Order-Sorted Generalization

María Alpuente a,1  Santiago Escobar a,2  José Meseguer b,3  Pedro Ojeda a,4

a Dpto. de Sistemas Informáticos y Computación, Universidad Politécnica de Valencia, Spain
b Department of Computer Science, University of Illinois at Urbana-Champaign, USA.

Abstract

Generalization, also called anti-unification, is the dual of unification. Given terms \( t \) and \( t' \), a generalization is a term \( t'' \) of which \( t \) and \( t' \) are substitution instances. The dual of a most general unifier (mgu) is that of least general generalization (lgg). In this work, we extend the known untyped generalization algorithm to an order-sorted typed setting with sorts, subsorts, and subtype polymorphism. Unlike the untyped case, there is in general no single lgg. Instead, there is a finite, minimal lgg's, so that any other generalization has at least one of them as an instance. Our generalization algorithm is expressed by means of an inference system for which we give a proof of correctness. This opens up new applications to partial evaluation, program synthesis, and theorem proving for typed reasoning systems and typed rule-based languages such as ASF+SDF, Elan, OBJ, Cafe-OBJ, and Maude.

Keywords: least general generalization, partial evaluation, order–sorted reasoning

1 Introduction

Generalization, also called anti-unification, is the dual of unification. Given terms \( t \) and \( t' \), a generalization of \( t \) and \( t' \) is a term \( t'' \) of which \( t \) and \( t' \) are substitution instances. The dual of a most general unifier (mgu) is that of least general generalization (lgg), that is, a generalization that is a substitution instance of any other one. Generalization is a formal reasoning component of many program analysis and transformation methods, including theorem provers, and program analysis and transformation tools (see, e.g., [12,22,8,24]).

Although generalization goes back to work of Plotkin [25], Reynolds [27], and Huet [14] and has been studied in detail by other authors (see for example the survey [17]), to the best of our knowledge, all generalization algorithms, with the exception

---

1 Email: alpuente@dsic.upv.es
2 Email: sescobar@dsic.upv.es
3 Email: meseguer@cs.uiuc.edu
4 Email: pojeda@dsic.upv.es
5 This work has been partially supported by the EU (FEDER) and the Spanish MEC TIN2007-68093-C02-02 project, UPV PAID-06-07 project, and Generalitat Valenciana GV06/285 and BFPI/2007/076 grants.

This paper is electronically published in
Electronic Notes in Theoretical Computer Science
URL: www.elsevier.nl/locate/entcs
of the works of Pfenning on generalization in the higher-order setting of the calculus of constructions [24], assume an untyped setting. However, many applications, for example to partial evaluation, theorem proving, and program learning, for typed rule-based languages such as ASF+SDF [6], Elan [7], OBJ [13], CafeOBJ [11], and Maude [9], require a first-order typed version of generalization which does not seem to be available: we are not aware of any existing algorithm. Moreover, several of the above-mentioned languages have an expressive order-sorted typed setting with sorts, subsorts (where subsort inclusions form a partial order and are interpreted semantically as set-theoretic inclusions of the corresponding data sets), and subsort-overloaded function symbols (a feature also known as subtype polymorphism), so that a symbol, for example +, can simultaneously exist for various sorts in the same subsort hierarchy, such as + for natural, integers, and rationals, and its semantic interpretations agree on common data items.

In a way similar to the dual case of order-sorted unification, a case which, in contrast, has indeed been studied in detail (see, e.g. [28,21,30]), the extension of the generalization algorithm to the order-sorted setting is nontrivial. In particular, the existence and uniqueness of generalizations is typically lost. That is, first of all there is no lgg at all if two terms are unrelated in the sort ordering; and if they are related (their sorts are both in the same connected component of the partial order of sorts), then there is in general no single lgg associated to a pair of terms. Instead, there is a finite and minimal set of least general generalizations, so that any other generalization has at least one of those as an instance. Such a set of lggs is the dual analogue of a minimal and complete set of unifiers for non-unitary unification algorithms, such as those for order-sorted unification, e.g., [28,21,30], and for equational unification (see, e.g., [5,29]). Note that this situation is quite different from the higher-order typed generalization algorithm of Pfenning [24], where for any two higher-order patterns, either there is no lgg (because the types are incomparable), or there is a unique lgg. A related definition of generalization is given in [1] for feature terms, an extended notion of terms that is also based on ordered sorts.

