

Higher Order Generalization

Jianguo Lu ^{*}; Masateru Harao [†]; Masami Hagiya [‡]

Abstract

Generalization is a fundamental operation of inductive inference. While first order syntactic generalization (anti-unification) is well understood, its various extensions are needed in applications. This paper discusses syntactic higher order generalization in a higher order language $\lambda 2[1]$. Based on the *application ordering*, we proved the least general generalization exists and is unique up to *renaming*. An algorithm to compute the least general generalization is presented.

Keywords: higher order logic, unification, anti-unification, generalization.

1 Introduction

The meaning of the word generalization is so general that we can find its occurrences in almost every area of study. In computer science, especially in the area of artificial intelligence, generalization serves as a foundation of inductive inference, and finds its applications in diverse areas such as inductive logic programming [9], theorem proving [10], program derivation [4][5]. In the strict sense, generalization is a dual problem of first order unification and is often called (ordinary) anti-unification. More specifically, it can be formulated as: given two terms t and s , find a term r and substitutions θ_1 and θ_2 , such that $r\theta_1 = t$ and $r\theta_2 = s$. Ordinary anti-unification was well understood as early as in 1970 [11]. Due to the fact that it is inadequate in many problems, there are extensions of ordinary anti-unification from various aspects.

One direction of extending the anti-unification problem is to take into consideration of some kinds of background information as in [9]. Another direction of extension is to promote the order of the underlying language. The problem with higher order generalization is that without some restrictions, the generalization is not well-defined. For

^{*}Address: Robotics Institute, School of Computer Science, Carnegie Mellon University, Pittsburgh PA 15213, USA. Email: jglu@cs.cmu.edu.

[†]Address: Department of Artificial Intelligence, Kyushu Institute of Technology, Iizuka 820, Fukuoka, Japan. Email: harao@dumbo.ai.kyutech.ac.jp.

[‡]Address: Department of Information Science, Graduate School of Science, University of Tokyo, 7-3-1 Hongo, Bunkyo-ku, Tokyo 113, JAPAN. Email: hagiya@is.s.u-tokyo.ac.jp.

example, the common generalizations of Aa and Bb without restriction would be: $fx, fa, fb, fab, fA, fB, \dots, f(Aa, Bb), f(g(A, B), g(a, b)), \dots$, where f and g are variables. Actually, there are infinite number of generalizations. Obviously, some restrictions must be imposed on higher order generalization.

This paper is devoted to the study of higher order generalization. More specifically, we study the conditions under which the least higher order generalization exist and unique. The most closely related works are [10] [3].

[10] studied generalization in a restricted form of calculus of constructions [2], where terms are higher-order patterns, i.e., free variables can only apply to distinct bound variables. One problem of the generalization in higher-order patterns is the over generalization. For example, the least generalization of Aa and Ba would be a single variable x instead of fa or fx , where we suppose A, B, a are constants, and f, x are variables. Another problem of higher-order pattern is that it is inadequate to express some problems. In particular, it can not represent recursion in its terms.

This motivated the study of generalization in $M\lambda$ [3]. In $M\lambda$, free variables can apply to object term, which can contain constants and free variables in addition to bound variables. In this sense, $M\lambda$ extends $L\lambda$. On the other hand, it also added some restrictions. One restriction is that $M\lambda$ is situated in a simply typed λ calculus instead of calculus of constructions. Another restriction is $M\lambda$ does not have type variables, hence it can only generalize two terms of the same type. The result is not satisfactory in that the least general generalization is unique up to *substitution*. That means any two terms beginning with functional variables are considered equal.

Unlike the other approaches, which mainly put restrictions on the situated language, we mainly restrict the notion of the ordering between terms. Our discussion is situated in a restricted form of the language λ_2 [1]. The reason to choose λ_2 is that it is a simple calculus which allows type variables. It can be used to formalise various concepts in programming languages, such as type definition, abstract data types, and polymorphism. The restriction we added is that abstractions should not occur inside arguments. In the restricted language λ_2 , we propose the following:

- an ordering between terms, called *application ordering* (denoted as \succeq), which is similar to, but not the same as the substitution (instantiation) ordering [11][10].
- A kind of restriction on orderings, called *subterm restriction* (the corresponding ordering is denoted as \succeq_S), which is implicit in first order languages, but usually not assumed in higher order languages.
- An extension to the ordering, called *variable freezing* (the corresponding ordering is denoted as \succeq_{SF}), which makes the ordering more useful while keeping the matching and generalization problems decidable.
- A generalization method based on the afore-mentioned ordering.

Based on the \succeq_{SF} ordering, we have the following results similar to the first order anti-unification:

- For any two terms t and s , $t \succeq_{SF} s$ is decidable.
- The least general generalization exists.
- The least general generalization is unique up to *renaming*.

2 Preliminaries

The syntax of the restricted $\lambda 2$ can be defined as follows[1]:

Definition 1 (types and terms) *The set of types is defined as:*

$$\begin{aligned} V &= \{\alpha, \alpha_1, \alpha_2, \dots\}, \text{ (type variables),} \\ C &= \{\gamma, \gamma_1, \gamma_2, \dots\}, \text{ (type constants),} \\ T &= V|C|T \rightarrow T|[V]T, \text{ (types).} \end{aligned}$$

The set of terms is defined as:

$$\begin{aligned} X &= \{x, x_1, x_2, \dots\}, \text{ (variables),} \\ A &= \{a, a_1, a_2, \dots\}, \text{ (constants),} \\ \Lambda_1 &= X|A|\Lambda_1\Lambda_1|\Lambda T, \text{ (terms without abstraction),} \\ \Lambda &= \Lambda_1|[X : T]\Lambda|[V]\Lambda, \text{ (terms).} \end{aligned}$$

Here for the purpose of convenience, we use $[x : \sigma]$ instead of $\lambda x : \sigma$. Also, we use the same notation $[V]$ to denote ΛV (and $\forall V$), since we can distinguish among λ, Λ and \forall from the context.

