th International Conference on Machine Learning, 286–294, 1990.

[Han58] Hanson, N. R.: *Patterns of discovery*, Cambridge University Press, 1958.

[Har85] Haraguchi, M.: *Towards a mathematical theory of analogy*, Bulletin of Informatics and Cybernetics **21**, 29–56, 1985.

[HaA86] Haraguchi, M. and Arikawa, S.: *Analogical reasoning using transformations of rules*, Bulletin of Informatics and Cybernetics **22**, 1–8, 1986.

[Hir92] Hirata, K.: *Finding minimal models by the semantic tableaux system*, RIFIS Technical Report **72**, 1992.

[Hir93a] Hirata, K.: *A classification of abduction: abduction for logic programming*, RIFIS Technical Report **77**, 1993.

[Hir93b] Hirata, K.: *A classification of abduction: abduction for logic programming*, Machine Intelligence **14** (to appear).

[Hir94a] Hirata, K.: *Stable model semantics and circumscription*, Bulletin of Informatics and Cybernetics **26**, 1–12, 1994.

[Hir94b] Hirata, K.: *Rule-generating abduction for recursive Prolog*, RIFIS Technical Report **85**, 1994.

[Hir94c] Hirata, K.: *Rule-generating abduction for recursive Prolog*, Proceedings of the 4th International Workshop on Analogical and Inductive Inference, Lecture Notes in Artificial Intelligence **872**, 121–136, 1994.

[Hir94d] Hirata, K.: *A note on rule-finding abduction*, RIFIS Technical Report **95**, 1994.

[HiA94a] Hirowatari, E. and Arikawa, S.: *Incorporating explanation-based generalization and analogical reasoning*, Bulletin of Informatics and Cybernetics **26**, 13-34, 1994.

[HiA94b] Hirowatari, E. and Arikawa, S.: *Partially isomorphic generalization and analogical reasoning*, Proceedings of European Conference on Machine Learning (1994), Lecture Notes in Artificial Intelligence **784**, 363–366, 1994.

[HSME88] Hobbs, J. R., Stickel, M., Martin, P. and Edwards, D: *Interpretation as abduction*, Proceedings of the 26th Annual Meeting of the Association for Computing Linguistics, 95–103, 1988.

[Ino92] Inoue, K.: *Principles of abduction*, Journal of Japanese Society for Artificial Intelligence **7**, 48–59, 1992 (in Japanese).

[KKT92] Kakas, A. C., Kowalski, R. A. and Toni, F.: *Abductive logic programming*, Journal of Logic and Computation **2**, 719–770, 1992.

[KM90] Kakas, A. C. and Mancarella, P.: *Generalized stable models: a semantics for abduction*. Proceedings of the 9th European Conference on Artificial Intelligence, 385–391, 1990.

[Kon92] Konolige, K.: *Abduction versus closure in causal theories*, Artificial Intelligence **53**, 255–272, 1992.

[Kuh70] Kuhn, T. S.: *The structure of scientific revolutions*, University of Chicago Press, 1970.

[Kun87] Kunifuji, S.: *Hypothesis-based reasoning*, Journal of Japanese Society for Artificial Intelligence **2**, 22–87, 1987 (in Japanese).

[Lak76] Lakatos, I.: *Proofs and refutations: the logic of mathematical discovery*, Cambridge University Press, 1976.

[Lev89] Levesque, H. J.: *A knowledge-level account of abduction*, Proceeding of the 11th International Joint Conference on Artificial Intelligence, 1061–1067, 1989.

[Lif85] Lifschitz, V.: *Computing circumscription*, Proceedings of the 9th International Joint Conference on Artificial Intelligence, 121–127, 1985.

[Lin89] Ling, X.: *Learning and invention of Horn clause theories - a constructive method*, Methodologies for Intelligent Systems **4**, 323–331, 1989.

[LU89] Ling, X. and Ungar, L: *Inventing theoretical terms in inductive learning of functions - search and constructive methods*, Methodologies for Intelligent Systems **4**, 332–341, 1989.

[Llo87] Lloyd, J. W.: *Foundations of logic programming (2nd edn.)*, Springer-Verlag, 1987.

