Higher order generalization and its application in program verification

Jianguo Lu, John Mylopoulos
Department of Computer Science, University of Toronto
{jglu,jm}@cs.toronto.edu

Masateru Harao
Department of Artificial Intelligence, Kyushu Institute of Technology
harao@dumbo.ai.kyutech.ac.jp

Masami Hagiya
Department of Information Science, University of Tokyo
hagiya@is.s.u-tokyo.ac.jp

Generalization is a fundamental operation of inductive inference. While first order syntactic generalization (anti-unification) is well understood, its various extensions are often needed in applications. This paper discusses syntactic higher order generalization in a higher order language $\lambda 2$[1]. Based on the *application ordering*, we prove that least general generalization exists for any two terms and is unique up to *renaming*. An algorithm to compute the least general generalization is also presented. To illustrate its usefulness, we propose a program verification system based on higher order generalization that can reuse the proofs of similar programs.
Keywords: higher order logic, unification, anti-unification, generalization, program verification.

1. Introduction

The word "generalization" is ubiquitous and one can find it used 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 [17], theorem proving [19], program derivation [6,9], and machine learning[18]. In a strict technical sense, generalization is a dual problem to that of first order unification and is often called (ordinary) anti-unification [1]. More specifically, it can be formulated as follows: 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 1970 [22,20]. They proved the existence of a unique least general generalization for first-order terms and came up with a generalization algorithm. However, due to the fact that it is inadequate for many problems, there have been many extensions of ordinary anti-unification along different directions.

One direction of extending the anti-unification problem is to take into consideration some kinds of background information as in [17]. One typical example is the relative least general generalization under θ subsumption [21]. There are various generalization methods in the area of inductive logic programming. More recently, there have been proposals for generalization operations under implication[13], and in constraint logic[18].

Another direction of extension is to promote the order of the underlying language. The problem with higher order generalization is that without some restrictions, generalization is not well-defined. For example, suppose we have two terms Aa and Bb, where A and B are functional constants and a and b are individual constants. The common generalizations of Aa and Bb without restriction could be any of the following: 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 an infinite number of generalizations in this simple example. [3] regards all these generalizations are equal up to renaming, hence in their framework least general generalization exists and is unique. 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 exists and is unique. The study is directly motivated by our research on analogical(inductive) programming and analogical(inductive) theorem proving[15,7]. The most closely related works are [19,3]. Other related works are [4,5].

[19] studied generalization in a restricted form of calculus of constructions [2], where terms are higher-order patterns, i.e., free variables can only apply

[1] The words generalization and anti-unification are often used interchangeably. Here we will use anti-unification to denote the pure syntactic first order anti-unification, i.e., instantiation as the ordering, Robinson's formulation as the language. We use generalization to denote its various extensions.

to distinct bound variables. One problem with the generalization in higher-order patterns is that of overgeneralization. Taking the above example, the least generalization of Aa and Ba would be a single variable x instead of fa or fx. 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. For example, the generalization of $[x : N]fac(succ(x))$ and $[x : N]sum(succ(x))$ would be $[f : N \to N][x : N]f(x)$, while in most cases we would hope that the generalization would be $[f : N \to N][x : N]f(succ(x))$. In fact, in higher-order patterns, all n-ary functions having different heads will be generalized into the same term $[x_1, x_2, ..., x_n]f(x_1, x_2, ..., x_n)$. The structure inside each term is not considered at all by the generalization operation.

This motivated the study of generalization in $M\lambda$ [3]. In Mλ, free variables can apply to an 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 adds some restrictions. One restriction is that $M\lambda$ is situated in a simply typed λ calculus instead of calculus of constructions. Another restriction is that $M\lambda$ 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*. This means that any two terms beginning with functional variables are considered equal.

Unlike other approaches, which mainly put restrictions on the situated language, we focus on restricting the notion of the ordering between terms. Our discussion is situated in a restricted form of the language $\lambda 2$[1]. The reason for choosing $\lambda 2$ is that it is a simple calculus which allows type variables. It can be used to formalize various concepts in programming languages, such as type definitions, abstract data types, and polymorphism. It would also be desirable if we could situate our discussion in LF[8]. But LF does not have type variables, which means that we could only generalize two terms of the same type. The one restriction we added to $\lambda 2$ is that abstractions should not occur inside arguments. This restriction is required so that we can use the results of [12]. 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 [20,22,19].
- A kind of restriction on orderings, called *subterm restriction* (the correspond-

ing 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 aforementioned ordering.

Based on the $\succeq_{SF}$ ordering, we have the following results similar to those for 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 rest of the paper is structured as follows. In the next section we introduce some basic notations used in this paper. In Section 3, we present various orderings, i.e., the usual application ordering($\succeq$), the application ordering with subterm restriction($\succeq_S$), and the application ordering with subterm restriction and variable freezing extension($\succeq_{SF}$). In section 4 we provide the generalization procedure. In Section 5, we demonstrate how generalization is used in program verification.

