Analogical Logic Program Synthesis from Examples

Ken Sadohara and Makoto Haraguchi

Department of Systems Science Tokyo Institute of Technology
4259 Nagatsuta, Midori-ku, Yokohama 227, Japan
E-mail:sadohara@sys.titech.ac.jp and makoto@sys.titech.ac.jp

Abstract. The purpose of this paper is to present a theory and an algorithm for analogical logic program synthesis from examples. Given a source program and examples, the task of our algorithm is to find a program which explains the examples correctly and is similar to the source program. Although we can define a notion of similarity in various ways, we consider a class of similarities from the viewpoint of how examples are explained by a program. In a word, two programs are said to be similar if they share a common explanation structure at an abstract level. Using this notion of similarity, we formalize an analogical logic program synthesis and show that our algorithm based on a framework of model inference can identify a desired program.

1 Introduction

This paper is concerned with Logic Program Synthesis from examples (LPS). LPS is generally regarded as one of the frameworks of learning from examples, and has been widely studied by many authors [5, 11, 9]. Any LPS system receives an example set from a target program, and tries to find the target program in a huge search space. Some researchers [13, 10] have pointed out that the use of analogy might be helpful in LPS. Such Analogical Logic Program Synthesis (ALPS) systems try to find a correct program[1] which is similar to a source program. They have considered that analogy is useful for reducing the search space to a space of "similar programs". However, no studies have established or proved that the use of analogy really makes such a contribution. Furthermore, there exists a criticism that analogy makes LPS more difficult. Because the ALPS system must find not only a correct program but also a similarity; that is, the system must find which program is appropriate as a source program and how the correct program is similar to the source program. Even if we admit such a disadvantage, we think that ALPS is worth investigating because of another role of analogy which is pointed out in [2]. It is the role as a device to shift a bias. The bias is a tendency to select a class of programs from the correct programs. Therefore, an ALPS system inheriting the role of analogy enables us to lead the system to identify a desired program , depending on which program we give the system as a source program. For example, let us consider programs for sorting lists. An

[1] A correct program means a program which can explain given examples correctly.

ALPS system might be able to identify *insert-sort* program rather than *naive-sort* program, provided we give the system a source program to which *insert-sort* is similar, such as *natural-number*[2]. To borrow from Michalski's word [4], we can change a *preference criterion* of programs dynamically. This aspect of ALPS seems to be important from a viewpoint of the change of representation (the program transformation). In the last section, we briefly discuss this viewpoint. These reasons mentioned above motivate us to investigate ALPS.

In ALPS, a notion of similarities between logic programs plays a crucial role. In the literatures [7, 10], the authors have precisely defined that notion from which only small classes of similarities are derived. On the other hand, in the literature [13], the authors have considered a wider class of similarities that has no firm theoretical basis. Because of the lack of an appropriate theoretical basis on similarities between programs, we have not been able to evaluate ALPS's usefulness. A purpose of this paper is to present a class of non-trivial similarities between programs with a firm theoretical basis.

There may exist various kinds of similarities between logic programs. According to [7] and [10], a source program is regarded as a second-order schema. A program is considered to be similar to the source program, provided it is an instance of the schema. For example, a program

$$Q = \left\{ \begin{array}{l} \texttt{ancestor(X,Y)} \leftarrow \texttt{parent(X,Y)} \\ \texttt{ancestor(X,Y)} \leftarrow \texttt{ancestor(X,Z),parent(Z,Y)} \end{array} \right\}$$

is similar to a source program

$$P = \left\{ \begin{array}{l} \texttt{connected(X,Y)} \leftarrow \texttt{link(X,Y)} \\ \texttt{connected(X,Y)} \leftarrow \texttt{connected(X,Z),link(Z,Y)} \end{array} \right\}$$

because Q is an instance of P, where `connected` and `link` are instantiated to `ancestor` and `parent`, respectively. This kind of similarity can be thus defined as a symbol-to-symbol mapping, and is therefore too much dependent on their surface syntax. On the other hand, we can observe a more internal similarity between the above programs. In fact, they share a common explanation structure. If any ground atom `ancestor(`x`,`y`)` corresponds to the ground atom `connected(`x`,`y`)` and any ground atom `parent(`x`,`y`)` corresponds to the ground atom `link(`x`,`y`)`, then any proof of any ground atom `ancestor(`x`,`y`)` is similar to a proof of the ground atom `connected(`x`,`y`)`. Considering in this way, we justify our intuition that a logic program

$$\textit{reverse} = \left\{ \begin{array}{l} \texttt{rev([],[])} \leftarrow \\ \texttt{rev([A|X],Y)} \leftarrow \texttt{rev(X,Z),append(Z,[A],Y)} \end{array} \right\}$$

is similar to a logic program

$$\textit{natural-number} = \left\{ \begin{array}{l} \texttt{nn(0)} \leftarrow \\ \texttt{nn(s(X))} \leftarrow \texttt{nn(X)} \end{array} \right\}$$

[2] These programs are similar because they have a similar recursive structure. The programs mentioned here are in [12].

although *reverse* is not an instance of *natural-number*. Because examples of *reverse* are explained in the similar way (not in the same way) as examples of *natural-number* under the following correspondence between rev and nn. Any ground atom $\mathtt{rev}(x,y)$ corresponds to the ground atom $\mathtt{nn}(\mathtt{s}^n(\mathtt{0}))$, where x is a list whose length is $n \geq 0$ and $\mathtt{s}^n(\mathtt{0})$ is an abbreviation of $\underbrace{\mathtt{s}(\mathtt{s}(\cdots\mathtt{s}(}_{n}\mathtt{0})\cdots))$.