As it is usual in current treatments of different formal deduction mechanisms, and has become standard for the dual case of unification algorithms since Martelli and Montanari (see, e.g., [18,15]), we specify the generalization process by means of an inference system rather than by an imperative-style algorithm. Even for the known untyped generalization case, which we present as a special case to motivate its order-sorted extension, this has several expository and conceptual advantages, and we give an inference system that to the best of our knowledge is new. After this, we show how our unsorted calculus naturally extends to the new order-sorted generalization algorithm. We illustrate the use of the inference rules with several examples. Finally, we give a proof of correctness of our inference system.

As already mentioned, this opens up new applications to partial evaluation, program synthesis, and inductive theorem proving for first-order typed rule-based languages such as ASF+SDF, Elan, OBJ, CafeOBJ, and Maude, and to theorem proving tools, program learning tools, and partial evaluators for such languages. In our own work, we plan to use the above order-sorted generalization algorithm as a key component of a narrowing-based partial evaluator (PE) for programs in order-
sorted rule-based languages such as OBJ, CafeOBJ, and Maude. This will make available for such languages useful narrowing based PE techniques developed for the untyped setting in, e.g., [3,4]. We are also considering adding this generalization mechanism to an inductive theorem prover such a Maude’s ITP [10] to support automatic conjecture of lemmata. This will provide a typed analogue of similar automatic lemma conjecture mechanisms in untyped first-order inductive theorem provers such as Nqthm [8] and its ACL2 successor [16].

Related work

Plotkin [25] and Reynolds [27] gave an imperative–style algorithm for generalization, which are both essentially the same. Huet’s generalization algorithm [14], formulated as a pair of recursive equations, cannot be understood as an automated calculus, since some implicit (imprecise) assumptions in the treatment of variables are made, which are made explicit in our formulation. A deterministic reconstruction of Huet’s algorithm is given in [23] which does not consider types either. An operational definition of the least general generalization of clauses based on (order–sorted) feature terms is given in [1]. Finally, the algorithm for generalization in the calculus of constructions of [24] cannot be used for order-sorted theories.

2 Preliminaries

We follow the classical notation and terminology from [31] for term rewriting and from [19,20] for rewriting logic and order-sorted notions. We assume an order-sorted signature Σ with a finite poset of sorts (S, ≤) and a finite number of function symbols. We furthermore assume that: (i) each connected component in the poset ordering has a top sort, and for each s ∈ S we denote by [s] the top sort in the component of s; and (ii) for each operator declaration f : s_1 × ... × s_n → s in Σ, there is also a declaration f : [s_1] × ... × [s_n] → [s]. Throughout this paper, we assume that Σ has no ad-hoc operator overloading, i.e., any two operator declarations for the same symbol f : s_1 × ... × s_n → s and f : s'_1 × ... × s'_n → s', must necessarily have [s_1] = [s'_1], ..., [s_n] = [s'_n], [s] = [s'].

We assume an S-sorted family X = {X_s}_{s ∈ S} of disjoint variable sets with each X_s countably infinite. A fresh variable is a variable that appears nowhere else. T_Σ(X) is the set of terms of sort s, and T_Σ,s is the set of ground terms of sort s. We write T(Σ,X) and T(Σ) for the corresponding term algebras. We assume that T_Σ,s ≠ ∅ for every sort s.

For a term t, we write Var(t) for the set of all variables in t. Term positions are represented as strings of natural numbers and are endowed with the prefix ordering ≤ on strings. The set of positions of a term t is written Pos(t), and the set of non-variable positions Pos_Σ(t). The root position of a term is ω. The subterm of t at position p is t|_p and t|_p[u] is the term t where t|_p is replaced by u. By root(t) we denote the symbol occurring at the root position of t.