[McC80] McCarthy, J.: *Circumscription - a form of non-monotonic reasoning*, Artificial Intelligence **13**, 81–132, 1980.

[McC86] McCarthy, J.: *Applications of circumscription to formalizing commonsense knowledge*, Artificial Intelligence **28**, 89–116, 1986.

[McD82] McDermott, D.: *Nonmonotonic logic II : nonmonotonic modal theories*, Journal of Association for Computing Machinery **29**, 33–57, 1982.

[MD80] McDermott, D. and Doyle, J.: *Non-monotonic logic I*, Artificial Intelligence **13**, 41–72, 1980.

[Men87] Mendelson, E.: *Introduction to mathematical logic (3rd edn.)*, Wadsworth & Brooks, 1987.

[Moo85] Moore, R. C.: *Semantical considerations on nonmonotonic logic*, Artificial Intelligence **25**, 75–94, 1985.

[MB88] Muggleton, S. and Buntine, W.: *Machine invention of first-order predicates by inverting resolution*, Proceedings of the 5th International Conference on Machine Learning, 339–352, 1988; in [Mug92].

[Mug92] Muggleton, S. (ed.): Inductive logic programming, Academic Press, 1992.

[MKKC86] Mitchell, T. M., Keller, R. M. and Kedar-Cabelli, S. T., *Explanation-based generalization: a unifying view*, Machine Learning **1**, 47–80, 1986.

[Pei65] Peirce, C. S.: *Collected papers of Charles Sanders Peirce (1839-1914)*, Hartshone, C. S. and Weiss, P.(eds.), The Belknap Press, 1965.

[PP90] Pirri, F. and Pizzuti, C.: *A stable model semantics for diagnostic hypothesis*, Proceedings of Pacific Rim International Conference on Artificial Intelligence '90, 766–771, 1990.

[Plo70] Plotkin, G. D.: *A note on inductive generalization*, Machine Intelligence **5**, 153–163, 1970.

[Plo71] Plotkin, G. D.: *A further note on inductive generalization*, Machine Intelligence **6**, 101–124, 1971.

[Pol54a] Polya, G.: *Induction and analogy in mathematics (Mathematical and plausible reasoning Vol.1)*, Princeton University Press, 1954.

[Pol54b] Polya, G.: *Patterns of plausible inference (Mathematical and plausible reasoning Vol.2)*, Princeton University Press, 1954.

[Pol57] Polya, G.: *How to solve it: a new aspect of mathematical method (2nd edn.)*, Princeton University Press, 1957.

[Poo88] Poole, D.: *A logical framework for default reasoning*, Artificial Intelligence **36**, 27–47, 1988.

[Popl73] Pople, Jr. H. E.: *On the mechanization of abductive logic*, Proceedings of the 3rd International Joint Conference on Artificial Intelligence, 147–152, 1973.

[Popp59] Popper, K. R.: *The logic of scientific discovery*, Hutchinson, 1959.

[Reic38] Reichenbach, H.: *Experience and prediction*, University of Chicago Press, 1938.

[Reit80] Reiter, R.: *A logic for default reasoning*, Artificial Intelligence **13**, 81–132, 1980.

[SL90] Selman, B. and Levesque, H. J.: *Abductive and default reasoning: a computational core*, Proceedings of the 8th National Conference on Artificial Intelligence, 343–348, 1990.

[Sha81] Shapiro, E. Y. *Inductive inference of theories from facts*, Research Report 192, Yale University, 1981.

[SS86] Sterling, L. and Shapiro, E.: *The art of Prolog*, The MIT Press, 1986.

[SS94] Sterling, L. and Shapiro, E.: *The art of Prolog (2nd edn.)*, The MIT Press, 1994.

[Sti91] Stickel, M. E.: *A Prolog-like inference system for computing minimum-cost abductive explanation in natural language interpretation*, Annals of Mathematics and Artificial Intelligence **4**, 89–106, 1991.

[Tha88] Thagard, P.: *Computational philosophy of science*, MIT Press, 1988.

[Uey79] Ueyama, S.: *The theory of abduction*, Jinbungakuhou **43**, 103–155, 1978 (in Japanese).