In this paper, we formalize such a non-trivial similarity from the viewpoint of how the examples are explained. Although the paper [13] has considered this kind of similarity, it has not formalized the similarity completely.

The ALPS problem which we consider here is stated informally as follows.

Given

- a source program P and
- a set of positive examples E^+ and a set of negative examples E^-, where the examples are ground atoms,

Find a target program Q such that $Q \vdash E^+$, $Q \nvdash E^-$ and Q is similar to P.

Firstly we present a formal theory on ALPS, especially a theory on similarities between logic programs. Secondly we show an algorithm for ALPS which is obtained by extending Shapiro's incremental model inference algorithm [11], and prove that the algorithm identifies a solution in the limit.

2 Preliminaries

In this paper, concepts for logic programs are based on [3], unless stated otherwise. For any clause C and D, we define pre-order $C \geq D$, which is called θ-subsumption, iff there exists a substitution θ such that $C\theta \subseteq D$. In addition, for any set of clauses S and T, we define $S \geq T$ iff for any clause C in T, there exists a clause D in S such that $D \geq C$. For any definite clause $C = A \leftarrow B_1, \ldots, B_n$, $\mathrm{He}(C)$ denotes the positive literal A and $\mathrm{Bo}(C)$ denotes the set of atoms $\{B_1, \ldots, B_n\}$. A ground goal is a clause of the form $\leftarrow B_1, \ldots, B_n$, where each $B_i (1 \leq i \leq n)$ is a ground atom. The symbol $\bot$ denotes the empty clause. In the remainder of this paper, we consider a logic program as a finite set of definite clauses whose lengths are finite.

For any function symbol or any predicate symbol s, $\sharp s$ denotes the arity of s. For any set of function symbols Σ and any set of predicate symbols Π, we call the pair $\langle \Sigma, \Pi \rangle$ a vocabulary. For any logic program P, $\mathrm{V}(P)$ denotes a vocabulary $\langle \Sigma, \Pi \rangle$, where Σ is a set of function symbols including function symbols occurring in P and Π a set of predicate symbols including predicate symbols occurring in P. Moreover we assume that Σ and Π are finite sets. When we consider several logic programs, we assume that their vocabularies are disjoint.

Throughout this paper, we assume a set of variables $\mathcal{V}$, and a set of special constants $\mathcal{C}$ whose elements never appear in any program. We also assume that $\mathcal{V}$ and $\mathcal{C}$ contain enough elements.

For any set of function symbols Σ and any set of predicate symbols Π, $\mathrm{Trm}(\Sigma)$ denotes the set of terms constructed from Σ, $\mathcal{V}$ and $\mathcal{C}$. Likewise, $\mathrm{Sub}(\Sigma)$, $\mathrm{Atm}(\Sigma, \Pi)$ and $\mathrm{Cls}(\Sigma, \Pi)$ denote the set of substitutions constructed from $\mathrm{Trm}(\Sigma)$, the set of atoms constructed from $\mathrm{Trm}(\Sigma)$ and Π, and the set of clauses constructed from $\mathrm{Atm}(\Sigma, \Pi)$ respectively. $\mathrm{Exp}(\Sigma, \Pi)$ denotes $\mathrm{Trm}(\Sigma) \cup \mathrm{Sub}(\Sigma) \cup \mathrm{Atm}(\Sigma, \Pi) \cup \mathrm{Cls}(\Sigma, \Pi)$ and we call each elements of this set an expression. In addition, $\mathrm{Trm}(P)$, $\mathrm{Sub}(P)$, $\mathrm{Atm}(P)$ and $\mathrm{Cls}(P)$ denote $\mathrm{Trm}(\Sigma)$, $\mathrm{Sub}(\Sigma)$, $\mathrm{Atm}(\Sigma, \Pi)$ and $\mathrm{Cls}(\Sigma, \Pi)$ respectively for any logic program P, where $\mathrm{V}(P) = \langle \Sigma, \Pi \rangle$.

For any logic program P, $\mathrm{B}(P)$ denotes the Herbrand base which is the set of ground atoms constructed from $\mathrm{V}(P)$ and $\mathcal{C}$. A mapping $\mathrm{T}_P : 2^{\mathrm{B}(P)} \to 2^{\mathrm{B}(P)}$ is defined as follows. For any $I \subseteq \mathrm{B}(P)$,

$$\mathrm{T}_P(I) = \left\{ A \in \mathrm{B}(P) \;\middle|\; \begin{array}{l} \text{There exists a ground instance } A \leftarrow B_1, \ldots, B_n \\ \text{of a clause in } P \text{ such that } \{B_1, \ldots, B_n\} \subseteq I \end{array} \right\}$$

Then,

$$\begin{aligned} &\mathrm{T}_P \uparrow 0 = \emptyset \\ &\mathrm{T}_P \uparrow n = \mathrm{T}_P(\mathrm{T}_P \uparrow (n-1)) \text{ for any positive integer } n \\ &\mathrm{T}_P \uparrow \omega = \textstyle\bigcup_{n<\omega} \mathrm{T}_P \uparrow n \text{ for the first transfinite ordinal } \omega \end{aligned}$$

$\mathrm{M}(P)$ denotes the least Herbrand model of P. It is known that $\mathrm{M}(P) = \mathrm{T}_P \uparrow \omega$.

For any mapping ϕ, $\mathcal{D}(\phi)$ denotes the domain of ϕ and $\phi \mid_S$ denotes the restriction of ϕ whose domain is $S \cap \mathcal{D}(\phi)$.

For any substitution $\{X_1/t_1, \ldots, X_n/t_n\}$, if each $t_i (1 \leq i \leq n)$ is a special constant then we call it a special substitution. In addition, a ground substitution θ is grounding substitution of a clause C if $C\theta$ is a ground clause.