A substitution σ is a sorted mapping from a finite subset of X, written Dom(σ), to T(Σ,X). The set of variables introduced by σ is Ran(σ). The identity substitution is id. Substitutions are homomorphically extended to T(Σ,X). The application of a substitution σ to a term t is denoted by tσ. The restriction of σ to a set of vari-
ables $V$ is $\sigma|_V$. Composition of two substitutions is denoted by juxtaposition, i.e., $\sigma\sigma'$. We call a substitution $\sigma$ a renaming if there is another substitution $\sigma^{-1}$ such that $\sigma\sigma^{-1}|\text{Dom}(\sigma) = \text{id}$. Substitutions are sort–preserving, i.e., for any substitution $\sigma$, if $x \in \mathcal{X}_s$, then $x\sigma \in \mathcal{T}_S(\mathcal{X})_s$.

We write the sort associated to a variable explicitly with a colon and the sort, i.e. $x:\text{Nat}$. We assume pre-regularity of the signature $\Sigma$, ensuring that every term $t$ has a unique least sort, denoted by $\text{LS}(t)$. Therefore, the top sort in the connected component of $\text{LS}(t)$ is denoted by $[\text{LS}(t)]$. Since the poset $(\mathcal{S}, \leq)$ is finite and each connected component has a top sort, given any two sorts $s$ and $s'$ in the same connected component, the set of least upper bound sorts of $s$ and $s'$, although not necessarily a singleton set, always exists and is denoted by $\text{LUBS}(s,s')$.

### 3 Untyped Least General Generalization

We revisit untyped generalization, going back to Plotkin [25], Reynolds [27], and Huet [14], giving a new inference system that will be useful in our subsequent extension of this algorithm to the order–sorted setting given in Section 4. Throughout this section, we assume terms $t \in \mathcal{T}_S(\Sigma, \mathcal{X})$ for $\Sigma$ an unsorted signature (i.e., there is only one sort).

Let $\leq$ be the standard instantiation quasi–ordering on terms given by the relation of being “more general”, i.e. $t$ is more general than $s$ (i.e. $s$ is an instance of $t$), written $t \leq s$, iff there exists $\theta$ such that $t\theta = s$. Most general unification of a (unifiable) set $M$ is the least upper bound (most general instance) of $M$ under $\leq$. Generalization corresponds to the greatest lower bound. Given a non–empty set $M$ of terms, the term $w$ is a generalization of $M$ if, for all $s \in M$, $w \leq s$. A term $w$ is the least general generalization of $M$ if $w$ is a generalization of $M$ and, for each other generalization $u$ of $M$, $u \leq w$.

The non-deterministic generalization algorithm $\lambda$ of Huet [14] (also treated in detail in [17]) is as follows. Let $\Phi$ be any bijection between $\mathcal{T}(\Sigma, \mathcal{X}) \times \mathcal{T}(\Sigma, \mathcal{X})$ and a set of variables $V$. The recursive function $\lambda$ on $\mathcal{T}(\Sigma, \mathcal{X}) \times \mathcal{T}(\Sigma, \mathcal{X})$ that computes the lgg of two terms is given by:

- $\lambda(f(s_1, \ldots, s_m), f(t_1, \ldots, t_m)) = f(\lambda(s_1, t_1), \ldots, \lambda(s_m, t_m))$, for $f \in \Sigma$
- $\lambda(s, t) = \Phi(s, t)$, otherwise.

Central to this algorithm is the global function $\Phi$ that is used to guarantee that the same disagreements are replaced by the same variable in both terms.

However, there are some implicit (imprecise) assumptions in the algorithm regarding the treatment of variables that are hidden in the global function $\Phi$. Actually, an explicit condition on $\Phi$ for handling variables is missing, namely that variables of the original terms should not be kept by $\Phi$. For instance, consider the following function $\Phi$ from [17] which does not fulfill this condition: $\Phi(x, a) = x$ and $\Phi(b, a) = y$. This function gives that $\lambda(f(x, f(x, f(a, b))))$ changes the original term to $f(x, f(x, f(a, y)))$, which is correct. However, if we add $\Phi(x, b) = x$, this function produces an erroneous result in $\lambda(f(x, x), f(a, b)) = f(x, x)$. The solution to repair this flaw is to replace all occurrences of a variable $x$ in the original terms with a new constant $c_x$ before using the algorithm and, after using the algorithm, replace...
In the following, we provide a novel set of inference rules for computing the least generalization (lgg) of two terms, avoiding implicit, obscure assumptions by using a store of already solved generalization sub-problems. This algorithm can also be used (thanks to associativity and commutativity of lgg) to compute the lgg of an arbitrary set of terms by successively computing the lgg of two elements of the set in the obvious way.