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:
$V = \{\alpha, \alpha_1, \alpha_2, ...\}$, (type variables),
$C = \{\gamma, \gamma_1, \gamma_2, ...\})$, (type constants),
$T = V|C|T \rightarrow T|[V]T$, (types).
The set of terms is defined as:
$X = \{x, x_1, x_2, ...\}$, (variables),
$A = \{a, a_1, a_2, ...\}$, (constants),
$\Lambda_1 = X|A|\Lambda_1\Lambda_1|\Lambda T$,(terms without abstraction),
$\Lambda = \Lambda_1|[X:T]\Lambda|[V]\Lambda$, (terms).

Here for purposes 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:

1. $\dfrac{(x : \sigma) \in \Gamma}{\Gamma \vdash x : \sigma}$ $(start)$
2. $\dfrac{\Gamma \vdash t : (\sigma \to \tau) \quad \Gamma \vdash s : \sigma}{\Gamma \vdash ts : \tau}$ $(\to E)$
3. $\dfrac{\Gamma, x : \sigma \vdash t : \tau}{\Gamma \vdash [x : \sigma]t : (\sigma \to \tau)}$ $(\to I)$
4. $\dfrac{\Gamma \vdash t : [\alpha]\sigma}{\Gamma \vdash t\tau : \sigma[\tau/\alpha]}$ $(\forall E)$
5. $\dfrac{\Gamma \vdash t : \sigma \quad \alpha \notin FV(\Gamma)}{\Gamma \vdash [\alpha]t : [\alpha]\sigma}$ $(\forall I)$

We say that 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, unless specified otherwise, we assume that all terms are closed, and in long $\beta\eta$ normal form. The symbol $=$ denotes $\alpha\beta\eta$-convertibility. Given $\Delta \equiv [x_1 : \sigma_1][x_2 : \sigma_2]...[x_n : \sigma_n]$ and term t, $[\Delta]t$ denotes $[x_1 : \sigma_1][x_2 : \sigma_2]...[x_n : \sigma_n]t$. When type information is not important, $[x : \sigma]t$ is abbreviated as $[x]t$. $[x, y : \sigma]$ is an abbreviation for $[x : \sigma][y : \sigma]$, and $\sigma_1, \sigma_2, ..., \sigma_k \to \sigma_{k+1}$ is an abbreviation for $\sigma_1 \to \sigma_2 \to ... \to \sigma_k \to \sigma_{k+1}$. As usual, terms are associated to the left, i.e., $tsr \equiv (ts)r$. Sometimes, we write tsr as $t(s, r)$. Types are associated to the right, i.e., $\alpha \to \beta \to \gamma \equiv \alpha \to (\beta \to \gamma)$.

Following [19], 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 $\{1, 2, ..., n\}$ into $\{1, 2, ..., p\}$. A *renaming* of a term $[x_1 : \sigma_1][x_2 : \sigma_2]...[x_p : \sigma_p]t$ is a valid and closed term $[x_{\phi(1)} : \sigma_{\phi(1)}][x_{\phi(2)} : \sigma_{\phi(2)}]...[x_{\phi(n)} : \sigma_{\phi(n)}]t$. Intuitively, *renaming* amounts to permuting variables, also dropping some of the abstractions when allowed. For example, $[x_3, x_1 : \gamma]Ax_1x_3$ is a renaming of $[x_1, x_2, x_3 : \gamma]Ax_1x_3$.

We will also use the following conventions unless specified otherwise. $\alpha, \alpha_1, ..., \beta, \beta_1$ ... range over type variables, i.e., the elements of V. $\gamma, \gamma_1, ...$ range over type constants, i.e., the elements of C. $\sigma, \sigma_1, ..., \tau, \tau_1, ...$ range over arbitrary elements of the set T. $x, x_1, x_2..., y, y_1, y_2, ..., z, z_1, z_2, ...$ range over both term variables and type variables, i.e., the elements in $V \cup X$. $a, b, ...$ range over term constants, i.e., the elements in A. $c, c_1, c_2, ...$ range over both term constants and type constants, i.e., the arbitrary elements in $C \cup A$, $t, t_1, t_2, ..., s, s_1, s_2, ..., r, r_1, r_2, ...$ range over terms and types, i.e., the elements in $\Lambda \cup T$.

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 $r_1, r_2, ..., r_k$, such that $tr_1r_2...r_k$ is valid, and $tr_1r_2...r_k = s$. Here k is a natural number.

To distinguish $\succeq$ with the usual instantiation ordering (denote it as $\geq$. $t \geq s$ if there exist a substitution θ such that $t\theta = s$.), 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 $\succeq_F$ will be defined in section 3.3.

Example 4. The following are some examples of the application ordering.

$[\alpha][f : \alpha \to \alpha \to \alpha][x, y : \alpha]fxy$
$\succeq [f : \gamma \to \gamma \to \gamma][x, y : \gamma]fxy$
$\succeq [x, y : \gamma]Axy$
$\succeq [y : \gamma]Aay$
$\succeq Aab.$

Proposition 5. $\succeq$ is reflexive and transitive.