3 Similarities between Logic Programs

There may exist various kinds of similarities between logic programs. In this section, we consider a class of similarities from the viewpoint of how examples are explained. This is because we think that programming can be viewed as fixing the way of explanation of examples. Formally, it can be viewed as giving a *Primitive Explanation Structure* defined as follows.

Definition 1. Let P be a logic program. The following relation $R \subseteq \mathrm{B}(P) \times 2^{\mathrm{B}(P)}$ is called *Primitive Explanation Structure (PES)* of P.

$$R = \left\{ (A, S) \;\middle|\; \begin{array}{l} A \leftarrow B_1, \ldots, B_n \text{ is a ground instance of a clause in } P, \\ \{B_1, \ldots, B_n\} \subseteq S \subseteq \mathrm{B}(P) \end{array} \right\}$$

Because generalizing this possibly infinite PES, we get a finite expression of the structure and this is a program. Therefore, we consider similarities between programs based on this PES as follows: a target logic program Q is similar to a source logic program P if we can abstract the PES of Q into that of P. So, we first define a notion of abstraction relation between the PESs (the programs).

A partial mapping defined as follows enables to abstract the PES of the target program.

Definition 2. Let P and Q be logic programs. A partial mapping $\phi : \mathrm{B}(Q) \to \mathrm{B}(P)$ is called an *abstraction mapping from Q to P*. For $J \subseteq \mathrm{B}(Q)$, $\phi(J) = \{\phi(B) \mid B \in J \cap \mathcal{D}(\phi)\}$.

The following proposition shows properties of abstraction mappings.

Proposition 3. *Let P and Q be logic programs. Let ϕ be an abstraction mapping from Q to P. For any $I, J \subseteq \mathrm{B}(Q)$,*

$$I \subseteq J \Rightarrow \phi(I) \subseteq \phi(J) \qquad \phi(I \cup J) = \phi(I) \cup \phi(J)$$

Now, using the abstraction mappings, we define a class of abstraction relations between programs. The following definition says that for any logic programs P and Q, if we can abstract the PES of Q into that of P then P is more abstract than Q. That is, P is more abstract than Q when there exists an abstraction mapping ϕ from Q to P such that for any $\langle A, S\rangle$ in PES of Q, $\langle \phi(A), \phi(S)\rangle$ is in PES of P.

Definition 4. Let P and Q be logic programs. Let ϕ be an abstraction mapping from Q to P. P is *more abstract than Q w.r.t. ϕ* iff

$$\mathrm{T}_P(\phi(I)) \supseteq \phi(\mathrm{T}_Q(I))$$

for any $I \subseteq \mathrm{B}(Q)$.

From the definition, we have the following theorem.

Theorem 5. *Let P and Q be logic programs. Let ϕ be an abstraction mapping from Q to P. If P is more abstract than Q w.r.t. ϕ then* $\mathrm{M}(P) \supseteq \phi(\mathrm{M}(Q))$.

The following proposition shows an equivalence condition of the abstraction relation. This is useful for decision whether there exists an abstraction relation w.r.t a given abstraction mapping.

Proposition 6. *Let P and Q be logic programs. P is more abstract than Q w.r.t. an abstraction mapping ϕ iff for any ground instance C of any clause in Q such that* $\mathrm{He}(C) \in \mathcal{D}(\phi)$, *there exists a ground instance D of a clause in P such that* $\mathrm{He}(D) = \phi(\mathrm{He}(C))$ *and* $\mathrm{Bo}(D) \subseteq \phi(\mathrm{Bo}(C))$.

Example 1. Using Proposition 6, we can confirm the following abstraction relations.

For programs

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

and

$$plus = \left\{ \begin{array}{l} \texttt{plus(0,X,X)} \leftarrow \\ \texttt{plus(s(X),Y,s(Z))} \leftarrow \texttt{plus(X,Y,Z)} \end{array} \right\}$$

Let ϕ_1 be an abstraction mapping such that

$$\phi_1(\mathtt{append}(x,y,z)) = \mathtt{plus}(\mathtt{s}^n(\mathtt{0}),\mathtt{s}^m(\mathtt{0}),\mathtt{s}^{n+m}(\mathtt{0}))$$

where x, y and z are lists whose lengths are $n \geq 0$, $m \geq 0$ and $n+m$ respectively. Then, *plus* is more abstract than *append* w.r.t. ϕ_1.

For *natural-number* and *reverse* in the introduction, Let ϕ_2 be an abstraction mapping such that

$$\phi_2(\mathtt{rev}(x,y)) = \mathtt{nn}(\mathtt{s}^n(\mathtt{0}))$$

where x is a list whose length is $n \geq 0$. Then, *natural-number* is more abstract than *reverse* w.r.t. ϕ_2.

For *natural-number* and

$$\mathit{parity} = \left\{ \begin{array}{l} \mathtt{even(0)} \leftarrow \\ \mathtt{odd(s(X))} \leftarrow \mathtt{even(X)} \\ \mathtt{even(s(X))} \leftarrow \mathtt{odd(X)} \end{array} \right\}$$

if we define an abstraction mapping ϕ_3 as

$$\phi_3(\mathtt{even}(x)) = \mathtt{nn}(x), \phi_3(\mathtt{odd}(x)) = \mathtt{nn}(x)$$

then *natural-number* is more abstract than *parity* w.r.t. ϕ_3.

Now, let us consider how the abstractions defined above affect a proof tree. The consideration amplifies abstraction of the explanation structure mentioned in introduction.

Definition 7. Let P be a logic program. *Ground refutation node in* P is a pair $\langle G,\ C\rangle$, where $G \subseteq \mathrm{B}(P)$ and C is a ground instance of a clause in P such that $\mathrm{He}(C) \in G$.