In our reformulation, we represent a generalization problem between terms \( s \) and \( t \) as a constraint \( s \equiv x \equiv t \), where \( x \) is a fresh variable that stands for a (most general) generalization of \( s \) and \( t \). By means of this representation, any generalization \( w \) of \( s \) and \( t \) is given by a substitution \( \theta \) such that \( x\theta = w \).

We compute the least general generalization of \( s \) and \( t \) by means of a transition system \((\text{Conf}, \rightarrow)\) \[26\] where \( \text{Conf} \) is a set of configurations and the transition relation \( \rightarrow \) is given by a set of inference rules. Besides the constraint component, i.e., a set of constraints of the form \( t_i \equiv t_i' \), and the substitution component, i.e., the partial substitution computed so far, configurations also include an extra component, called the store. This store\(^6\) plays the role of the function \( \Phi \) of Huet’s generalization algorithm, with the difference that our stores are local to the system configurations, whereas \( \Phi \) can instead be understood as a global repository. We note that the non–globality of the store will be the key for computing a minimal and complete set of solutions for the order–sorted case.

**Definition 3.1** A configuration, written as \( \langle C \mid S \mid \theta \rangle \), consists of three components:

- the constraint component \( C \), i.e., a conjunction \( s_1 \equiv x_1 \equiv t_1 \land \ldots \land s_n \equiv x_n \equiv t_n \) that represents the set of unsolved constraints
- the store component \( S \), that records the set of already solved constraints, and
- the substitution component \( \theta \), that consists of bindings for some of the variables \( x_1, \ldots, x_n \) present in constraints \( s_i \equiv t_i \) of \( C \) and \( S \).

Starting from the initial configuration \( \langle t \equiv x \equiv t' \mid \emptyset \mid \text{id} \rangle \), configurations are transformed until a terminal configuration \( \langle \emptyset \mid S \mid \theta \rangle \) is reached. Then, the lgg of \( t \) and \( t' \) is given by \( x\theta \). As we will see, \( \theta \) is unique up to renaming.

The transition relation \( \rightarrow \) is given by the smallest relation satisfying the rules in Figure 1. In this paper, variables of terms \( t \) and \( s \) in a generalization problem \( t \equiv x \equiv s \) are considered as constants, since they are never instantiated. The meaning of the rules is as follows.

- The rule **Decompose** is the syntactic decomposition generating new constraints to be solved.
- The rule **Recover** checks if a constraint \( t \equiv x \equiv s \in C \) with \( \text{root}(t) \neq \text{root}(s) \), is already solved, i.e., there is already a constraint \( t \equiv y \equiv s \in S \) for the same conflict pair

---

\(^6\) Our notion of store appears to be comparable to the history set \( \text{Hist} \) of [1], though we came up with the idea of store independently.
Example 3.2 Let \( t = f(g(a), g(y), a) \) and \( s = f(g(b), g(y), b) \) be two terms. We apply the inference rules of Figure 1 and the substitution obtained by the lgg algorithm is \( \theta = \{ x \mapsto f(g(x_4), g(y), x_4), x_1 \mapsto g(x_4), x_2 \mapsto g(y), x_5 \mapsto y, x_3 \mapsto x_4 \} \), where the lgg is \( x\theta = f(g(x_4), g(y), x_4) \). Note that variable \( x_4 \) is repeated, to ensure the least general generalization of two terms is unique up to variable renaming \([17]\).
\[
\text{lgg}(f(g(a), g(y), a), f(g(b), g(y), b))
\]