The assignment rules of $\lambda 2$ are listed here for ease of reference:

Definition 2 *Let σ, γ are types. $\Gamma \vdash t : \sigma$ is defined by the following axiom and rules:*

(start) $\Gamma \vdash x : \sigma$, if $(x : \sigma) \in \Gamma$;

$$(\rightarrow E) \frac{\Gamma \vdash t : (\sigma \rightarrow \tau), \Gamma \vdash s : \sigma}{\Gamma \vdash ts : \tau}$$

$$(\rightarrow I) \frac{\Gamma, x : \sigma \vdash t : \tau}{\Gamma \vdash [x : \sigma]t : (\sigma \rightarrow \tau)}$$

$$(\forall E) \frac{\Gamma \vdash t : [\alpha]\sigma}{\Gamma \vdash t\tau : \sigma[\alpha := \tau]}$$

$$(\forall I) \frac{\Gamma \vdash t : \sigma}{\Gamma \vdash [\alpha]t : [\alpha]\sigma}, \text{ if } \alpha \notin FV(\Gamma).$$

We call a term t is valid (under Γ) if there is a type σ such that $\Gamma \vdash t : \sigma$. We use $Typ(t)$ to denote the type of t . *Atoms* are either constants or variables. By *closed terms* we mean the terms that do not contain occurrences of free variables. In the following discussion, if not specified otherwise, we assume all terms are closed, and in long $\beta\eta$ normal form. Given $\Delta \equiv [x_1 : \sigma_1][x_2 : \sigma_2] \dots [x_n : \sigma_n]$ and term t , $[\Delta]t$ denotes $[x_1 : \sigma_1][x_2 : \sigma_2] \dots [x_n : \sigma_n]t$. When type information is not important, $[x : \sigma]t$ is abbreviated as $[x]t$.

Following [10], we have a similar notion of *renaming*. Given natural numbers n and p , a partial permutation ϕ from n into p is an injective mapping from $\{1, 2, \dots, n\}$ into $\{1, 2, \dots, p\}$. A *renaming* of a term $[x_1 : \sigma_1][x_2 : \sigma_2] \dots [x_p : \sigma_p]t$ is a valid and closed term $[x_{\phi(1)} : \sigma_{\phi(1)}][x_{\phi(2)} : \sigma_{\phi(2)}] \dots [x_{\phi(n)} : \sigma_{\phi(n)}]t$. Intuitively, *renaming* is to permute and to drop some of the abstractions when allowed. For example, $[x_3, x_1 : \gamma]Ax_1x_3$ is a renaming of $[x_1, x_2, x_3 : \gamma]Ax_1x_3$.

3 Application orderings

3.1 Application ordering (\succeq)

Definition 3 (\succeq) *Given two terms t and s . t is more general than s (denoted as $t \succeq s$) if there exists a sequence of terms and types r_1, r_2, \dots, r_k , such that $tr_1r_2 \dots r_k$ is valid, and $tr_1r_2 \dots r_k = s$. Here k is a natural number.*

To distinguish \succeq with the usual instantiation ordering (denote it as \geq), we call \succeq the *application ordering*. Compared with the instantiation ordering, the application ordering does not lose generality in the sense that for every two terms t and s in $\lambda 2$, if $t \geq s$, and t_1 and s_1 are the closed form of t and s , then $t_1 \succeq_F s_1$, where \succeq_F is defined in section 3.3.

Example 1 *The following are some examples of the application ordering.*

$$\begin{aligned} & [\alpha][f : \alpha \rightarrow \alpha \rightarrow \alpha][x, y : \alpha]fxy \\ \succeq & [f : \gamma \rightarrow \gamma \rightarrow \gamma][x, y : \gamma]fxy \\ \succeq & [x, y : \gamma]Axy \\ \succeq & [y : \gamma]Aay \\ \succeq & Aab. \end{aligned}$$

\succeq is reflexive and transitive:

Proposition 1 *For any terms t, t_1, t_2, t_3 ,*

1. $t \succeq t$;
2. *If $t_1 \succeq t_2, t_2 \succeq t_3$, then $t_1 \succeq t_3$.*

Proof:

1. Trivial.
2. Since $t_1 \succeq t_2, t_2 \succeq t_3$, there are a sequence of terms or types $r_{11}, r_{12}, \dots, r_{1n}, r_{21}, r_{22}, \dots, r_{2m}$, such that $t_1r_{11}r_{12} \dots r_{1n} = t_2$, and $t_2r_{21}r_{22} \dots r_{2m} = t_3$, hence $t_1r_{11}r_{12} \dots r_{1n}r_{21}r_{22} \dots r_{2m} = t_3$.

□

3.2 Application ordering with subterm restriction (\succeq_S)

Because \succeq is too general to be of practical use, we restrict the relation to \succeq_S , called subterm restriction. First of all, we define the notion of subterms.

Definition 4 (subterm) *The set of subterms of term t (denoted as $\text{subterm}(t)$) is defined as $\text{decm}(\text{norm}(t)) \cup \{\text{Typ}(r') \mid r' \in \text{decm}(\text{norm}(t))\}$.*

Here $\text{norm}(t)$ is to get the $\beta\eta$ normal form for the term t . $\text{decm}(r)$ is to decompose terms recursively into a set of its components, which is defined as:

1. $\text{decm}(c) = \{c\}$ (constants remain the same);
2. $\text{decm}(z) = \{\}$; (variables are filtered out);
3. $\text{decm}(ts) = \text{decm}(t) \cup \text{decm}(s) \cup \{ts\}$, if there is no variable in ts ;
 $= \text{decm}(t) \cup \text{decm}(s)$, otherwise;
4. $\text{decm}([d]t) = \text{decm}(t)$.