Proof. The reflexivity is trivial. For the transitivity, suppose $t_1 \succeq t_2$, $t_2 \succeq t_3$. There is a sequence of terms or types $r_{11}, r_{12}, ..., r_{1n}$, $r_{21}, r_{22}, ..., r_{2m}$, such that $t_1r_{11}r_{12}...r_{1n} = t_2$, and $t_2r_{21}r_{22}...r_{2m} = t_3$, hence $t_1r_{11}r_{12}...r_{1n}r_{21}r_{22}...r_{2m}{=}t_3$. □

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 6 (subterm).The set of subterms of term t (denoted as $subterm(t)$) is defined as $norm(decm(t)) \cup \{Typ(r) | r \in decm(t)\}$.
Here $norm(t)$ returns the $\beta\eta$ normal form for the term t. $decm(r)$ is to decompose terms recursively into a set of its components, which is defined as:

1. $decm(c) = \{c\}$ (constants are unaffected by $decm$);
2. $decm(z) = \{\}$; (variables are filtered out);
3. $decm(ts) = decm(t) \cup decm(s) \cup \{ts\}$, if there are no variables in ts;
   $\quad = decm(t) \cup decm(s)$, otherwise;
4. $decm([d]t) = decm(t)$.

Example 7. Assume $A : \gamma \to \gamma \to \gamma, B : \gamma \to \gamma$,

$$\begin{aligned}
&subterm([x : \gamma]Axa)\\
&\quad = norm(decm([x : \gamma]Axa)) \cup \{Typ(r) \mid r \in decm([x : \gamma]Axa)\}\\
&\quad = norm(decm(Axa)) \cup \{Typ(r) \mid r \in decm([x : \gamma]Axa)\}\\
&\quad = norm(A, a) \cup \{Typ(r) \mid r \in decm([x : \gamma]Axa)\}\\
&\quad = \{[x, y : \gamma]Axy, a\} \cup \{Typ(r) \mid r \in \{A, a\}\}\\
&\quad = \{[x, y : \gamma]Axy, a, \gamma, \gamma \to \gamma \to \gamma\}\\
&subterm([f : \gamma \to \gamma][x : \gamma]f(Bx)) = \{[x : \gamma]Bx, \gamma \to \gamma\}.
\end{aligned}$$

As we can see, the subterms do not contain free variables. Actually, there are no bound variables except for the term having its η normal form (the term $[x, y : \gamma]Axy$ in the above example). Here we exclude the *identity* and *projection* functions as subterms. This is essential to guarantee that there exists least generalization in the application ordering. The intuition behind this is that when we match two higher order terms, in general there are *imitation* rule and *projection* rule [11]. Here only *imitation* rule is used. For our purpose, it is *projection* rule that brings about the unpleasant results and additional complexities in higher order generalizations.

Definition 8 ($\succeq_S$).

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 $r_1, r_2, ..., r_k$, such that $tr_1r_2...r_k = s$. Here $r_i \in subterm(s), i \in \{1, 2, ..., k\}$, and k is a natural number.

Example 9. Here are some examples of the $\succeq_S$ relation.
$[f][x]fx \succeq_S Aa$;
$[f][x]fx \succeq_S Bbc$;
$[\alpha][x : \alpha]x \succeq_S [x : \gamma]x \succeq_S Aa$;
$[f][x]fx \not\succeq_S a$, since the only subterm of a is a.

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

Proposition 10. $\succeq_S$ is decidable, reflexive, and transitive.

Proof. The decidability follows from the fact that the $subterms$ of t_1 are finite. The reflexivity is obvious. For the transitivity, suppose $t_1 \succeq_S t_2$, $t_2 \succeq_S t_3$. By definition of $\succeq_S$, there is a sequence of terms or types $r_{11}, r_{12}, ..., r_{1n} \in subterm(t_2)$, $r_{21}, r_{22}, ..., r_{2m} \in subterm(t_3)$, such that $t_1r_{11}r_{12}...r_{1n} = t_2$, and $t_2r_{21}r_{22}...r_{2m} = t_3$. Hence $t_1r_{11}r_{12}...r_{1n}r_{21}r_{22}...r_{2m} = t_3$. Besides, since we can not eliminate constants in t_2 when applying terms to it, and $r_{11}, r_{12}, ..., r_{1n}$ are constants in t_2, so $r_{11}, r_{12}, ..., r_{1n}$ must also be subterms of t_3. Hence we have $t_1 \succeq_S t_3$. □

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

Consider the following two terms:
$t \equiv [x][y]Axy$,
$s \equiv [x]Axa$.

In the instantiation ordering, suppose x and y are free variables, Axy is more general than Axa by the substitution $[a/y]$. Or, by application interpretation of substitution, $([y]Axy)a = Axa$. But $t \not\succeq_S s$, i.e., we can not find a term r such that $tr = s$. The problem is, before we instantiate y, we must instantiate x first.

To address this problem, we propose the following:

Definition 11 ($\succeq_F$).

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, then try to do generalization. The word *freeze* comes from [12], 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 12 ($\succeq_{SF}$).
$t \succeq_{SF} s$, 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 13. The following relations hold:

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

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

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

$[\alpha][f:\alpha\to\alpha][x:\alpha]fx \not\succeq_{SF} [\alpha][x:\alpha]x$, since *identity* and *projection* functions are not subterms.