[vHB88] van Harmelen, F. and Bundy, A.: *Explanation-based generalization = partial evaluation*, Artificial Intelligence **36**, 401–412, 1988.

[Yam92] Yamamoto, A.: *Procedural semantics and negative information of elementary formal system*, Journal of Logic Programming **13**, 89–97, 1992.

[Yon82] Yonemori, Y.: *Peirce's semiotics*, Keisou Syobou, 1982 (in Japanese).

Chapter 8

Appendix: Prolog Implementation

"You really are an automaton – a calculating machine," I cried.
— *"The Sign of Four"*

In this appendix, we express the three Prolog programs, *loop-elimination*, *rule-finding abduction with analogy*, and *PROPOSE*. All of them have been realized by K-Prolog Compiler version 3.10.

8.1 Loop-Elimination

We describe our realization of the loop-elimination for rule-finding abduction as the following Prolog programs.

```
%%%%% loop-elimination of Fact
%%%%% le_main(Fact,NewFact) :-
%%%%%     NewFact is loop-elimination of Fact.
%%%%% le(PFact,Fact,NewFact) :-
%%%%%     PFact is a priori fact, Fact is a clause to eliminate loop,
%%%%%     as a result. Then, we can obtain NewFact as loop-elimination.

le_main(fact(World,clause(Head,Bodies)),fact(World,clause(Head,NewBody))) :-
        setof(Y,le(fact(World,clause(Head,Bodies)),Y),List),
        isort(List,List2),
        List2 = [MaxFact|List3],
        MaxFact = fact(World,clause(Head,NewBody)).

le(fact(World,clause(Head,Bodies)),fact(World,clause(Head,NewBody))) :-
        fact(PW,clause(PH,PB)),
        PW \= World,
        le(Bodies,PH,NewBody).

le(SingleBody,PH,NewBody) :-
```

```
        PH = SingleBody ->
        (fact(PW,clause(PH,PB)),
        functor(PH,F1,Arg),
        functor(SingleBody,F2,Arg),
           (F1 = F2 ->
              Arg1 is Arg+1,
              functor(TrueBody,true,Arg1),
              arg(1,TrueBody,F2),
              (between(2,N,Arg1),
                  M is N-1,
                  arg(M,SingleBody,Var),
                  arg(N,TrueBody,Var)),
              replace_term(SingleBody,SingleBody,TrueBody,NewBody) ;
              fail),!).

le((Body1,Body2),PH,(NewBody1,NewBody2)) :-
        !,
        (fact(PW,clause(PH,_)),le(Body1,PH,NewBody1)),
        (fact(PW,clause(PH2,_)),le(Body2,PH2,NewBody2)).

le(Body,PH,Body) :- fact(PW,clause(PH,PB)),!.

isort([],[]).
isort([E|X],Z) :- isort(X,Y),insert(E,Y,Z).

insert(E,[],[E]).
insert(E,[X|Y],[X|Z]) :- E @< X,insert(E,Y,Z).
insert(E,[X|Y],[E,X|Y]) :- E @>= X.

between(Low,Low,Up).
between(Low,I,Up) :- between(Low,Old,Up),I is Old+1,((I > Up,!,fail);true).
```