Definition 8. Let P be a logic program and G a ground goal. *Ground refutation of* G *in* P is a finite sequence $\langle G_1,\ C_1\rangle \cdots \langle G_n,\ C_n\rangle$ of ground refutation nodes in P, where

1. $G_1 = \mathrm{Bo}(G)$.
2. For all i $(1 \leq i \leq n-1)$, $G_{i+1} = G_i \setminus \{\mathrm{He}(C_i)\} \cup \mathrm{Bo}(C_i)$.
3. $G_n = \{A\}$ and $C_n = A \leftarrow$, where $A \in \mathrm{B}(P)$.

The following theorem describes how the abstractions affect ground refutations in a target program. For any given ground refutation in a target program, the procedure in the theorem abstracts it into a ground refutation in a source program.

Theorem 9. *Let P and Q be logic programs. Assume P is more abstract than Q w.r.t. an abstraction mapping ϕ. For any ground goal G and any ground refutation GR of G in Q, any sequence GR' obtained by the following procedure is a ground refutation of $\phi(G)$ in P, where $\phi(G)$ denotes $\leftarrow B_1,\ldots,B_n$ $(\{B_1,\ldots,B_n\} = \phi(\mathrm{Bo}(G)))$ and $\phi(G) \neq \bot$.*

1. *Assume* $GR = \langle G_1,\ C_1\rangle \cdots \langle G_n,\ C_n\rangle$. *Let* GR' *be the empty sequence* ε *and* G'_1 *be* G_1.
2. *For all* $1 \leq i \leq n$, *if* $\mathrm{He}(C_i) \in \mathcal{D}(\phi) \cap G'_i$ *then*
   (a) *non-deterministically choose a ground instance* D *of a clause in* P *such that* $\mathrm{He}(D) = \phi(\mathrm{He}(C_i))$ *and* $\mathrm{Bo}(D) \subseteq \phi(\mathrm{Bo}(C_i))$, *where there always exists such a clause because of Proposition 6,*
   (b) $G'_{i+1} \leftarrow G'_i \setminus \phi^{-1}(\{\mathrm{He}(D)\}) \cup \{A \in \mathrm{Bo}(C_i) \mid \phi(A) \in \mathrm{Bo}(D)\}$, *and*
   (c) *let* GR' *be the concatenation of* GR' *and* $\langle \phi(G'_i),\ D\rangle$.
   else let G'_{i+1} *be* G'_i.

In the procedure, ϕ^{-1} *is defined as* $\phi^{-1}(A) = \{B \mid \phi(B) = A\}$ *and for any ground clause* C *such that* $\mathrm{He}(C) \in \mathcal{D}(\phi)$, $\phi(C)$ *denotes* $\phi(\mathrm{He}(C)) \leftarrow B_1, \ldots, B_n$ $(\{B_1, \ldots, B_n\} = \phi(\mathrm{Bo}(C)))$.

There exist three types of abstraction of resolution by the above procedure. The following example shows them.

Example 2. Let a target program Q and a source program P as follows.

$$Q = \left\{ \begin{array}{l} s_1 \leftarrow \\ s_2 \leftarrow \\ t \leftarrow \\ r \leftarrow \\ p \leftarrow q, r \\ q \leftarrow s_1, s_2, t \end{array} \right\} \quad P = \left\{ \begin{array}{l} s' \leftarrow \\ t' \leftarrow \\ p' \leftarrow q' \\ q' \leftarrow s' \end{array} \right\}$$

Let an abstraction mapping ϕ be $\phi(s_1) = \phi(s_2) = s'$, $\phi(t) = t'$, $\phi(p) = p'$ and $\phi(q) = q'$. Then P is more abstract than Q w.r.t. ϕ.

In the Fig. 1, GR is a ground refutation of a goal $\leftarrow p$ in Q and GR' is a ground refutation of a goal $\leftarrow p'$ in P obtained by the procedure stated above, where GR and GR' are represented by ordered binary tree. There exist three types of abstraction of resolution. The first one stems from the partiality of the abstraction mapping ϕ: the resolution (a) is abstracted because $r \notin \mathcal{D}(\phi)$. The second one stems from that ϕ is not injective: the resolution (b_1) and (b_2) are simplified into (b') because $\phi(s_1) = \phi(s_2) = s'$. The last one stems from that the image of a ground clause of Q is weaker than the corresponding clause of P: the resolution (c) is abstracted because $\phi(q \leftarrow s_1, s_2, t) = q' \leftarrow s', t'$ is weaker than $q' \leftarrow s'$.

The notion of abstraction defined in this paper differs from the notions of abstraction in [6, 14, 1] at least in abstraction mapping's partiality. As far as we concern ground atoms, our notion of abstraction is a partial *TI-abstraction* as Theorem 5 shows. The difference comes from a difference of motivation: while their notions of abstraction are motivated by theorem-proving with abstraction, our notion of abstraction is motivated by extracting similarity with abstraction.

Using the class of abstraction relations between logic programs we have seen, we define a class of similarities between programs.