Example 2 *Assume $A : \gamma \rightarrow \gamma \rightarrow \gamma, B : \gamma \rightarrow \gamma$,*
 $\text{subterm}([x : \gamma]Axa) = \{[x, y : \gamma]Axy, a, \gamma, \gamma \rightarrow \gamma \rightarrow \gamma\}$
 $\text{subterm}([f : \gamma \rightarrow \gamma][x : \gamma]f(Bx)) = \{[x : \gamma]Bx, \gamma \rightarrow \gamma\}$.

As we can see, the subterms do not contain free variables. Actually, there is no bound variables except the term having its η normal form (the $[x, y : \gamma]Axy$ in the above example). Here we exclude the *identity* and *projection* functions as subterms. This is essential to guarantee there exists least generalization in the application ordering. The intuitive behind this is that when we match two higher order terms, in general there are *imitation* rule and *projection* rule [6]. Here only *imitation* rule is used. We regard it is *projection* rule that brings about the unpleasant results and the complexities in higher order generalizations.

Definition 5 (\succeq_S) *Given two terms t and s . t is more general than s by subterms (denoted as $t \succeq_S s$), if there exists a sequence of r_1, r_2, \dots, r_k , such that $tr_1r_2\dots r_k = s$. Here $r_i \in \text{subterm}(s), i \in \{1, 2, \dots, k\}$, and k is a natural number.*

Example 3 $[f][x]fx \succeq_S Aa;$
 $[f][x]fx \succeq_S Bbc;$
 $[\alpha][x : \alpha]x \succeq_S [x : \gamma]x \succeq_S Aa;$
 $[f][x]fx \not\succeq_S a$, since the only subterm of a is a .

Due to the finiteness of $\text{subset}(s)$, the ordering \succeq_S becomes much easier to manage than \succeq .

Proposition 2 *For any terms t_1, t_2, t_3 ,*

1. There exists a procedure to decide if $t_1 \succeq_S t_2$.
2. If $t_1 \succeq_S t_2$, $t_2 \succeq_S t_3$, then $t_1 \succeq_S t_3$.

Proof:

1. Since the *subterm* of t_1 is finite, it is obvious.
2. Since $t_1 \succeq_S t_2$, $t_2 \succeq_S t_3$, there are a sequence of terms or types $r_{11}, r_{12}, \dots, r_{1n} \in \text{subterm}(t_2)$, $r_{21}, r_{22}, \dots, r_{2m} \in \text{subterm}(t_3)$, such that $t_1 r_{11} r_{12} \dots r_{1n} = t_2$, and $t_2 r_{21} r_{22} \dots r_{2m} = t_3$. Hence $t_1 r_{11} r_{12} \dots r_{1n} r_{21} r_{22} \dots r_{2m} = t_3$. Besides, since we can not eliminate constants in t_2 when applying terms to it, and $r_{11}, r_{12}, \dots, r_{1n}$ are constants in t_2 , so $r_{11}, r_{12}, \dots, r_{1n}$ must also be the subterms of t_3 . Hence we have $t_1 \succeq_S t_3$.

□

3.3 Application ordering with subterm restriction and variable freezing extension (\succeq_{SF})

The ordering \succeq_S is restrictive in that $[x][y]Axy \not\succeq_S [x]Axa$. To solve this problem, we have:

Definition 6 (\succeq_F) *t is a generalization of s by variable freezing, denoted as $t \succeq_F s$, if either*

- $t \succeq s$, or
- for an arbitrary type constant or term constant c such that sc is valid, $t \succeq_F sc$.

Intuitively, here we first freeze some variables in s as a constant, then try to do generalization. The word *freeze* comes from [7], which has the notion that when unifying two free variables, we can regard one of them as a constant.

The ordering \succeq_F is too general to be managed, so we have the following restricted form:

Definition 7 (\succeq_{SF}) *t \succeq_{SF} s, if either*

- $t \succeq_S s$, or
- For an arbitrary type constant or term constant c such that sc is valid, $t \succeq_{SF} sc$.

Now we have $[x][y]Axy \succeq_{SF} [x]Axa$. The notion of \succeq_{SF} not only mimics, but also extends the usual meaning of instantiation ordering. For example, we have $[x, y]Axy \succeq_{SF} [x]Axx$, which can not be obtained in the instantiation ordering.

Example 4 *The following relations holds:*

$$\begin{aligned} [\alpha][x : \alpha]x &\succeq_{SF} [\alpha][f : \alpha \rightarrow \alpha][x : \alpha]fx \\ &\succeq_S [f : \gamma \rightarrow \gamma][x : \gamma]fx \\ &\succeq_S [x : \gamma]Ax \\ &\succeq_S Aa; \end{aligned}$$

$$[f][z, x, y]f(Axy, z) \succeq_S [z, x, y]A(Axy, z) \succeq_S A(Aab, Aab);$$

$$[f][z, x, y]f(Axy, z) \succeq_{FS} [x, y]A(Axy, Axy); \text{ since } Axy \text{ is not a subterm of } [x, y]A(Axy, Axy).$$

$$[\alpha][f : \alpha \rightarrow \alpha][x : \alpha]fx \not\succeq_{SF} [\alpha][x : \alpha]x, \text{ since identity and projection functions are}$$

not subterms.

Proposition 3 *For any terms t and s ,*

1. $t \succeq_{SF} s$ iff there exists a sequence (possibly an empty sequence) of new, distinct constants c_1, c_2, \dots, c_k , such that $sc_1c_2\dots c_k$ is of atomic type, and $t \succeq_S sc_1c_2\dots c_k$.
2. There exists a procedure to decide if $t \succeq_{SF} s$.
3. Suppose $t = s$. If $t \succeq_{SF} r$, then $s \succeq_{SF} r$. If $r \succeq_{SF} t$, then $r \succeq_{SF} s$.