8.2 Rule-Finding Abduction with Analogy

The rule-finding abduction with analogy in Section 5.6 is realized as the following Prolog program. In this program, the least generalization `lg` refers to Flach's Prolog text [Fla94], and the partially isomorphic generalizations `pig_rule` and `pig` Hirowatari and Arikawa [HiA94b].

```
%%%%%%%%%% Rule-Finding Abduction with Analogy             %%%%%%%%%%
%%%%%%%%%%      using Partially Isomorphic Generalization %%%%%%%%%%

ab_ana(TG,TG,Pair,WorldTarget) :-
        functor(TG,Pred,Arity),functor(BG,Pred,Arity),
        world(WorldBase,WorldTarget),
        fact(WorldBase,(BG:-true)),
        analogy((TG:-true),(BG:-true),Pair).
```

```
ab_ana(TG,TGs,Pair,WorldTarget) :-
        functor(TG,Pred,Arity),functor(BG,Pred,Arity),
        provable(TG,BG,TGs,BGs,WorldTarget),
        not TG==TGs,
        analogy((TG:-TGs),(BG:-BGs),Pair).

provable(TG,BG,TL,BL,WorldTarget) :-
        rule(TG,BG,TGs,BGs,WorldTarget),
        provable(TGs,BGs,TL,BL,WorldTarget).

provable((TG,TGs),(BG,BGs),(TL,TLs),(BL,BLs),WorldTarget) :-
        provable(TG,BG,TL,BL,WorldTarget),
        provable(TGs,BGs,TLs,BLs,WorldTarget).

provable(TG,BG,TG,BG,WorldTarget) :-
        world(WorldBase,WorldTarget),fact(WorldBase,(BG:-true)),!.

rule(TG,BG,TGs,BGs,WorldTarget) :-
        world(WorldBase,WorldTarget),
        fact(WorldBase,(BG:-BGs)),
        not BGs=true,
        pig_rule((BG:-BGs),(PG:-PGs)),
        copy((PG:-PGs),(TG:-TGs)).

copy(Old,New) :-
        (retract('$maker'(_)),fail;
         assert('$maker'(Old)),retract('$maker'(New))),!.

analogy((TG:-true),(BG:-true),Pairing) :-
        pig(BG,GG),analogy(TG,GG,BG,Pairing),!.

analogy((TG:-TL),(BG:-BL),Pairing) :-
        pig_rule((BG:-BL),(GG:-GL)),
        analogy((TG:-TL),(GG:-GL),(BG:-BL),Pairing).

analogy((TL1,TL2),(GL1,GL2),(BL1,BL2),Pairing) :-
        analogy(TL1,GL1,BL1,Pairing1),
        analogy(TL2,GL2,BL2,Pairing2),!,
        append(Pairing1,Pairing2,Pairing).

analogy(TA,GA,BA,P) :-
        lg(TA,GA,LGG,[],S1,[],S2),
        lg(BA,GA,LGG,[],S3,[],S4),
        pairing1(P,S1,S2,S3,S4).

pairing1([],[],[],[],[]).
pairing1([T1--T2|X],[V<-T2|S1],[V<-W|S2],[V<-T1|S3],[V<-W|S4]) :-
        pairing1(X,S1,S2,S3,S4).

%%%%%%%%% least generalization by Flach %%%%%%%%%
```

```
lg(Term1,Term2,Term1,S1,S1,S2,S2) :-
        Term1 == Term2,!.
lg(Term1,Term2,V,S1,S1,S2,S2) :-
        subs_lookup(S1,S2,Term1,Term2,V),!.
lg(Term1,Term2,Term,S10,S1,S20,S2) :-