Proposition 14. For any terms t and s,

1. $t \succeq_{SF} s$ iff there exists a sequence (possibly empty) of new, distinct constants $c_1, c_2, ..., c_k$, such that $sc_1c_2...c_k$ is of atomic type, and $t \succeq_S sc_1c_2...c_k$.
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 proof is trivial. Now suppose s is of type $\sigma \to \tau$, 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 $c_1, c_2, ..., c_k$, such that $sc_1c_2...c_k$ is of atomic type, and $t \succeq_S sc_1c_2...c_k$. By definition of $\succeq_{SF}$, $t \succeq_{SF} sc_1c_2...c_{k-1}$, $t \succeq_{SF} sc_1c_2...c_{k-2}$, ..., $t \succeq_{SF} s$.

2. Since $t \succeq_{SF} s$ iff $t \succeq_S sc_1c_2...c_k$, 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 $c_1, c_2, ..., c_k$, such that $rc_1c_2...c_k$ is of atomic type, and $t \succeq_S rc_1c_2...c_k$. Moreover, there exists a sequence of terms or types $r_1, ..., r_i$, such that $tr_1...r_i = rc_1c_2...c_k$. Since $t = s$, we have $sr_1...r_i = rc_1c_2...c_k$, $s \succeq_{SF} r$.
   The second proposition can be proved in a similar way.

□

Proposition 15. Suppose $t_1 \equiv [\Delta]hs_1s_2...s_m$, $t_2 \equiv [\Delta']h's'_1s'_2...s'_n$, and $t_1 \succeq_{SF} t_2$, then

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

Proof. Suppose $[\Delta'] = [z_1, z_2, ..., z_j]$. Since $t_1 \succeq_{SF} t_2$, we have $[\Delta]hs_1s_2...s_m \succeq_S (h's'_1s'_2...s'_m)[\bar{c}/\bar{z}]$, where $\bar{z}$ is a sequence $z_1, ..., z_j$, $\bar{c}$ is a sequence of new constant symbols $c_1, ..., c_j$. Now, suppose each variable in $h's'_1s'_2...s'_m$ is fixed as a new constant. Then $hs_1s_2...s_m$ should match $h's'_1s'_2...s'_m$ in the sense of [Huet78]. As we know, the complete minimal matches are generated by the imitation rule and the projection rule. Since the substitutions in the projection rule are $\{h \to [x_1, ..., x_m]x_i | i \in \{1, ..., m\}\}$. They do not satisfy our subterm restriction (remember the projection functions like $[x_1, ..., x_m]x_i$ will never be a subterm of any term). Thus the only way to match two terms is by using the imitation rule. By imitation rule we have substitutions

$\{h \to [x_1, ..., x_m]h'(h_1 x_1 ... x_m)...(h_n x_1 ... x_m)\}$, where $h_1, ... h_n$ are new variables. On the other hand, the subterms of $h' s'_1 s'_2 ... s'_n$ [2] whose head is h' could only be:

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

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

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

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

Proposition 16. 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 assume that
   $t_1 \equiv [\Delta]h s_1 s_2 ... s_m$,
   $t_2 \equiv [\Delta']h' r_{11} ... r_{1i} s'_1 s'_2 ... s'_m$,
   $t_3 \equiv [\Delta'']h'' r_{21} ... r_{2j} r_{31} ... r_{3i} s''_1 s''_2 ... s''_m$.

   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 $k \in \{1, ..., m\}$. 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 $h'' r_{21} ... r_{2j} r_{31} ... r_{3i}$.

□

[2] Here the variables are frozen.

Definition 17 ($\cong$).

$t \cong s$ is defined as $t \succeq_{SF} s$ and $s \succeq_{SF} t$.

Example 18. $[x, y]Axy \cong [y, x]Axy \cong [z, x, y]Axy$. This is because

$[x, y]Axy \succeq_S ([y, x]Axy)ab$, hence

$[x, y]Axy \succeq_{SF} [y, x]Axy$.

Similarly, it can be derived that

$[y, x]Axy \succeq_{SF} [x, y]Axy$,

$[x, y]Axy \succeq_{SF} [z, x, y]Axy$, and

$[z, x, y]Axy \succeq_{SF} [x, y]Axy$.

Proposition 19. $t \cong s$ iff t is a renaming of s.

Proof.