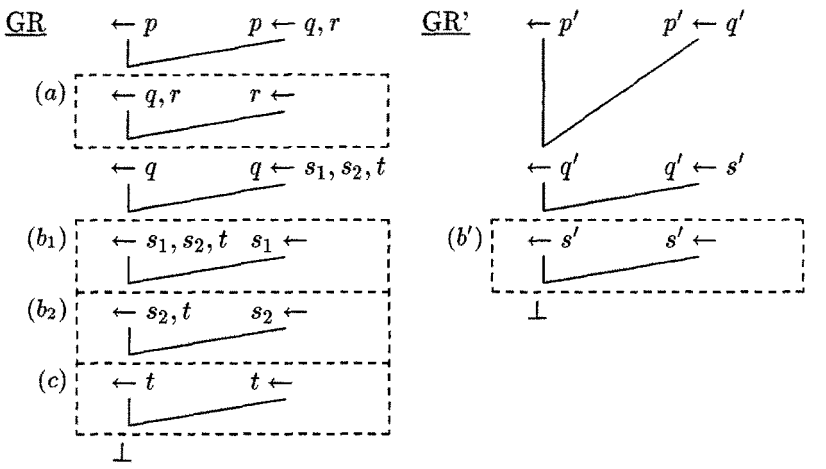

Fig. 1.

Definition 10. Let P and Q be logic programs. Let ϕ be an abstraction mapping from Q to P. Q is *similar to* P *w.r.t* ϕ iff P is more abstract than Q w.r.t. ϕ.

Q is *similar to* P iff there exists an abstraction mapping ϕ and Q is similar to P w.r.t. ϕ.

4 Restricted Similarities for Decidability

If the Herbrand base of a target program is finite then the PES is also finite and the decision whether the target program is similar to a source program w.r.t. an abstraction mapping is clearly decidable. Then, when the Herbrand base is infinite, is the decision decidable? This question is open. In this section, we restrict the class of the similarities defined in the previous section so that the decision is decidable even if the Herbrand base is infinite. By restricting the class of abstraction mappings in the preceding section, we restrict the class of the similarities. The restricted class of abstraction mappings can deal with the following abstractions, which are introduced in [6].

1. Renaming symbols abstraction. For a ground atom A, the predicate symbol and function symbols appearing in A are renamed in some systematic way. The renaming is not necessarily one-to-one. For example, ground atoms `cup(a)` and `bottle(a)` are renamed into `container(a)`.
2. Permuting arguments abstraction. For a ground atom A, the order of the arguments of the predicate symbol or function symbols appearing in A changes in some systematic way. For example, a ground atom `child(a,father(a))` is changed into `parent(father(a),a)`. In this example, renaming symbols abstraction is applied at the same time.
3. Deleting arguments abstraction. For a ground atom A, certain arguments of the predicate symbol or function symbols appearing in A are deleted in some

systematic way. Let us consider a ground atom append([a,b],[c],[a,b,c]). By deleting the second and the third arguments of the predicate symbol append and the first argument of the function symbol [_|_], and renaming append, [_|_] and [] into nn, s and 0 respectively, the ground atom is abstracted into nn(s(s(0))). Note that the propositional abstraction is a special case of this (all arguments of all predicate symbols are deleted).

Definition 11. Let Σ_1 and Σ_2 be sets of function symbols. Let Π_1 and Π_2 be sets of predicate symbols. Let $\mathcal{S}$ is the set of sequences of different natural numbers including the null sequence ε. A pair $\langle \phi^{\Sigma}, \phi^{\Pi} \rangle$ is called a *symbol-abstraction mapping*, where ϕ^{Σ} is a total mapping $\phi^{\Sigma} : \Sigma_1 \to \Sigma_2 \times \mathcal{S}$ and ϕ^{Π} is a partial mapping $\phi^{\Pi} : \Pi_1 \to \Pi_2 \times \mathcal{S}$, and they have the following properties.

- For any $f \in \Sigma_1$, $\phi^{\Sigma}(f) = \langle g, a_1 \cdots a_{\sharp g} \rangle$, where $\{a_1, \ldots, a_{\sharp g}\} \subseteq \{1, \ldots, \sharp f\}$.
- For any $p \in \mathcal{D}(\phi^{\Pi})$, $\phi^{\Pi}(p) = \langle q, a_1 \cdots a_{\sharp q} \rangle$, where $\{a_1, \ldots, a_{\sharp q}\} \subseteq \{1, \ldots, \sharp p\}$.

$\Phi^s(\langle \Sigma_1, \Pi_1 \rangle, \langle \Sigma_2, \Pi_2 \rangle)$ denotes the class of symbol-abstraction mappings

In the above definition, $\phi^{\Sigma}(f) = \langle g, a_1 \cdots a_m \rangle$ means that any term s whose function symbol is f is mapped to a term t whose function symbol is g and the a_i-th argument of s is mapped to the i-th argument of t for all $1 \leq i \leq m$.

Definition 12. For any symbol-abstraction mapping $\langle \phi^{\Sigma}, \phi^{\Pi} \rangle \in \Phi^s(\langle \Sigma_1, \Pi_1 \rangle, \langle \Sigma_2, \Pi_2 \rangle)$, a partial mapping $\phi : \mathrm{Exp}(\Sigma_1, \Pi_1) \to \mathrm{Exp}(\Sigma_2, \Pi_2)$ is defined as follows.

1. For any variable $X \in \mathcal{V}$, $\phi(X) = X$.
2. For any special constant $c \in C$, $\phi(c) = c$.
3. For any constant $c \in \Sigma_1$, $\phi(c) = c'$, where $\phi^{\Sigma}(c) = \langle c', \varepsilon \rangle$.
4. For any term $f(t_1, \ldots, t_n) \in \mathrm{Trm}(\Sigma_1)$, $\phi(f(t_1, \ldots, t_n)) = g(\phi(t_{a_1}), \ldots, \phi(t_{a_m}))$, where $\phi^{\Sigma}(f) = \langle g, a_1 \cdots a_m \rangle$.
5. For any atom $p(t_1, \ldots, t_n) \in \mathrm{Atm}(\Sigma_1, \Pi_1)$ such that $p \in \mathcal{D}(\phi^{\Pi})$, $\phi(p(t_1, \ldots, t_n)) = q(\phi(t_{a_1}), \ldots, \phi(t_{a_m}))$, where $\phi^{\Pi}(p) = \langle q, a_1 \cdots a_m \rangle$.
6. For any clause $C \in \mathrm{Cls}(\Sigma_1, \Pi_1)$ such that $\mathrm{He}(C) \in \mathcal{D}(\phi)$, $\phi(C) = \phi(\mathrm{He}(C)) \leftarrow \phi(B_1), \ldots, \phi(B_n)$, where $\{B_1, \ldots, B_n\} = \mathrm{Bo}(C) \cap \mathcal{D}(\phi)$.
7. For any substitution $\theta = \{X_1/s_1, \ldots, X_n/s_n\} \in \mathrm{Sub}(\Sigma_1)$, $\phi(\theta) = \{X_1/\phi(s_1), \ldots, X_n/\phi(s_n)\}$.