        nonvar(Term1),nonvar(Term2),
        functor(Term1,F,N),functor(Term2,F,N),!,
        functor(Term,F,N),
        lg_args(N,Term1,Term2,Term,S10,S1,S20,S2).
lg(Term1,Term2,V,S10,[V<-Term1|S10],S20,[V<-Term2|S20]).

lg_args(0,Term1,Term2,Term,S1,S1,S2,S2).
lg_args(N,Term1,Term2,Term,S10,S1,S20,S2) :-
        N>0,N1 is N-1,
        arg(N,Term1,Arg1),
        arg(N,Term2,Arg2),
        arg(N,Term,Arg),
        lg(Arg1,Arg2,Arg,S10,S11,S20,S21),
        lg_args(N1,Term1,Term2,Term,S11,S1,S21,S2).

subs_lookup([V<-T1|Subs1],[V<-T2|Subs2],Term1,Term2,V) :-
        T1 == Term1,T2 == Term2,!.
subs_lookup([S1|Subs1],[S2|Subs2],Term1,Term2,V) :-
        subs_lookup(Subs1,Subs2,Term1,Term2,V).

%%%%%%%%% Partially Isomorphic Generalization %%%%%%%%%
%%%%%%%%%     by Hirowatari and Arikawa       %%%%%%%%%

pig(Atom1,Atom) :-
        nonvar(Atom1),functor(Atom1,F,N),N>0,!,
        functor(Atom,F,N),pig(N,Atom1,Atom,0).

pig(0,_,_,_) :-!.
pig(N,Atom1,Atom,Cnt1) :-
        Cnt1=0,
        arg(N,Atom1,Arg1),
        search(Arg1,Arg,Atom1,Cnt1,Cnt),
        (nonvar(Arg) -> pig_A(N,Arg,Atom1,Atom,Cnt) ;
                pig_B(N,Arg1,Atom1,Atom)),!.
pig(N,Atom1,Atom,Cnt1) :-
        arg(N,Atom1,Arg1),
        search1(Arg1,Arg2,Atom1,Cnt1,Cnt2),
        search(Arg2,Arg,Atom1,Cnt1,Cnt),
        (nonvar(Arg) -> pig_A(N,Arg,Atom1,Atom,Cnt) ;
                pig_B(N,Arg1,Atom1,Atom)),!.

pig_A(N,Arg,Atom1,Atom,Cnt) :-
        replace_term(Atom1,Arg,Var,Atom2),pig(N,Atom2,Atom,Cnt).

pig_B(N,Arg,Atom1,Atom) :-
M is N-1,arg(N,Atom,Arg),pig(M,Atom1,Atom,0).
```

```
search1(Term1,Term,Atom,Cnt1,Cnt) :- var(Term1),!.
search1(Term1,Term,Atom,Cnt1,Cnt) :-
functor(Term1,F,N),check1(N,Term1),Cnt is Cnt1-1,!.
search1(Term1,Term,Atom,Cnt1,Cnt) :-
functor(Term1,F,N),search2(N,Term1,Term,Atom,Cnt1,Cnt).

search2(N,Term1,Term,Atom,Cnt1,Cnt) :-
        arg(N,Term1,Arg1),
        search1(Arg1,Term,Atom,Cnt1,Cnt2),
        (Cnt2=0,Term is Arg1 ;
                M is N-1,search2(M,Term1,Term,Atom,Cnt2,Cnt)).

search(Term1,Term,Atom,Cnt1,Cnt) :-
        (check(Term1,Term,Atom,Cnt1,Cnt) ;
                search_A(Term1,Term,Atom,Cnt1,Cnt)).

search_A(Term1,Term,Atom,Cnt,Cnt) :- var(Term1),!.
search_A(Term1,Term,Atom,Cnt1,Cnt) :-
        functor(Term1,F,N),check1(N,Term1),Cnt is Cnt1+1,!.
search_A(Term1,Term,Atom,Cnt1,Cnt) :-
        functor(Term1,F,N),search_B(N,Term1,Term,Atom,Cnt1,Cnt).

search_B(0,_,_,_,Cnt,Cnt) :- !.