($\Rightarrow$) Assume $t \cong s$, then $t \succeq_{SF} s$ and $s \succeq_{SF} t$. Suppose $t \equiv [\Delta]ht_1t_2...t_m$, $s \equiv [\Delta']h's_1s_2...s_n$. Since $t \succeq_{SF} s$, we have $m \geq n$. Similarly, we have $n \geq m$. So, $m = n$. If h is a constant, then $h' = h$. Similarly, we note that 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 differ 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 $k \in \{1, ..., m\}$. By inductive hypothesis, $[\Delta]t_k$ and $[\Delta']s_k$ only differ 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

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

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

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

Case 2. Suppose

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

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

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

Case 3. Suppose

$t \equiv [g : \gamma_1 \to \gamma_2 \to ... \to \gamma_i \to \gamma]gt_1t_2...t_m$,
$s \equiv [f : \gamma_{\phi(1)} \to \gamma_{\phi(2)} \to ... \to \gamma_{\phi(i)} \to \gamma]ft_{\phi(1)}t_{\phi(2)}...t_{\phi(m)}$,
$tA = s([x_{\phi(1)} : \gamma_{\phi(1)}, x_{\phi(2)} : \gamma_{\phi(2)}, ..., x_{\phi(i)} : \gamma_{\phi(i)}]Ax_1x_2...x_i)$, hence $t \cong s$.

$\square$

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 is 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 that 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:

$t \equiv [\Delta]h(t_1, t_2, ..., t_k)$,
$s \equiv [\Delta']h'(r_1, ..., r_i, s_1, s_2, ..., s_k)$, where h and h' are atoms. Suppose
$[\Delta, \Delta', \Delta_1]t'_1 = Gen([\Delta]t_1, [\Delta']s_1, \mathcal{C})$,
$[\Delta, \Delta', \Delta_2]t'_2 = Gen([\Delta, \Delta_1]t_2, [\Delta', \Delta_1]s_2, \mathcal{C})$,
...
$[\Delta, \Delta', \Delta_k]t'_k = Gen([\Delta, \Delta_{k-1}]t_k, [\Delta', \Delta_{k-1}]s_k, \mathcal{C})$,
$Typ(h) = \sigma_1, \sigma_2, ..., \sigma_k \to \sigma_{k+1}$,
$Typ(h'(r_1, ..., r_i)) = \tau_1, \tau_2, ..., \tau_k \to \tau_{k+1}$.

The generalization algorithm could be defined as in figure 1.

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

Example 20. Some examples of least general generalization.

$[x : \gamma]x \sqcup Aa = [x : \gamma][\alpha][y : \alpha]y \cong [\alpha][y : \alpha]y$, if Aa is not of type γ;
$[x : \gamma]x \sqcup Aa = [x : \gamma][y : \gamma]y \cong [x : \gamma]x$, if Aa is of type γ;

$Gen(t, s, \mathcal{C})$:
Case 1: $h = h' : Gen(t, s, \mathcal{C}) = [\Delta, \Delta', \Delta_k]h(t'_1, t'_2, ..., t'_k)$;
Case 2: $h \neq h'$:
Case 2.1: $\exists x.((h, h'(r_1, ..., r_i)), x) \in \mathcal{C}$:
$Gen(t, s, \mathcal{C}) = [\Delta, \Delta', \Delta_k]x(t'_1, t'_2, ..., t'_k)$
Case 2.2: $\neg\exists x.((h, h'(r_1, ..., r_i)), x) \in \mathcal{C}$,
Case 2.2.1: $Typ(h) = Typ(h'(r_1, ..., r_i))$:
$Gen(t, s, \mathcal{C}) = [\Delta, \Delta', \Delta_k][x : \sigma_1, \sigma_2, ..., \sigma_k \to \sigma_{k+1}]x(t'_1, t'_2, ..., t'_k)$
$\mathcal{C} := \{((h, h'(r_1, ..., r_i)), x)\} \cup \mathcal{C}$;
Case 2.2.2: $Typ(h) \neq Typ(h'(r_1, ..., r_i))$:
Not losing generality, suppose $\sigma_j \neq \tau_j, j \in \{1, 2, ..., k, k+1\}$.
Case 2.2.2.1: $\exists \alpha_j.((\sigma_j, \tau_j), \alpha_j) \in \mathcal{C}$:
$Gen(t, s, \mathcal{C}) = [\Delta, \Delta', \Delta_k][x : \sigma_1, ..., \alpha_j, ... \to \sigma_{k+1}]x(t'_1, t'_2, ..., t'_k)$;
$\mathcal{C} := \{((h, h'(r_1, ..., r_i)), x)\} \cup \mathcal{C}$;
Case 2.2.2.2: $\neg\exists \alpha.((\sigma_j, \tau_j), \alpha_j) \in \mathcal{C}$:
$Gen(t, s, \mathcal{C}) =$
$[\Delta, \Delta', \Delta_k][\alpha_j][x : \sigma_1, ..., \alpha_j, ... \to \sigma_{k+1}]x(t'_1, t'_2, ..., t'_k)$;
$\mathcal{C} := \{((h, h'(r_1, ..., r_i)), x)\} \cup \mathcal{C}$;
$\mathcal{C} := \{((\sigma_j, \tau_j), \alpha_j)\} \cup \mathcal{C}$.