Moreover, for any set of clauses $S \subseteq \mathrm{Cls}(\Sigma_1, \Pi_1)$, $\phi(S) = \{\phi(C) \mid C \in S \cap \mathcal{D}(\phi)\}$. $\Phi(\langle \Sigma_1, \Pi_1 \rangle, \langle \Sigma_2, \Pi_2 \rangle)$ denotes the class of the mappings defined as above.

Let P and Q be logic programs. For any $\phi \in \Phi(\mathrm{V}(Q), \mathrm{V}(P))$, note that $\phi\,|_{\mathrm{B}(Q)}$ is an abstraction mapping from Q to P. So, we also call ϕ an abstraction mapping.

Example 3. Abstraction mappings ϕ_1, ϕ_2 and ϕ_3 introduced in Example 1 are obtained as follows.

$$
\begin{array}{lll}
\phi_1^{\Sigma}([]) = \langle 0, \varepsilon \rangle & \phi_2^{\Sigma}([]) = \langle 0, \varepsilon \rangle & \phi_3^{\Sigma}(0) = \langle 0, \varepsilon \rangle \\
\phi_1^{\Sigma}([\_|\_]) = \langle \mathtt{s}, 2 \rangle & \phi_2^{\Sigma}([\_|\_]) = \langle \mathtt{s}, 2 \rangle & \phi_3^{\Sigma}(\mathtt{s}) = \langle \mathtt{s}, 1 \rangle \\
\phi_1^{\Pi}(\mathtt{append}) = \langle \mathtt{plus}, 1 \cdot 2 \cdot 3 \rangle & \phi_2^{\Pi}(\mathtt{rev}) = \langle \mathtt{nn}, 1 \rangle & \phi_3^{\Pi}(\mathtt{even}) = \langle \mathtt{nn}, 1 \rangle \\
 & & \phi_3^{\Pi}(\mathtt{odd}) = \langle \mathtt{nn}, 1 \rangle
\end{array}
$$

Lemma 13. *Let Σ_1 and Σ_2 be sets of function symbols. Let Π_1 and Π_2 be sets of predicate symbols. Let ϕ be in $\Phi(\langle\Sigma_1,\Pi_1\rangle,\langle\Sigma_2,\Pi_2\rangle)$. For any $t \in \mathrm{Trm}(\Sigma_1)$, any $A \in \mathrm{Atm}(\Sigma_1,\Pi_1)$, any $S \subseteq \mathrm{Atm}(\Sigma_1,\Pi_1)$, and any $\theta \in \mathrm{Sub}(\Sigma_1)$,*

$$\phi(t\cdot\theta)=\phi(t)\cdot\phi(\theta) \quad \phi(A\cdot\theta)=\phi(A)\cdot\phi(\theta) \quad \phi(S\cdot\theta)=\phi(S)\cdot\phi(\theta)$$

Theorem 14. *Let P and Q be logic programs. Q is similar to P w.r.t. $\phi \in \Phi(\mathrm{V}(Q),\mathrm{V}(P))$ iff $P \geq \phi(Q)$.*

Since whether $P \geq \phi(Q)$ or not is decidable, the decision whether Q is similar to P w.r.t. ϕ is also decidable.

Example 4. For the abstraction mapping ϕ_2 in Example 3, *natural-number* $= \phi_2$(*reverse*). Therefore, *reverse* is similar to *natural-number* w.r.t. ϕ_2. Likewise, we can confirm similarities w.r.t. ϕ_1 and ϕ_3.

5 ALPS Algorithm

In this section, using the restricted class of the similarities introduced in the preceding section, we consider an algorithm for ALPS. By virtue of a decidability of the similarities, we get such an algorithm easily. Here, we show an algorithm using Shapiro's [11, page 33] incremental model inference algorithm. Concepts and notations for the model inference are based on [11], unless stated otherwise.

Using the terminology of the model inference, we restate the problem in the introduction as follows.

Assume
- vocabulary $\langle\Sigma,\Pi\rangle$,
- observational language L_o which is the set of ground atoms in $\mathrm{Atm}(\Sigma,\Pi)$
- hypothesis language $L_h \subseteq \mathrm{Cls}(\Sigma,\Pi)$,

Given
- a source program P and
- an oracle for unknown model M over $\langle\Sigma,\Pi\rangle$,

Find a finite L_o-complete axiomatization of M which is similar to P.

L_o-complete axiomatization of M ([11, page 8]) means a subset of L_h which is true in M and deduces all positive examples in L_o.

We get the following result of this problem.

Theorem 15. *If M is a h-easy model and there exists a solution in L_h then the Algorithm 1 identifies a solution in the limit.*

The algorithm is illustrated as follows: while switching abstraction mappings, the algorithm searches the hypothesis space constrained by the similarity w.r.t the abstraction mapping with the top-down (i.e. from general to specific) strategy.