\[\begin{array}{l}
\text{Initial Configuration} \\
(f(g(a), g(y), a) \equiv f(g(b), g(y), b) \mid \emptyset \mid id) \\
\text{Decompose} \\
(g(a) \equiv g(b) \land g(y) \equiv g(y) \land a \equiv b \mid \emptyset \mid \{x \mapsto f(x_1, x_2, x_3)\}) \\
\text{Decompose} \\
(a \equiv b \land g(y) \equiv g(y) \equiv a \equiv b \mid \emptyset \mid \{x \mapsto f(x_4, x_2, x_3, x_1 \mapsto g(x_4)\}) \\
\text{Decompose} \\
(g(y) \equiv g(y) \land a \equiv b \mid a \equiv b \mid \{x \mapsto f(x_4, x_2, x_3, x_1 \mapsto g(x_4)\}) \\
\text{Decompose} \\
(y \equiv y \land a \equiv b \mid a \equiv b \mid \{x \mapsto f(x_4, g(x_3), x_1 \mapsto g(x_4), x_2 \mapsto g(x_5)\}) \\
\end{array}
\]

\[
\text{Recover} \\
(\emptyset \mid a \equiv b \mid \{x \mapsto f(x_4, g(y), x_4, x_1 \mapsto g(x_4), x_2 \mapsto g(y), x_3 \mapsto y, x_3 \mapsto x_4)\})
\]

Figure 2. Computation trace for unsorted generalization of terms \(f(g(a), g(y), a)\) and \(f(g(b), g(y), b)\)

position of \(t\) and \(t'\) if \(\text{root}(t|_p) \neq \text{root}(t'|_p)\) and for all \(q < p, \text{root}(t_q) \equiv \text{root}(t'_q)\), and the pair \((t|_p, t'|_p)\) is then called a conflict pair of \(t\) and \(t'\). Also, note that given a constraint \(\equiv t'\), \(x\) is always a (most general) generalization of \(t\) and \(t'\).

**Lemma 3.4** Given terms \(t\) and \(t'\) and a fresh variable \(x\) such that \(\{t \equiv t' \mid \emptyset \mid id\} \rightarrow^* \{\emptyset \mid S \mid \emptyset\}\), a constraint \(u \equiv v\) is in \(S\) iff there exists a conflict position \(p\) of \(t\) and \(t'\) such that \(t|_p = u\) and \(t'|_p = v\).

**Lemma 3.5** Given terms \(t\) and \(t'\) and a fresh variable \(x\) such that \(\{t \equiv t' \mid \emptyset \mid id\} \rightarrow^* \{C \mid S \mid \emptyset\}\), then \(x\theta\) is a generalization of \(t\) and \(t'\).

**Lemma 3.6** Given terms \(t\) and \(t'\) and a fresh variable \(x\) such that \(\{t \equiv t' \mid \emptyset \mid id\} \rightarrow^* \{\emptyset \mid S \mid \emptyset\}\), then \(\{y \in X \mid \exists u \equiv v \in S\} \subseteq \text{Ran}(\theta)\), and \(\text{Ran}(\theta) = \text{Var}(x\theta)\).

Soundness and completeness is proved as follows.

**Theorem 3.7** Given terms \(t\) and \(t'\) and a fresh variable \(x\), \(u\) is the lgg of \(t\) and \(t'\) if and only if \(\{t \equiv t' \mid \emptyset \mid id\} \rightarrow^* \{\emptyset \mid S \mid \emptyset\}\) and there is a renaming \(\rho\) s.t. \(u\rho = x\theta\).

**Proof** We rely on the already known existence and uniqueness of the lgg of \(t\) and \(t'\) and reason by contradiction. By Lemma 3.5, \(x\theta\) is a generalization of \(t\) and \(t'\). If \(x\theta\) is not the lgg of \(t\) and \(t'\), then there is a term \(u\) which is the lgg of \(t\) and \(t'\) and a substitution \(\rho\) such that \(u\rho\) is not a variable renaming and \(x\theta\rho = u\). Since, by Lemma 3.6, \(\text{Ran}(\theta) = \text{Var}(x\theta)\), we can always choose \(\rho\) with \(\text{Dom}(\rho) = \text{Var}(x\theta)\). If \(\rho\) is not a variable renaming, either:

(i) there are variables \(y, y' \in \text{Var}(x\theta)\) and a variable \(z\) such that \(y\rho = y'\rho = z\), or

(ii) there is a variable \(y \in \text{Var}(x\theta)\) and a non-variable term \(v\) such that \(y\rho = v\).