Figure 1. Generalization Algorithm

$[x]Axx \sqcup [x]Aax \cong [x, y]Axy$;
$Aa \sqcup Bb \cong [f][x]fx$, if A and B are of the same type;
$Aa \sqcup Bb \cong [\alpha][f : \alpha \to \gamma][x : \alpha]fx$, if $A : \gamma_1 \to \gamma$ and $B : \gamma_2 \to \gamma$;

Example 21. Here is an example of generalizing segments of programs. For clarity the segments are written in the usual notation. Let
$t \equiv [x]map1(cons(a, x)) = cons(succ(a), map1(x))$,
$s \equiv [x]map2(cons(a, x)) = cons(sqr(a), map2(x))$.
Suppose the types are
$map1 : List(Nat) \to Nat$; $succ : Nat \to Nat$,
$map2 : List(Nat) \to Nat$; $sqr : Nat \to Nat$.
Then
$t \sqcup s \cong$

$[f : List(Nat) \to Nat; g : Nat \to Nat)][x]$
$f(cons(a, x)) = cons(g(a), f(x)))$.

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 22. 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 $t \sqcup s \cong t \cong s$.

Proof.

1. It can be verified that for each case of the algorithm, we obtain a more general term.
2. It is obvious since we decompose the terms recursively.
3. Since $t \succeq s$, we can suppose
   $t \equiv [\Delta]hs_1s_2...s_m$,
   $s \equiv [\Delta']h'r_{11}...r_{1i}s'_1s'_2...s'_m$,and
   $[\Delta]s_k \succeq [\Delta']s'_k, k \in \{1, ..., m\}$.
   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 $h'r_{11}...r_{1i}$. Other cases can be proved in a similar way. Now we can suppose $t \sqcup s \equiv [\Delta''][f]ft_1t_2...t_m$.
   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 if s_k has

only one occurrence in t. Then t' and t only differ 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 proposition 7.3 we can prove the result.

5. It is obvious from the algorithm.

6. Not losing generality, we can suppose
$t \equiv [\Delta]hs_1s_2...s_m$,
$s \equiv [\Delta']h'r_{11}...r_{1i}s'_1s'_2...s'_m$,
$r \equiv [\Delta'']h''r_{21}...r_{2j}r_{31}...r_{3i}s''_1s''_2...s''_m$,
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 $k \in \{1, ..., m\}, p \in \{1, ..., i\}$, 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 $t_1, ..., t_m$, $t'_1, ..., t'_m$, and $t_1, ..., t''_m$, respectively.
We rename the variables in $[\Gamma]t_1, ..., [\Gamma]t_m$ such that there are multiple occurrences of a variable x in $[\Gamma]t_1, ..., [\Gamma]t_m$ if and only if its corresponding places in $s_1, ..., s_m$ hold a same term, and its corresponding places in $s'_1, ..., s'_m$ hold another same term. Similarly, we rename the terms $[\Gamma']t'_k, [\Gamma'']t''_k$. Then
$(t \sqcup s) \sqcup r$
$\cong [\Gamma][f]ft_1...t_m \sqcup [\Delta'']h''r_{21}...r_{2j}r_{31}...r_{3i}s''_1s''_2...s''_m$
$\cong [\Gamma''][f]ft''_1...t''_m$,
$t \sqcup (s \sqcup r)$
$\cong [\Delta]hs_1s_2...s_m \sqcup [\Gamma'][g]gr_1...r_it'_1...t'_m$
$\cong [\Gamma''][g]gt''_1...t''_m$.
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 $[\Delta]hr_1r_2...r_n$ and $[\Delta']hr_1r_2...r_n$. It is obvious that $[\Delta]hr_1r_2...r_n \sqcup r \cong [\Delta']hr_1r_2...r_n \sqcup r$.

8. Since $t \succeq s$, we have $t \sqcup s \cong t$, hence
   $t \sqcup r$
   $\cong (t \sqcup s) \sqcup r$ (by proposition 7.7)
   $\cong t \sqcup (s \sqcup r)$ (commutativity)
   $\succeq s \sqcup r$ (by proposition 7.1).

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 now prove:

Theorem 23. $t \sqcup s$ 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 $r \sqcup t \cong r$, $r \sqcup s \cong r$.
$r \sqcup (t \sqcup s)$
$\cong (r \sqcup r) \sqcup (t \sqcup s)$ (idempotency)
$\cong (r \sqcup t) \sqcup (r \sqcup s)$ (commutativity and associativity)
$\cong r \sqcup r$ (absorption)
$\cong r$ (idempotency).
Hence by proposition 7.1 we have $r \succeq t \sqcup s$. □

Higher order generalization can be used to find schemata of programs, proofs, 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 can also find applications in analogy analysis[14,9]. It is commonly recognized that a good way to obtain the concrete correspondence between two problems is to obtain the generalization of the two problems first. During the generalization process, we should preserve 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.

In the following section we will introduce the application of higher order generalization in reusing program proofs.

5. Reuse of program proofs

The verification of the correctness of software being developed is proved extremely difficult. We propose a method to reuse program proofs based on our higher order generalization method. A type checker(proof checker) is implemented, and more proof examples are available at http://www.cs.toronto.edu/ jglu/proof.html.

The prefix notation of the syntax of a small programming language can be defined as:

Definition 24 (Syntax of a small language).