search_B(N,Term1,Term,Atom,Cnt1,Cnt) :-
        arg(N,Term1,Arg1),search(Arg1,Term,Atom,Cnt1,Cnt2),
        (nonvar(Term) ;
                M is N-1,search_B(M,Term1,Term,Atom,Cnt2,Cnt)).

check(Term,Term,Atom,Cnt,Cnt) :- atomic(Term),!.
check(Term,Term,Atom,Cnt,Cnt) :-
        nonvar(Term),functor(Term,F,N),
        check2(N,Term,[],List1),
        replace_term(Atom,Term,Var,Atom1),
        var_set(Atom1,List2),
        intersection(List1,List2,List3),List3=[].

check1(0,_) :- !.
check1(N,Term) :- arg(N,Term,Arg),var(Arg),!,M is N-1,check1(M,Term).

check2(0,_,List,List) :-!.
check2(N,Term,List1,List) :-
        arg(N,Term,Arg),var(Arg),!,
union(Arg,List1,List2),
        M is N-1,check2(M,Term,List2,List).

var_set(Atom,List) :- functor(Atom,F,N),var_set(N,Atom,[],List).

var_set(0,_,List,List) :- !.
var_set(N,Atom,List1,List) :-
        arg(N,Atom,Arg),
```

```
        var_check(Arg,List1,List2),
        M is N-1,
        var_set(M,Atom,List2,List).

var_check(Term,List,List) :- atomic(Term),!.
var_check(Term,List1,List) :- var(Term),!,union(Term,List1,List).
var_check(Term,List1,List) :-
        functor(Term,F,N),
        arg(N,Term,Arg),
        var_check(Arg,List1,List2),
        M is N-1,
        var_set(M,Term,List2,List).

intersection([],X,[]).
intersection([X|R],Y,[X|Z]) :- member(X,Y),!,intersection(R,Y,Z).
intersection([X|R],Y,Z) :- intersection(R,Y,Z).

union(X,Y,Y) :- member(X,Y),!.
union(X,Y,[X|Z]) :- Y=Z.

member(X,[Y|_]) :- X==Y,!.
member(X,[_|Y]) :- member(X,Y).

%%%%%%%%%% Partially Isomorphic Generalization of Rule %%%%%%%%%%

pig_rule(Rule,PigRule) :-
        functor(Rule,:-,2),
        arg(1,Rule,Head),
        arg(2,Rule,Body),!,
        pig_atoms((Head,Body),(PigHead,PigBody)),!,
        functor(PigRule,:-,2),
        arg(1,PigRule,PigHead),
        arg(2,PigRule,PigBody).

pig_atoms(Atoms,PigAtoms) :-
        list(Atoms,Pred,Num,List),
        append([newatom],List,Lists),
        NewAtom =.. Lists,
        pig(NewAtom,PigNewAtom),!,
        PigNewAtom =.. NewLists,
        append([newatom],NewList,NewLists),
        list(PigAtoms,Pred,Num,NewList).

list((Atom,Atoms),[Pred1|Pred2],[Num1|Num2],List) :-
        list(Atom,Pred1,Num1,List1),
        list(Atoms,Pred2,Num2,List2),
        append(List1,List2,List).
list(Atom,F,N,List) :-
        functor(Atom,F,N),
        Atom =.. List1,
        append([F],List,List1).
```

```
append([],A,A) :- !.
append([X|A],B,[X|C]) :- append(A,B,C).
```

