

# Generalization in $\lambda 2$

Jianguo Lu <sup>\*</sup>, Masateru Harao <sup>†</sup>, Masami Hagiya <sup>‡</sup>

**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  $\theta_1$  and  $\theta_2$ , 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 example, the common generalizations of  $Aa$  and  $Bb$  without restriction would be:  $fx$ ,  $fa$ ,  $fb$ ,  $fab$ ,  $fA$ ,  $fB$ , ...,  $f(Aa, Bb)$ ,  $f(g(A, B), g(a, b))$ , ..., 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  $\lambda$  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  $\lambda 2$ [1]. The reason to choose  $\lambda 2$  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  $\lambda 2$ , 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].

---

<sup>\*</sup>Address: Robotics Institute, School of Computer Science, Carnegie Mellon University, Pittsburgh PA 15213, USA. Email: [jglu@cs.cmu.edu](mailto:jglu@cs.cmu.edu).

<sup>†</sup>Address: Department of Artificial Intelligence, Kyushu Institute of Technology, Iizuka 820, Fukuoka, Japan. Email: [harao@dumbo.ai.kyutech.ac.jp](mailto:harao@dumbo.ai.kyutech.ac.jp).

<sup>‡</sup>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](mailto:hagiya@is.s.u-tokyo.ac.jp).

- 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*.

The syntax of the restricted  $\lambda 2$  can be defined as follows[1]. Here for the purpose of convenience, we use  $[x : \sigma]$  instead of  $\lambda x : \sigma$ . Also, we use the same notation  $[V]$  to denote  $\Lambda V$  (and  $\forall V$ ), since we can distinguish among  $\lambda, \Lambda$  and  $\forall$  from the context.

We call a term  $t$  is valid (under  $\Gamma$ ) if there is a type  $\sigma$  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  $\phi$  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  $[x_1, x_2, x_3 : \gamma]Ax_1x_3$ .

## 2 Application orderings

### 2.1 Application ordering ( $\succ$ )

**Definition 1** ( $\succ$ ) *Given two terms  $t$  and  $s$ .  $t$  is more general than  $s$  (denoted as  $t \succ s$ ) 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  $\succ$  with the usual instantiation ordering (denote it as  $\geq$ ), we call  $\succ$  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.*

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

$\succ$  is reflexive and transitive:

**Proposition 1** *For any terms  $t, t_1, t_2, t_3$ ,  $t \succ t$ . If  $t_1 \succ t_2$ ,  $t_2 \succ t_3$ , then  $t_1 \succ t_3$ .*

## 2.2 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 2 (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  $\eta$  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 3 ( $\succeq_S$ )** *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.*

An examples of the relation is  $[f][x]fx \succeq_S Aa$ . Due to the finiteness of  $\text{subterm}(s)$ , the ordering  $\succeq_S$  becomes much easier to manage than  $\succeq$ :

**Proposition 2** *For any terms  $t_1, t_2, t_3$ , there exists a procedure to decide if  $t_1 \succeq_S t_2$ . If  $t_1 \succeq_S t_2$ ,  $t_2 \succeq_S t_3$ , then  $t_1 \succeq_S t_3$ .*

## 2.3 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 4 ( $\succeq_F$ )**  *$t$  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 5 ( $\succeq_{SF}$ )**  *$t \succeq_{SF} s$ , if either  $t \succeq_S s$ , or For an arbitrary type constant or term constant  $c$  such that  $sc$  is valid, and  $t \succeq_{SF} sc$ .*

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. Another example is:

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

**Proposition 3** For any terms  $t$  and  $s$ ,  $t \succeq_{SF} s$  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  $t \succeq_S sc_1c_2\dots c_k$ . There exists a procedure to decide if  $t \succeq_{SF} s$ .

**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  $h = h'$ ,  $m = n$ .

**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  $t_1 \succeq_{SF} t_3$ .

**Definition 6** ( $\cong$ )  $t \cong s$  is defined as  $t \succeq_{SF} s$  and  $s \succeq_{SF} t$ .

**Example 3**  $[x, y]Axy \cong [y, x]Axy \cong [z, x, y]Axy$ .

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

### 3 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:

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

$$\begin{aligned}
Gen(t, s, \mathcal{C}) &= [\Delta, \Delta', \Delta_k][x : \sigma_1, \dots, \alpha_j, \dots \rightarrow \sigma_{k+1}]x(t'_1, t'_2, \dots, t'_k); \\
\mathcal{C} &:= \{((h, h'(r_1, \dots, r_i)), x)\} \cup \mathcal{C}; \\
\text{Case 2.2.2.2: } \neg \exists \alpha. ((\sigma_j, \tau_j), \alpha_j) \in \mathcal{C}: \\
Gen(t, s, \mathcal{C}) &= [\Delta, \Delta', \Delta_k][\alpha_j][x : \sigma_1, \dots, \alpha_j, \dots \rightarrow \sigma_{k+1}]x(t'_1, t'_2, \dots, t'_k); \\
\mathcal{C} &:= \{((h, h'(r_1, \dots, r_i)), x)\} \cup \mathcal{C}; \\
\mathcal{C} &:= \{((\sigma_j, \tau_j), \alpha_j)\} \cup \mathcal{C}.
\end{aligned}$$

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

**Example 4** *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 5** *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*

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

*Then*

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

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  $s$ ,  $Gen(t, s, \{\})$  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  $t \sqcup s \cong t \cong s$ .

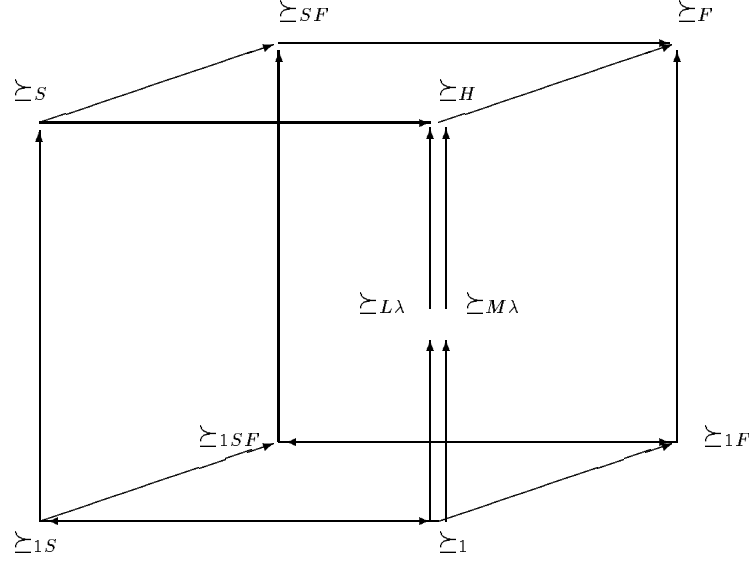
Based on the above propositions, we can have

**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  $r \succeq t \sqcup s$ .

## 4 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.