$Expr : Nat$

$Com : Type$

$assignment : [Var, Expr]Com$

$composition : [Com, Com]Com$

$ifthenelse : [Prop, Com, Com]Com$

$while : [Prop, Com]Com$

$hformula : [Prop, Com, Prop]Prop$

We use the well-known Hoare's notation[10] to denote the notions of program correctness. The Hoare formula $hformula(P, C, Q)$ means that if the pre-condition P holds before the execution of the program C, and C terminates, then after the execution the post-condition Q holds.

In the following, for the sake of clarity, the syntax of the small language is written in infix notation instead of prefix notation. For example, the Hoare formula $hformula(P, C, Q)$ is written as the usual form $\{P\}C\{Q\}$. The assignment statement $assignment(x, t)$ is written as $x := t$.

Now we can define the axiomatic semantics of the language as below:

Definition 25 (Axiomatic semantics).

$assign : [P][t : Expr][x : Var](\{([x]P)t\}x := t\{P\})$

$seq : [P, Q, R : Prop][c, d : Com]$

$\quad [p_1 : \{P\}c\{Q\}]$

$[p_2 : \{Q\}d\{R\}]$
$\{P\}c; d\{R\}$
$if : [P, Q, B : Prop]$
$[c, d : Com]$
$[p_1 : \{P \wedge B\}c\{Q\}]$
$[p_2 : \{P \wedge notB\}d\{Q\}]$
$\{P\}ifBthencelsed\{Q\}$
$while : [P, B : Prop]$
$[c : Com]$
$[p : \{P \wedge B\}c\{P\}]$
$\{P\}whileBdoc\{P \wedge notB\}$
$conR : [p_1 : \{P\}c\{Q\}]$
$[p_2 : [Q]R]$
$\{P\}c\{R\}$
$conL : [p_1 : \{P\}c\{Q\}]$
$[p_2 : [R]P]$
$\{R\}c\{Q\}$

These are the usual axiomatic rules encoded in the higher order logic. Taking the sequential rule *seq* for example, it is actually saying that if P, Q and R are propositions, c and d are commands, and if p_1 is a proof of $\{P\}c\{Q\}$, p_2 is a proof of $\{Q\}d\{R\}$, then $seq(P, Q, R, c, d, p_1, p_2)$ is the proof of $\{P\}c; d\{R\}$.

Here we follow the practice of intuitionistic type theory such as the one given by Martin-lof[16], where we can treat propositions as types. In type theory, a term t is a proof of a formula P can be denoted as t is of type P.

With the definition of the syntax and the semantics of this small language, we can now prove the correctness of programs.

Example 26. The proof of
$[P : Prop]\{([x : Nat]P)2\}x := 2\{P\}$
is
$[P]assign(P, 2, x)$,
which can be verified by showing that $[P]assign(P, 2, x)$ is of type
$[P : Prop]\{([x : Nat]P)2\}x := 2\{P\}$

Example 27. The following is a function and the specification to compute the maximal of natural numbers.

$H_1 \equiv [x, y, z : Nat]$
$\{true\}$
$if(x \geq y)then(z := x)else(z := y)$
$\{(z = x \vee z = y) \wedge z \geq x \wedge z \geq y\}$

Its proof is

$P_1 \equiv if(true, ((z = x \vee z = y) \wedge z \geq x \wedge z \geq y), x \geq y, assign(x, z), assign(y, z))$

Example 28. Now suppose we have the following specification and the program, which is to compute the minimal of two natural numbers:

$H_2 \equiv [x, y, z : Nat]$
$\{true\}$
$if(x \leq y)then(z := x)else(z := y)$
$\{(z = x \vee z = y) \wedge z \leq x \wedge z \leq y\}$

In general it is not easy to prove a program is correct with respect to a specification. But with the similar program in *Example 25*, we can, first, get the least general generalization of H_1 and H_2, which amounts to

$H \equiv [x, y, z : Nat]$
$\{true\}$
$if(x \otimes y)then(z := x)else(z := y)$
$\{(z = x \vee z = y) \wedge (z \otimes x) \wedge (z \otimes y)\}$

where $\otimes$ is a variable of type $Nat \to Nat \to Nat$.

With this generalization, we can find that a mapping exists between the operators $\geq$ and $\leq$. By replacing the $\geq$ in P_1 with $\leq$, we obtained a new proof P_2:

$P_2 \equiv if(true, ((z = x \vee z = y) \wedge z \leq x \wedge z \leq y), x \leq y, assign(x, z), assign(y, z))$

Finally, by running the type checker, P_2 is verified to be a proof of H_2 indeed.

Of course, most of the proof reusing case will be much more complicated as described above. [14,15] describe some rules and heuristics to map those correspondences.