8.3 PROPOSE

We also describe our realization of the algorithm *PROPOSE* in Section 6.4 as the following Prolog program.

```
%%%%%%%%%%%%% Algorithm PROPOSE %%%%%%%%%%%%%

propose(Fact,Atoms,(NewHead:-NewBody)) :-
           !,
           while_loop(Fact,(NewHead:-TmpNewBody)),
           (NewHead == TmpNewBody ->
               NewBody = true,
               const_gen(Fact,TmpHead1),
               list_gen_main(TmpHead1,TmpHead2),
               replace_term(TmpHead2,list,_,NewHead),!;
               NewBody = TmpNewBody,
               assert((NewHead:-NewBody)),
               rs_abd(Fact,Atoms),
               retract((NewHead:-NewBody))).

while_loop(X,(NewHead:-NewBody)) :-
           list_gen(X,List),
           member_of(W,List),
           common_list_var(W,NW),
           const_gen(NW,Head),
           create(Head,Body),
           create2(Head,Body,NewHead,NewBody).

%%%% rule-selecting abduction %%%%

rs_abd(Goal,Leaves) :- clause(Goal,Leaves). %% one step hypothesis

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%% construction of recursive program %%%%%%%%%%%%%%%%%%%
%%%% create(Head,Body) :-
%%%%     construct pred(Head)-reducing clause clause(Head,Body)

%%%% create a recursive part with reducing_rec/2
create(Head,Body) :-
           functor(Head,Pred,Arg),
           functor(Body,Pred,Arg),
           reducing(Head,Body,Arg).

reducing(_,_,0) :- !.
```

```
reducing(Head,Body,Arg) :-
           arg(Arg,Head,Term1),
           (var(Term1) -> arg(Arg,Body,Term1) ;
               proper_subterm(Term1,Term2),arg(Arg,Body,Term2)),
           Arg1 is Arg-1,
           reducing(Head,Body,Arg1).

%%%%% proper_subterm(Term1,Term2)
%%%%%   :- find all proper subterm Term2 of Term1
proper_subterm([X|Y],Y).
proper_subterm([],[]) :- !.

%%%%% create a recursive part with reducing_rec/2
create2(Head,Body,NewHead,NewBody) :-
           functor(Head,Pred,Arg),
           functor(Body,Pred,Arg),
           functor(NewHead,Pred,Arg),
           functor(NewBody,Pred,Arg),
           create2(Head,Body,NewHead,NewBody,Arg).

create2(Head,Body,NewHead,NewBody,0) :- !.
create2(Head,Body,NewHead,NewBody,Arg) :-
           arg(Arg,Head,TermHead),
           arg(Arg,Body,TermBody),
           replace_term(TermBody,TermBody,X,TermNewBody),
           replace_term(TermHead,TermBody,X,TermNewHead),
           arg(Arg,NewHead,TermNewHead),
           arg(Arg,NewBody,TermNewBody),
           Arg1 is Arg-1,
           create2(Head,Body,NewHead,NewBody,Arg1).

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%% list_gen(Term1,GenTerm)
%%%%% common (sub)term -> one variable

list_gen(Formula,List) :-
           setof(GenForm,list_gen_main(Formula,GenForm),TmpList),
           set(TmpList,List).

list_gen_main(Formula,GenForm) :-
           functor(Formula,Pred,Arg),
           between(1,I,Arg),
           between(1,J,Arg),
           I \= J,
           arg(I,Formula,List1),
           arg(J,Formula,List2),
           (sublist(List1,List2),List1 \== [],
           replace_term(Formula,List1,list,GenForm)) ; GenForm=Formula ,!.

sublist(X,X) :- list(X).
sublist(X,[W|Y]) :- sublist(X,Y).
```

```
list([]).
list([X|Y]) :- list(Y).

set(X,Y) :- set(X,[],Y).
set([Element|TmpList],X,List) :-
          (NewElement = Element  -> set(TmpList,[NewElement|X],List) ;
          set(TmpList,X,List)).
set([],X,X).

member_of(X,[X|_]).
member_of(X,[Y|Z]) :- member_of(X,Z).

pop(X,[X|L],L).
push(X,L,[X|L]).

common_list_var(A,B) :-
          functor(A,F,N),
          functor(B,F,N),
          common_list_var(A,B,N).
common_list_var(A,B,0) :- !.
common_list_var(A,B,N) :-
          arg(N,A,Term1),
          replace_term(Term1,list,_,Term2),
          arg(N,B,Term2),
          M is N-1,
          common_list_var(A,B,M).

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%% const_gen(Term1,Term2) :-
%%%%%     Term2 is a term which the constant symbols in Term1
%%%%%     is replaced by variable, in particular, same constant
%%%%%     symbol is replaced by same variable.

const_gen(Term1,Term2) :-
          clear,
          assert(constlist([])),
          assert(data(1)),
          gen(Term1,Term2),!.

gen(Term1,Term2) :- \+search_const(Term1) -> replace(Term1,Term2).

replace(Term1,Term2) :-
          constlist(List),
          length(List,N),
          replace(N,Term1,Term2).
replace([],[]) :- !.
replace(Term1,Term1) :- !.

replace(0,Term1,Term1) :- !.
replace(N,Term1,Term2) :-
```

```
            constlist(List),
            (N > 0 ->
               (pop(Const,List,NewList),
                   retract(constlist(List)),
                   replace_term(Term1,Const,Var,Term),
                   assert(constlist(NewList)),
                   M is N-1,
                   replace(M,Term,Term2))).

search_const(Term) :-
            functor(Term,F,Arg),
            data(N),
            (Arg >= N -> search_const(N,Term)),
            M is N+1,
            retract(data(N)),
            assert(data(M)),
            search_const(Term).

search_const(N,Term) :- (var(Term);Term=[]),!.
search_const(N,Term) :-
            functor(Term,F,M),
            (between(1,L,M),arg(L,Term,Subterm),search_const(L,Subterm)).
search_const(N,Term) :-
            functor(Term,F,0),
            constlist(OldList),
            (\+member_cut(Term,OldList) -> push(Term,OldList,NewList)),
            retract(constlist(OldList)),
            assert(constlist(NewList)).

between(Low,Low,Up).
between(Low,I,Up) :- between(Low,Old,Up),I is Old+1,((I > Up,!,fail);true).

member_cut(X,[X|Y]) :- !.
member_cut(X,[Y|Z]) :- member_cut(X,Z),!.

clear :- retractall(data(_)),retractall(constlist(_)).
retractall(X) :- retract(X),fail,!.
retractall(_).
```