The algorithm assumes the followings in the same way as [11].

1. The refinement operator ρ is complete for L_h and conservative for the resolution "$\vdash$".
2. $\alpha_1, \alpha_2, \alpha_3, \ldots$ is a enumeration of all elements of L_o and $\langle \alpha_1, V_1 \rangle, \langle \alpha_2, V_2 \rangle, \ldots$ is a enumeration of facts of M, where a fact $\langle \alpha, V \rangle$ means that $\alpha \in L_o$ has a validity V in M which is taught by the oracle for M.
3. h is a total recursive function.

Moreover, the algorithm assumes the followings.

1. $\phi_1, \ldots, \phi_{len}$ is a enumeration of all elements of $\Phi(\langle \Sigma, \Pi \rangle, \mathrm{V}(P))$ except for elements ϕ such that $\mathcal{D}(\phi^{\Pi}) = \emptyset$, where $len = |\Phi(\langle \Sigma, \Pi \rangle, \mathrm{V}(P))|$.
2. A set of clauses $L_\rho^m(\phi_i)$ denotes $\{C \in L_\rho^m \mid C \notin \mathcal{D}(\phi_i)$ or $P \geq \{\phi_i(C)\}\}$, where L_ρ^m denotes the source set of the marking m ([11, page 32]), i.e. the set of clauses which are not marked '*false*' and all antecedent clauses by ρ are marked '*false*' in m. A set of clauses $L_\rho^m(\phi_i) \mid_k$ denotes $\{C \in L_\rho^m(\phi_i) \mid \mathrm{size}(C) \leq k\}$, where $\mathrm{size}(C)$ denotes the size of the clause C ([11, page 23]). Note that $L_\rho^m(\phi_i) \mid_{k(i)}$ is computable.

The differences between the algorithm and Shapiro's algorithm are as follows. Firstly, our algorithm must find not only an axiomatization but also an abstraction mapping. So, the algorithm has two-dimensional search space: refinements and abstraction mappings. The algorithm can search this search space exhaustively. Secondly, our algorithm use $L_\rho^m(\phi_i)$ instead of L_ρ^m because a solution must be similar to the source program. By the constraint of the similarity, the hypothesis space L_h is reduced to $\bigcup_\phi L_h^\phi$, where $L_h^\phi = \{C \in L_h \mid C \notin \mathcal{D}(\phi)$ or $P \geq \{\phi(C)\}\}$. Of course, this does not mean that our algorithm converges on a solution faster than Shapiro's.

Algorithm 1

For all i $(1 \leq i \leq len)$, $S_F(i) \leftarrow \emptyset$, $S_T(i) \leftarrow \emptyset$, $n(i) \leftarrow 0$, *and* $k(i) \leftarrow 0$
$i \leftarrow 1$ *and mark* $\bot$ *'false'.*
repeat
  if *there exists an* $\alpha \in S_F(i)$ *such that* $L_\rho^m(\phi_i) \mid_{k(i)} \vdash_{n(i)} \alpha$ **then**
    apply the contradiction backtracing algorithm
      and mark the refuted hypothesis 'false'.
    $i \leftarrow i + 1$
  else if *there exists an* $\alpha_j \in S_T(i)$ *such that* $L_\rho^m(\phi_i) \mid_{k(i)} \forall_{h(j)} \alpha_j$ **then**
    $k(i) \leftarrow k(i) + 1$, $i \leftarrow i + 1$
  else
    output $L_\rho^m(\phi_i) \mid_{k(i)}$
    $n(i) \leftarrow n(i) + 1$ *and read a fact* $\langle \alpha_{n(i)}, V_{n(i)} \rangle$
    if $V_{n(i)} =$ true **then** $S_T(i) \leftarrow S_T(i) \cup \{\alpha_{n(i)}\}$
    else $S_F(i) \leftarrow S_F(i) \cup \{\alpha_{n(i)}\}$
  endif
  if $i > len$ **then** $i = 1$
forever

6 Concluding Remarks

In this paper, we firstly have proposed a class of similarities between programs and a theory of ALPS using the similarities. We believe that further research based on the theory reveals that ALPS is effective for improvement of LPS system's performance and useful as a device to shift a bias. Secondly, we have showed that an algorithm for ALPS which is obtained by extending Shapiro's incremental model inference algorithm, and the algorithm identifies a solution in the limit.

However, the algorithm is impractical. This mainly stems from the enumeration of abstraction mappings. To overcome the difficulty, we are now developing an algorithm which constructs an abstraction mapping step by step. In addition, this algorithm incorporates the idea from theorem-proving with abstraction.

Furthermore, we are now investigating to utilize the aspect of ALPS as a device to shift a bias for the change of representation (the program transformation). For instance, given *natural-number* as a source program, an ALPS system transforms *naive-sort* into *insert-sort* which is similar to *natural-number*. That is, from the examples provided by *naive-sort*, the ALPS system synthesizes *insert-sort*. We expect that this method enable us to refine our initial program into more efficient or more comprehensible one depending on a source program.