6. Conclusions

Using subterm restriction and a freezing extension, we define the ordering $\succeq_{SF}$. As we have shown, this ordering and the corresponding generalization have

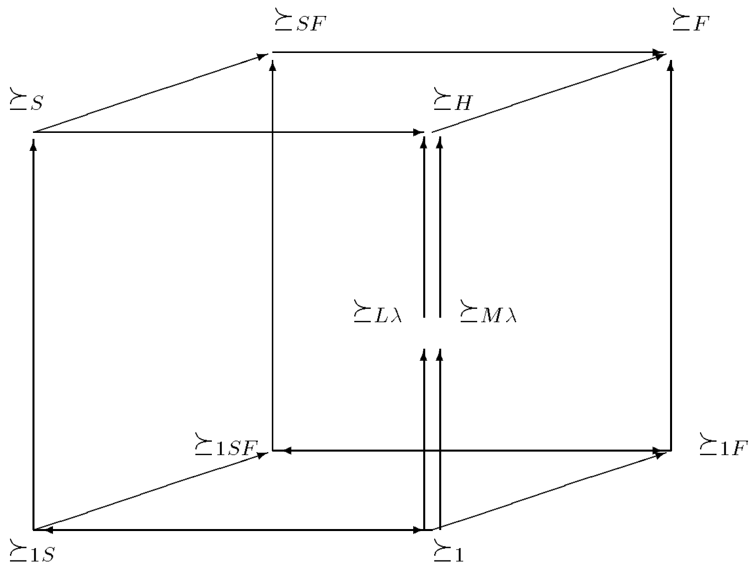

Figure 2. Generalization cube

nice properties, comparable to those of the first order anti-unification. Most notably, the least general generalization exists and is unique.

To summarize how our proposed higher order generalization compares with other kinds of generalizations, we offer the generalization cube as depicted in picture 2.

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 in the diagram represent 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 with $\succeq_{SF}$.

Our work differs from that of others in the following aspects. Firstly, we define a new ordering $\succeq_{SF}$. In terms of this ordering, we obtain a much more specific generalization in most of the case. For example, the terms Aab and Bab would be generalized as a single variable x in [19], or as fts in [3], where t and s

are arbitrary terms. In contrast, our generalization algorithm would return $[f]fab$ as least general generalization. Secondly, our approach can produce a meaningful generalization of terms of different types and terms of different arities, instead of a single variable x as in [19,3]. For example, our method will be able to produce the generalization of Aab and Bb as fb. And finally, our method is useful in applications, such as in analogical reasoning and inductive inference [9,14]. We also demonstrated in this paper its application in program verification.

Acknowledgements

We would like to thank the anonymous reviewers for their in depth comments and detailed suggestions.

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] K. Furukawa, M. Imai, and Randy Goebel, Hyper least general generalization and its application to higher-order concept learning, manuscript draft.

[5] Gegg-Harrison, Timothy S., Basic Prolog Schemata, CS-1989-20, Department of Computer Science, Duke University, 1989.

[6] M. Hagiya, Generalization from partial parametrization in higher order type theory, Theoretical Computer Science, Vol.63(1989), pp.113-139.

[7] Masateru Harao, Proof Discovery in LK System by Analogy, LNCS1345,pp.197-211, Proc. of ASIAN'97, 1997.12.

[8] Robert Harper, Furio Honsell, and Gordon Plotkin, A framework for defining logics. Journal of the Association for Computing Machinery, 40(1), 143-184, January 1993.

[9] R.Hasker, The replay of program derivations, Ph.D. thesis, Department of Computer Science, University of Illinois at Urbana-Champaign, 1995.

[10] C.A.R.Hoare, An axiomatic approach to computer programming, Communications of the ACM, 12, Oct. 1969.

[11] G.P.Huet, A unification algorithm for typed lambda calculus, Theoretical Computer Science, 1 (1975), 27-57.

[12] G.Huet, Bernard Lang, Proving and applying program transformations expressed with second order patterns, Acta Informatica 11, 31-55(1978)

[13] Peter Idestam-Almquist, Generalization of Horn clauses, Ph.D. dissertation, Department of Computer Science and Systems Science, Stockholm University and the Royal Institute of Technology, 1993.

[14] Jianguo Lu, Jiafu Xu, Analogical Program Derivation based on Type Theory, Theoretical Computer Science, Vol.113, North Holland 1993, pp.259-272.

[15] Jianguo Lu, Bo Yi, An Approach to Analogical Theorem Proving, in Shi(ed.), IFIP Transactions A-19, Automated Reasoning, North Holland, 1992, pp. 285-294.

[16] P. Martin-lof, Intuitionistic type theory, Studies in Proof Theory, Bibliopolis, Napoli, 1984.

[17] Stephen Muggleton, Inductive logic programming, New generation computing, 8(4):295-318, 1991.

[18] Charles David Page jr., Anti-unification in constraint logic: foundations and applications to learnability in first order logic, to speed-up learning, and to deduction, Ph.D. Dissertation, University of Illinois at Urbana-Champaign, 1993.

[19] 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.

[20] Plotkin, G. D., A note on inductive generalization, Machine Intelligence 5, Edinburgh University Press 1970, pp. 153-163.

[21] Plotkin, G.D., A further note on inductive generalization, Machine Intelligence 6, Edinburgh University Press 1971, pp. 101-124.

[22] John C. Reynolds, Transformational systems and the algebraic structure of atomic formulas, Machine Intelligence 5, Edinburgh University Press 1970, 135-151.