Proof:

1. (\Rightarrow) Suppose $t \succeq_{SF} s$. If s is of atomic type, then it is trivial. Now suppose s is of type $\sigma \rightarrow \tau$, c is a constant of type σ . If $t \succeq_S s$, then $t \succeq_S sc$. If $t \not\succeq_S s$. By definition of \succeq_{SF} , there exists c such that $t \succeq_{SF} sc$.
 (\Leftarrow) Suppose there exists a sequence of new constants c_1, c_2, \dots, c_k , such that $sc_1c_2\dots c_k$ is of atomic type, and $t \succeq_S sc_1c_2\dots c_k$. By definition of \succeq_{SF} , $t \succeq_{SF} sc_1c_2\dots c_{k-1}$, $t \succeq_{SF} sc_1c_2\dots c_{k-2}$, ..., $t \succeq_{SF} s$.
2. Since $t \succeq_{SF} s$ iff $t \succeq_S sc_1c_2\dots c_k$, and we know \succeq_S is decidable, hence $t \succeq_{SF} s$ is decidable.
3. If $t \succeq_{SF} r$, then there exists a sequence of new constants c_1, c_2, \dots, c_k , such that $rc_1c_2\dots c_k$ is of atomic type, and $t \succeq_S rc_1c_2\dots c_k$. There exists a sequence of terms or types r_1, \dots, r_i , such that $tr_1\dots r_i = rc_1c_2\dots c_k$. Since $t = s$, we have $sr_1\dots r_i = rc_1c_2\dots c_k$, $s \succeq_{SF} r$.

The second proposition can be proved in the similar way.

□

Proposition 4 *Suppose $t_1 \equiv [\Delta]hs_1s_2\dots s_m$, $t_2 \equiv [\Delta']h's'_1s'_2\dots s'_n$, and $t_1 \succeq_{SF} t_2$, then*

1. $m \leq n$,
2. $[\Delta]s_k \succeq_{SF} [\Delta']s'_{k+n-m}$, for $k \in \{1, 2, \dots, m\}$,
3. If h is a constant, then h' must be a constant, and $h = h', m = n$.

Proof: Suppose $[\Delta'] = [z_1, z_2, \dots, z_j]$. Since $t_1 \succeq_{SF} t_2$, we have

$[\Delta]h s_1 s_2 \dots s_m \succeq_S (h' s'_1 s'_2 \dots s'_m)[\bar{c}/\bar{z}]$, where \bar{z} is a sequence z_1, \dots, z_j , \bar{c} is a sequence of new constant symbols c_1, \dots, c_j . Now, suppose each variable in $h' s'_1 s'_2 \dots s'_m$ is fixed as a new constants, $h s_1 s_2 \dots s_m$ should match $h' s'_1 s'_2 \dots s'_m$ in the sense of [Huet78]. As we know, the complete minimal matches are generated by the imitation rule and the projection rule. Since in the projection rule, the substitutions are $\{h \rightarrow [x_1, \dots, x_m]x_i | i \in \{1, \dots, m\}\}$, which do not satisfy our subterm restriction. Thus the only way to match is by using the imitation rule. By imitation rule we have substitutions

$\{h \rightarrow [x_1, \dots, x_m]h'(h_1 x_1 \dots x_m) \dots (h_n x_1 \dots x_m)\}$, where h_1, \dots, h_n are new variables. On the other hand, the subterms of $h' s'_1 s'_2 \dots s'_n$ whose head is h' could only be:

$[x_1, \dots, x_n]h'x_1 x_2 \dots x_n,$
 $[x_2, \dots, x_n]h's'_1 x_2 \dots x_n,$
 $\dots \dots$
 $[x_{i+1}, \dots, x_n]h's''_1 s''_2 \dots s''_i x_{i+1} \dots x_n,$
 $\dots \dots,$

where each s''_j is either s'_j , or other possible terms inside the arguments if h' also occurs in the arguments. So the possible substitution must be $h \rightarrow [x_{i+1}, \dots, x_n]h's''_1 s''_2 \dots s''_i x_{i+1} \dots x_n$, where $i + m = n$. After the substitution, we have to match the terms $h's''_1 \dots s''_{n-m} s_1 s_2 \dots s_m$ and $h's'_1 s'_2 \dots s'_n$, i.e., $[\Delta]s_k \succeq_{SF} [\Delta']s'_{k+n-m}$, for $k \in \{1, 2, \dots, m\}$.

When h is a constant, it is obvious that $h' = h$.

□

It is clear that \succeq_{SF} is reflexive and transitive:

Proposition 5 For any terms t, t_1, t_2, t_3 ,

1. $t \succeq_{SF} t$.
2. If $t_1 \succeq_{SF} t_2$, $t_2 \succeq_{SF} t_3$, then $t_1 \succeq_{SF} t_3$.

Proof:

1. Obvious.

2. We can suppose

$t_1 \equiv [\Delta]h s_1 s_2 \dots s_m,$
 $t_2 \equiv [\Delta']h'r_{11} \dots r_{1i} s'_1 s'_2 \dots s'_m,$
 $t_3 \equiv [\Delta'']h''r_{21} \dots r_{2j} r_{31} \dots r_{3i} s''_1 s''_2 \dots s''_m,$

Case 1: $m = 0$, then it is easy to verify $t_1 \succeq_{SF} t_3$.

Case 2: $m > 0$. We have $[\Delta]s_k \succeq_{SF} [\Delta']s'_k \succeq_{SF} [\Delta'']s''_k$, for $k \in \{1, \dots, m\}$. By inductive hypothesis, $[\Delta]s_k \succeq_{SF} [\Delta'']s''_k$. If h is a constant, we have $h = h' = h''$, $i = j = 0$, thus $t_1 \succeq_{SF} t_3$. If h is a variable, let h substitute $h''r_{21} \dots r_{2j} r_{31} \dots r_{3i}$.