A Appendix

Proof sketch of Theorem 5. We can show $T_P \uparrow n \supseteq \phi(T_Q \uparrow n)$ by induction on n $(0 \leq n < \omega)$. Since $\bigcup_{n<\omega} T_P \uparrow n \supseteq \bigcup_{n<\omega} \phi(T_Q \uparrow n) = \phi(\bigcup_{n<\omega} T_Q \uparrow n)$, $M(P) \supseteq \phi(M(Q))$ is proved. □

Proof sketch of Theorem 9. We can verify the conditions in Definition 8. For instance, the condition 2 is verified as follows. Let $N_k = \langle \phi(G'_i),\ D_k \rangle$ and $N_{k+1} = \langle \phi(G'_j),\ D_{k+1} \rangle$ be ground refutation nodes in GR', where $1 \leq i < j \leq n$. Then, $G'_j = G'_i \setminus \phi^{-1}(\{\mathrm{He}(D_k)\}) \cup \{A \in \mathrm{Bo}(C_i) \mid \phi(A) \in \mathrm{Bo}(D_k)\}$.

$$\begin{aligned}\phi(G'_j) &= \phi(G'_i \setminus \phi^{-1}(\{\mathrm{He}(D_k)\})) \cup \phi(\{A \in \mathrm{Bo}(C_i) \mid \phi(A) \in \mathrm{Bo}(D_k)\}) \\ &\quad \text{(By Proposition 3.)} \\ &= \phi(G'_i \setminus \phi^{-1}(\{\mathrm{He}(D_k)\})) \cup \mathrm{Bo}(D_k) = \phi(G'_i) \setminus \{\mathrm{He}(D_k)\} \cup \mathrm{Bo}(D_k) \\ &\quad \text{(By } \mathrm{Bo}(D_k) \subseteq \phi(\mathrm{Bo}(C_i)).)\end{aligned}$$

□

Proof sketch of Theorem 14. (if part) For any $I \subseteq B(Q)$ and any $A \in \phi(T_Q(I))$, there exists a clause $C \in Q$ and a grounding substitution $\theta \in \mathrm{Sub}(Q)$ of C such that $\phi(\mathrm{He}(C)\theta) = A$ and $\mathrm{Bo}(C)\theta \subseteq I$. By the assumption, there exists a clause $D \in P$ and a substitution $\sigma \in \mathrm{Sub}(P)$ such that $D\sigma \subseteq \phi(C)$. Thus, by Proposition 3 and Lemma 13, $\mathrm{He}(D)\sigma\phi(\theta) = \phi(\mathrm{He}(C)\theta) = A$ and $\mathrm{Bo}(D)\sigma\phi(\theta) \subseteq \phi(\mathrm{Bo}(C)\theta) \subseteq \phi(I)$. This means $A \in T_P(\phi(I))$.

(only-if part) We prove it by the contraposition. Assume there exists a clause $C \in Q$ such that $D \not\geq \phi(C)$ for any clause $D \in P$. Then, there exists a special and grounding substitution $\theta \in \mathrm{Sub}(P)$ of $\phi(C)$ such that for any clause D in P and any grounding substitution $\sigma \in \mathrm{Sub}(P)$ of D, $D\sigma \not\subseteq \phi(C)\theta$. Therefore, $\phi(\mathrm{He}(C))\theta \notin \mathrm{T}_P(\phi(\mathrm{Bo}(C))\theta)$. On the other hand, there exists a ground substitution $\mu \in \mathrm{Sub}(Q)$ such that $\theta\mu$ is a grounding substitution of C. If we assume $I = \mathrm{Bo}(C)\theta\mu$ then $\phi(\mathrm{He}(C)\theta\mu) = \phi(\mathrm{He}(C))\theta \in \phi(\mathrm{T}_Q(I))$. This means $\mathrm{T}_P(\phi(I)) \not\supseteq \phi(\mathrm{T}_Q(I))$ because $\phi(I) = \phi(\mathrm{Bo}(C))\theta$. □

Proof sketch of Theorem 15. We can show that if $L_\rho^m(\phi_i) \mid_{k(i)}$ is not a solution then the algorithm transfers the index i to next value. Let us fix ϕ_i, Since $\left\langle L_o, L_h^{\phi_i} \right\rangle$ is admissible pair and ρ is complete for $L_h^{\phi_i}$, we can show that the algorithm identifies a solution if the solution is in $L_h^{\phi_i}$ by Theorem 6.3 in [11]. □

References

1. Fausto Giunchiglia and Toby Walsh. A theory of abstraction. *Artificial Intelligence*, 57:323–389, 1992.
2. Bipin Indurkhya. On the role of interpretive analogy in learning. In *Proceeding of ALT'90*, pages 174–189, 1990.
3. J.W. Lloyd. *Foundations of Logic Programming.* Springer Verlag, second edition, 1987.
4. Ryszard S. Michalski. A theory and methodology of inductive learning. *Artificial Intelligence*, 20:111–161, 1983.
5. Stephen Muggleton. Inductive logic programming. In *Inductive Logic Programming.* ACADEMIC PRESS, 1992.
6. David A. Plaisted. Theorem proving with abstraction. *Artificial Intelligence*, 16:47–108, 1981.
7. Luc De Raedt and Maurice Bruynooghe. Constructive induction by analogy. In *Proceeding of ML'89*, pages 476–477, 1989.
8. Luc De Raedt and Maurice Bruynooghe. Interactive concept-learning and constructive induction by analogy. *Machine Learning*, 8:107–150, 1992.
9. Céline Rouveirol. Extension of inversion of resolution applied to theory completion. In *Inductive Logic Programming.* ACADEMIC PRESS, 1992.
10. Seiichiro Sakurai and Makoto Haraguchi. Towards learning by abstraction. In *Proceeding of ALT'91*, pages 288–298, 1991.
11. Ehud Y. Shapiro. Inductive inference of theories from facts. Technical Report 192, Yale University Computer Science Dept., 1981.
12. Leon Sterling and Ehud Shapiro. *The Art of Prolog.* The MIT Press, 1986.
13. Birgit Tausend and Siegfied Bell. Analogical reasoning for logic programming. In *Inductive Logic Programming.* ACADEMIC PRESS, 1992.
14. Josh D. Tenenberg. Abstracting first-order theories. In *Change of Representation and Inductive Bias*, pages 67–79. Kluwer Academic Publishers, 1990.