¹Here the variables are frozen.

□

Definition 8 (\cong) $t \cong s$ is defined as $t \succeq_{SF} s$ and $s \succeq_{SF} t$.

Example 5 $[x, y]Axy \cong [y, x]Axy \cong [z, x, y]Axy$.

Proposition 6 $t \cong s$ iff t is a renaming of s .

Proof: (\Rightarrow) Assume $t \cong s$, then $t \succeq_{SF} s$ and $s \succeq_{SF} t$. Suppose

$t \equiv [\Delta]ht_1t_2\dots t_m$, $s \equiv [\Delta']h's_1s_2\dots s_n$. Since $t \succeq_{SF} s$, we have $m \geq n$. And similarly, we have $n \geq m$. So, $m = n$. If h is a constant, then $h' = h$. Similarly, we can have if h' is a constant then $h' = h$. Hence, h and h' must be either the same constant, or a variable.

Case 1. $m = 0$. Obviously t and s only differs by renaming.

Case 2. $m > 0$. We have $[\Delta]t_k \succeq_{SF} [\Delta']s_k$, and $[\Delta']s_k \succeq_{SF} [\Delta]t_k$, for $k \in \{1, \dots, m\}$. By inductive hypothesis, $[\Delta]t_k$ and $[\Delta']s_k$ only differs by variable renaming. On the other hand, h and h' are either variables or the same constant.

(\Leftarrow) We only need to consider the following two cases:

Case 1. Suppose

$$t \equiv [x_1, x_2, \dots, x_i]ht_1t_2\dots t_m,$$

$$s \equiv [x_{\phi(1)}, x_{\phi(2)}, \dots, x_{\phi(i)}]ht_1t_2\dots t_m,$$

$$\text{Then } tc_1\dots c_i = sc_{\phi(1)}\dots c_{\phi(i)}, t \cong s.$$

Case 2. Suppose

$$t \equiv [x][x_1, x_2, \dots, x_i]ht_1t_2\dots t_m,$$

$$s \equiv [x_1, x_2, \dots, x_i]ht_1t_2\dots t_m,$$

where x does not occur in $ht_1t_2\dots t_m$. Then $tc = s$, $s \succeq_{SF} t$. Also, we have $t \succeq_{SF} s$, hence $t \cong s$.

Case 3. Suppose

$$t \equiv [g : \gamma_1 \rightarrow \gamma_2 \rightarrow \dots \rightarrow \gamma_i \rightarrow \gamma]gt_1t_2\dots t_m,$$

$$s \equiv [f : \gamma_{\phi(1)} \rightarrow \gamma_{\phi(2)} \rightarrow \dots \rightarrow \gamma_{\phi(i)} \rightarrow \gamma]ft_{\phi(1)}t_{\phi(2)}\dots t_{\phi(m)},$$

$$tA = s([x_{\phi(1)} : \gamma_{\phi(1)}, x_{\phi(2)} : \gamma_{\phi(2)}, \dots, x_{\phi(i)} : \gamma_{\phi(i)}]Ax_1x_2\dots x_i), \text{ hence } t \cong s.$$

□

4 Generalization

If $t \succeq_{SF} s_1$ and $t \succeq_{SF} s_2$, then t is called a *common generalization* of s_1 and s_2 . If t is a common generalization of s_1 and s_2 , and for any common generalization t_1 of s_1 and s_2 , $t_1 \succeq_{SF} t$, then t is called the *least general generalization (LGG)*. This section only concerned with \succeq_{SF} , hence in the following discussion the subscript SF is omitted.

The following algorithm $Gen(t, s, \{\})$ computes the least general generalization of t and s . Recall we assume t and s are closed terms. At the beginning of the procedure we suppose all the bound variables in t and s are distinct. Here an auxiliary (the third) global variable \mathcal{C} is needed to record the previous correspondence between terms in the

course of generalization, so that we can avoid to introduce unnecessary new variables. \mathcal{C} is a bijection between pairs of terms (and types) and a set of variables. Initially, \mathcal{C} is an empty set. Following the usual practice, it is sufficient to consider only long $\beta\eta$ -normal forms. Not losing generality, suppose t and s are of the following forms:

$$\begin{aligned}
t &\equiv [\Delta]h(t_1, t_2, \dots, t_k), \\
s &\equiv [\Delta']h'(r_1, \dots, r_i, s_1, s_2, \dots, s_k), \text{ where } h \text{ and } h' \text{ are atoms. Suppose} \\
[\Delta, \Delta', \Delta_1]t'_1 &= Gen([\Delta]t_1, [\Delta']s_1, \mathcal{C}), \\
[\Delta, \Delta', \Delta_2]t'_2 &= Gen([\Delta, \Delta_1]t_2, [\Delta', \Delta_1]s_2, \mathcal{C}), \\
&\dots \\
[\Delta, \Delta', \Delta_k]t'_k &= Gen([\Delta, \Delta_{k-1}]t_k, [\Delta', \Delta_{k-1}]s_k, \mathcal{C}), \\
Typ(h) &= \sigma_1, \sigma_2, \dots, \sigma_k \rightarrow \sigma_{k+1}, \\
Typ(h'(r_1, \dots, r_i)) &= \tau_1, \tau_2, \dots, \tau_k \rightarrow \tau_{k+1}. \\
Gen(t, s, \mathcal{C}): \\
\text{Case 1: } h = h' : Gen(t, s, \mathcal{C}) &= [\Delta, \Delta', \Delta_k]h(t'_1, t'_2, \dots, t'_k); \\
\text{Case 2: } h \neq h': \\
\text{Case 2.1: } \exists x.((h, h'(r_1, \dots, r_i)), x) \in \mathcal{C}: \\
Gen(t, s, \mathcal{C}) &= [\Delta, \Delta', \Delta_k]x(t'_1, t'_2, \dots, t'_k) \\
\text{Case 2.2: } \neg \exists x.((h, h'(r_1, \dots, r_i)), x) \in \mathcal{C}, \\
\text{Case 2.2.1: } Typ(h) = Typ(h'(r_1, \dots, r_i)): \\
Gen(t, s, \mathcal{C}) &= [\Delta, \Delta', \Delta_k][x : \sigma_1, \sigma_2, \dots, \sigma_k \rightarrow \sigma_{k+1}]x(t'_1, t'_2, \dots, t'_k) \\
\mathcal{C} &:= \{((h, h'(r_1, \dots, r_i)), x)\} \cup \mathcal{C}; \\
\text{Case 2.2.2: } Typ(h) \neq Typ(h'(r_1, \dots, r_i)): \\
\text{Not losing generality, suppose } \sigma_j \neq \tau_j, j \in \{1, 2, \dots, k, k+1\}. \\
\text{Case 2.2.2.1: } \exists \alpha_j.((\sigma_j, \tau_j), \alpha_j) \in \mathcal{C}: \\
Gen(t, s, \mathcal{C}) &= [\Delta, \Delta', \Delta_k][x : \sigma_1, \dots, \alpha_j, \dots \rightarrow \sigma_{k+1}]x(t'_1, t'_2, \dots, t'_k); \\
\mathcal{C} &:= \{((h, h'(r_1, \dots, r_i)), x)\} \cup \mathcal{C}; \\
\text{Case 2.2.2.2: } \neg \exists \alpha.((\sigma_j, \tau_j), \alpha_j) \in \mathcal{C}: \\
Gen(t, s, \mathcal{C}) &= [\Delta, \Delta', \Delta_k][\alpha_j][x : \sigma_1, \dots, \alpha_j, \dots \rightarrow \sigma_{k+1}]x(t'_1, t'_2, \dots, t'_k); \\
\mathcal{C} &:= \{((h, h'(r_1, \dots, r_i)), x)\} \cup \mathcal{C}; \\
\mathcal{C} &:= \{((\sigma_j, \tau_j), \alpha_j)\} \cup \mathcal{C}.
\end{aligned}$$

In the following, let $t \sqcup s \equiv Gen(t, s, \{\})$.

Example 6 *Some examples of least general generalization.*

$$\begin{aligned}
[x : \gamma]x \sqcup Aa &= [x : \gamma][\alpha][y : \alpha]y \cong [\alpha][y : \alpha]y, \text{ if } Aa \text{ is not of type } \gamma; \\
[x : \gamma]x \sqcup Aa &= [x : \gamma][y : \gamma]y \cong [x : \gamma]x, \text{ if } Aa \text{ is of type } \gamma; \\
[x]Axx \sqcup [x]Aax &\cong [x, y]Axy; \\
Aa \sqcup Bb &\cong [f][x]fx, \text{ if } A \text{ and } B \text{ is of the same type}; \\
Aa \sqcup Bb &\cong [\alpha][f : \alpha \rightarrow \gamma][x : \alpha]fx, \text{ if } A : \gamma_1 \rightarrow \gamma \text{ and } B : \gamma_2 \rightarrow \gamma;
\end{aligned}$$

Example 7 *Here is an example of generalizing segments of programs. For clarity the segments are written in usual notation. Let*

$$\begin{aligned}
t &\equiv [x]map1(cons(a, x)) = cons(succ(a), map1(x)), \\
s &\equiv [x]map2(cons(a, x)) = cons(sqr(a), map2(x)).
\end{aligned}$$

Suppose the types are

$$\begin{aligned} \text{map1} &: \text{List}(\text{Nat}) \rightarrow \text{Nat}; \text{succ} : \text{Nat} \rightarrow \text{Nat}, \\ \text{map2} &: \text{List}(\text{Nat}) \rightarrow \text{Nat}; \text{sqr} : \text{Nat} \rightarrow \text{Nat}. \end{aligned}$$

Then

$$t \sqcup s \cong [f : \text{List}(\text{Nat}) \rightarrow \text{Nat}; g : \text{Nat} \rightarrow \text{Nat}][x]f(\text{cons}(a, x)) = \text{cons}(g(a), f(x)).$$

The termination of the algorithm is obvious, since we recursively decompose the terms to be generalized, and the size of the terms strictly decreases in each step. What we need to prove is the uniqueness of the generalization. The following can be proved by induction on the definition of terms:

Proposition 7 1. (consistency) $t \sqcup s \succeq t, t \sqcup s \succeq s$.

2. (termination) For any two term t and s , $\text{Gen}(t, s, \{\})$ terminates.

3. (absorption) If $t \succeq s$, then $t \sqcup s \cong t$.

4. (idempotency) $t \sqcup t \cong t$.

5. (commutativity) $t \sqcup s \cong s \sqcup t$.

6. (associativity) $(t \sqcup s) \sqcup r \cong t \sqcup (s \sqcup r)$.

7. If $t \cong s$, then $t \sqcup r \cong s \sqcup r$.

8. (monotonicity) If $t \succeq s$, then for any term r , $t \sqcup r \succeq s \sqcup r$.

9. If $t \cong s$, then $t \sqcup s \cong t \cong s$.

Proof:

1. It can be verified that for each case of the algorithm, we obtained a more general term.
2. It is obvious since we decompose the terms recursively.
3. Since $t \succeq s$, we can suppose

$$\begin{aligned} t &\equiv [\Delta]hs_1s_2\dots s_m, \\ s &\equiv [\Delta']h'r_{11}\dots r_{1i}s'_1s'_2\dots s'_m, \text{ and} \\ &[\Delta]s_k \succeq [\Delta']s'_k, k \in \{1, \dots, m\}. \end{aligned}$$

If $m = 0$, then it is easy to verify the conclusion. Now suppose $m > 0$. Not losing generality, suppose h is a variable which does not occur in s , and has a single occurrence in t . h has the same type as $h'r_{11}\dots r_{1i}$. Other cases can be proved in a similar way. Now we can suppose $t \sqcup s \equiv [\Delta''][f]ft_1t_2\dots t_m$.

If s_k is a constant, then s'_k must be the same constant. Hence $t_k \equiv s_k$. If s_k is a variable, then t_k is a new variable. There are two cases: one is s_k has only one

occurrence in t . Then t' and t only differs by renaming. The other case is that s_k has multiple occurrences in t . Since $t \succeq s$, all the occurrences of s_k must correspond to a same term in s . Hence due to the presence of the global variable \mathcal{C} , all the occurrences of s_k are generalized as a same variable. Hence $t \cong t'$. By inductive hypothesis, we have

$$[\Delta]_{s_k} \sqcup [\Delta']_{s'_k} \cong [\Delta]_{s_k}.$$

4. From $t \succeq t$ and the proposition 7.3 we can have the result.
5. It is obvious from the algorithm.
6. Not losing generality, we can suppose

$$t \equiv [\Delta]hs_1s_2\dots s_m,$$

$$s \equiv [\Delta']h'r_{11}\dots r_{1i}s'_1s'_2\dots s'_m,$$

$$r \equiv [\Delta'']h''r_{21}\dots r_{2j}r_{31}\dots r_{3i}s''_1s''_2\dots s''_m,$$

and suppose h, h', h'' are distinct constants, t, s, r are of the same type. The other cases can be proved in a similar way. By inductive hypothesis, for $k \in \{1, \dots, m\}, p \in \{1, \dots, i\}$, we can suppose:

$$([\Delta]_{s_k} \sqcup [\Delta']_{s'_k}) \sqcup [\Delta'']_{s''_k} \cong [\Delta]_{s_k} \sqcup ([\Delta']_{s''_k} \sqcup [\Delta'']_{s''_k}),$$

$$[\Delta]_{s_k} \sqcup [\Delta']_{s'_k} \cong [\Gamma]t_k,$$

$$[\Gamma]t_k \sqcup [\Delta'']_{s''_k} \cong [\Gamma'']t''_k,$$

$$[\Delta']_{s'_k} \sqcup [\Delta'']_{s''_k} \cong [\Gamma']t'_k,$$

$$[\Delta]_{s_k} \sqcup [\Gamma']t'_k \cong [\Gamma'']t''_k,$$

$$[\Delta']r_{1p} \sqcup [\Delta'']r_{3p} \cong [\Gamma']r_p.$$

Here we suppose each $\Gamma, \Gamma', \Gamma''$ are large enough to cover all the abstractions in $t_1, \dots, t_m, t'_1, \dots, t'_m$, and t_1, \dots, t''_m , respectively.

We rename the variables in $[\Gamma]t_1, \dots, [\Gamma]t_m$ such that there are multiple occurrences of a variable x in $[\Gamma]t_1, \dots, [\Gamma]t_m$ if and only if its corresponding places in s_1, \dots, s_m hold a same term, and its corresponding places in s'_1, \dots, s'_m hold another same term. Similarly, we rename the terms $[\Gamma']t'_k, [\Gamma'']t''_k$. Then

$$\begin{aligned} & (t \sqcup s) \sqcup r \\ & \cong [\Gamma][f]ft_1\dots t_m \sqcup [\Delta'']h''r_{21}\dots r_{2j}r_{31}\dots r_{3i}s''_1s''_2\dots s''_m \\ & \cong [\Gamma''] [f]ft''_1\dots t''_m, \end{aligned}$$

$$\begin{aligned} & t \sqcup (s \sqcup r) \\ & \cong [\Delta]hs_1s_2\dots s_m \sqcup [\Gamma'][g]gr_1\dots r_it'_1\dots t'_m \\ & \cong [\Gamma''] [g]gt''_1\dots t''_m. \end{aligned}$$

Hence $(t \sqcup s) \sqcup r \cong t \sqcup (s \sqcup r)$.

7. Since $t \cong s$, t is a renaming of s . t and s must be of the forms $[\Delta]hr_1r_2\dots r_n$ and $[\Delta']hr_1r_2\dots r_n$. It is obvious that $[\Delta]hr_1r_2\dots r_n \sqcup r \cong [\Delta']hr_1r_2\dots r_n \sqcup r$.

8. Since $t \succeq s$, we have $t \sqcup s \cong t$, hence

$$\begin{aligned} & t \sqcup r \\ & \cong (t \sqcup s) \sqcup r && \text{(by proposition 7.7)} \\ & \cong t \sqcup (s \sqcup r) && \text{(commutativity)} \\ & \succeq s \sqcup r && \text{(by proposition 7.1).} \end{aligned}$$

9. From $t \cong s$, we have $t \succeq s, s \succeq t$. Hence $t \sqcup s \cong t, t \sqcup s \cong s \sqcup t \cong s$.

□

Based on the above propositions, we can have

Theorem 1 $t \sqcup s$ is the least general generalization of t and s , i.e., for any term r , if $r \succeq t, r \succeq s$, then $r \succeq t \sqcup s$.

Proof: Since $r \succeq t, r \succeq s$, we have $r \sqcup t \cong r, r \sqcup s \cong r$.

$$\begin{aligned} & r \sqcup (t \sqcup s) \\ & \cong (r \sqcup r) \sqcup (t \sqcup s) && \text{(idempotency)} \\ & \cong (r \sqcup t) \sqcup (r \sqcup s) && \text{(commutativity and associativity)} \\ & \cong r \sqcup r && \text{(absorption)} \\ & \cong r && \text{(idempotency).} \end{aligned}$$

Hence by proposition 7.1 we have $r \succeq t \sqcup s$.

□

Higher order generalization is mainly used to find schemata of programs, proof, or program transformations. For example, given first order clauses

$multiply(s(X), Y, Z) \leftarrow multiply(X, Y, W), add(W, Y, Z)$, and
 $exponent(s(X), Y, Z) \leftarrow exponent(X, Y, W), multiply(W, Y, Z)$,

we can obtain its least general generalization as

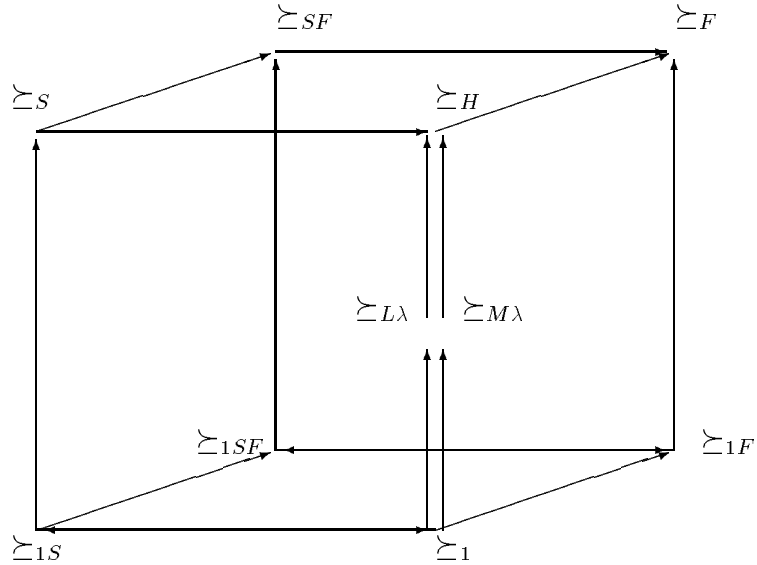
$P(s(X), Y, Z) \leftarrow P(X, Y, W), Q(W, Y, Z)$.

Higher order generalization also finds its applications in analogy analysis[8]. It is commonly recognized that a good way to obtain the concrete correspondence between two problems is to obtain the generalization of the two problem first. During the generalization process, we should preserve the structure as much as possible. By using the above higher order generalization method, we can find the analogical correspondence between two problems in the course of generalization.

5 Discussions

With the subterm restriction and the freezing extension, we defined the ordering \succeq_{SF} . As we have shown, this ordering and the corresponding generalization have nice properties almost the same as the first order anti-unification. Especially, the least general generalization exists and is unique.

To have a comparison with other kinds of generalizations, we have the following diagram:



Here each vertex represents a kind of ordering. For example, \succeq_H means the usual instantiate ordering in a higher order language, say $\lambda P2$ [1]. \succeq_1 the usual instantiation ordering in first order language, $\succeq_{M\lambda}$ the ordering in $M\lambda$, $\succeq_{L\lambda}$ the ordering in $L\lambda$ (i.e., in higher order patterns), etc.. The arrow means implication. For example, if $t \succeq_S s$, then $t \succeq_{SF} s$, and $t \succeq_H s$. It can be seen that the relations \succeq_{SF} and \succeq_H (also $\succeq_{L\lambda}$ and $\succeq_{M\lambda}$) are not comparable. By definition, \succeq_{1S} (the ordering \succeq_1 with the subterm restriction) is the same as \succeq_1 . That explains why we have good results in \succeq_{SF} .

Our work differs from the others in the following aspects. Firstly, we defined a new ordering \succeq_{SF} . In terms of this ordering, we obtain a much more specific generalization in general. For example, the terms Aab and Bab would be generalized as a single variable x in [10], or as fts in [3], where t and s are arbitrary terms. In contrast, we will have $[f]fab$ as its least general generalization. Secondly, our approach can produce a meaningful generalization of terms of different types and terms of different arities, instead a single variable x . And finally, our method is useful in applications, such as in analogical reasoning and inductive inference [5][8].

References

- [1] H. Barendregt, Introduction to generalized type systems, Journal of functional programming, Vol. 1, N0. 2, 1991. 124-154.
- [2] Coquand, T., Huet, G., The calculus of constructions, Information and Computation, Vol.76, No.3/4(1988), 95-120.

- [3] C.Feng, S.Muggleton, Towards inductive generalization in higher order logic, In D.Sleeman et al(eds.), Proceedings of the Ninth International Workshop on Machine Learning, San Mateo, California, 1992. Morgan Kaufman.
- [4] M. Hagiya, Generalization from partial parametrization in higher order type theory, Theoretical Computer Science, Vol.63(1989), pp.113-139.
- [5] R.Hasker, The replay of program derivations, Ph.D. thesis, Department of Computer Science, University of Illinois at Urbana-Champaign, 1995.
- [6] G.P.Huet, A unification algorithm for typed lambda calculus, Theoretical Computer Science, 1 (1975), 27-57.
- [7] G.Huet, Bernard Lang, Proving and applying program transformations expressed with second order patterns, Acta Informatica 11, 31-55(1978)
- [8] Jianguo Lu, Jiafu Xu, Analogical Program Derivation based on Type Theory, Theoretical Computer Science, Vol.113, North Holland 1993, pp.259-272.
- [9] Stephen Muggleton, Inductive logic programming, New generation computing, 8(4):295-318, 1991
- [10] Frank Pfenning, Unification and anti-unification in the calculus of constructions, Proceedings of the 6th symposium on logic in computer science, 1991. pp.74-85.
- [11] John C. Reynolds, Transformational systems and the algebraic structure of atomic formulas, Machine Intelligence 5, Edinburgh University Press 1970, 135-151.