In case (i), there are two conflict positions \(p, p'\) for \(t\) and \(t'\) such that \(u|_p = z = u|_{p'}\) and \(x\theta|_p = y\) and \(x\theta|_{p'} = y'\). In particular, this means that \(t|_p = t|_{p'}\) and \(t'|_p = t'|_{p'}\).
But this is impossible by Lemmas 3.4 and 3.6. In case (ii), there is a position \( p \) such that \( x \theta[p] = y \) and \( p \) is neither a conflict position of \( t \) and \( t' \) nor it is under a conflict position of \( t \) and \( t' \). But this is impossible by Lemmas 3.4 and 3.6. \( \square \)

4 Order–sorted Least General Generalizations

In this section, we generalize to the order–sorted setting the unsorted generalization algorithm presented in Section 3.

We consider two terms \( t \) and \( t' \) having the same top sort, otherwise they are incomparable and no generalization exists. Starting from the initial configuration \( \langle t \hat{=} t' \mid \emptyset \mid id \rangle \) where \( [s] = [LS(t)] = [LS(t')] \), configurations are transformed until a terminal configuration \( \langle \emptyset \mid S \mid \emptyset \rangle \) is reached. In the order–sorted setting the lgg in general, is not unique. Each terminal configuration \( \langle \emptyset \mid S \mid \emptyset \rangle \) provides an lgg of \( t \) and \( t' \) given by \( (x:[s]) \theta \).

The transition relation \( \rightarrow \) is given by the smallest relation satisfying the rules in Figure 3. The meaning of these rules is as follows.

- The rule Decompose is the syntactic decomposition generating new constraints to be solved. Fresh variables are initially assigned a top sort, which will be appropriately “downgraded” when necessary.
- The rule Recover is similar to the corresponding rule of Figure 1.
- The rule Solve checks that a constraint \( t \hat{=} t' \in C \), with \( root(s) \neq root(t) \), is not already solved. Then the solved constraint \( t \hat{=} t' \) is added to the store \( S \), and the substitution \( \{x \rightarrow z\} \) is composed with the substitution part, where \( z \) is a fresh variable with sort in the \( LUBS \) of the least sorts of both terms. Note that this is the only additional source of non-determinism (besides the choice of the constraint to work on) in our inference rules, in contrast to Figure 1. This extra non–determinism causes our rules to be non–confluent in general.

**Example 4.1** Let \( t = f(x:A) \) and \( s = f(y:B) \) be two terms where \( x \) and \( y \) are variables of sorts \( A \) and \( B \) respectively, and the sort hierarchy is shown in Figure 5. The typed definition of \( f \) is \( f : E \rightarrow E \). Starting from the initial configuration \( \langle f(x:A) \hat{=} f(y:B) \mid \emptyset \mid id \rangle \), we apply the inference rules of Figure 3 and the substitutions obtained by the lgg algorithm are \( \theta_1 = \{z:E \mapsto f(z_2:C), z_1:E \mapsto z_2:C\} \) and \( \theta_2 = \{z:E \mapsto f(z_3:D), z_1:E \mapsto z_3:D\} \), where the lgg is either \( (z:E) \theta_1 = f(z_2:C) \) or \( (z:E) \theta_2 = f(z_3:D) \). Note that \( \theta_1 \) and \( \theta_2 \) are incomparable, so that we have two possible lgg. The computation of both solutions is shown in Figure 4.

Before proving the correctness of the above inference system, we give an abstract characterization of the set of lgs of two terms \( t \) and \( t' \) such that \( [LS(t)] = [LS(t')] \). To simplify our notation, in what follows, we write \( t[s]p_1,\ldots,p_n \) instead of \( ((t[s]p_1)\cdots)[s]p_n \).

**Definition 4.2** Given terms \( t \) and \( t' \) such that \( [LS(t)] = [LS(t')] \), let \( (u_1,v_1),\ldots, (u_k,v_k) \) be the conflict pairs of \( t \) and \( t' \), and for each such conflict pair \( (u_i,v_i) \), let \( p_1^i,\ldots,p_n^i \) be the corresponding conflict positions, and let \( [s_i] = [LS(u_i)] = [LS(v_i)] \).
Decompose

\[
\begin{align*}
& \text{if } f \in (\Sigma \cup \mathcal{X}) \land f : [s_1] \times \ldots \times [s_n] \to [s] \\
& \forall x : [s]\ f(x_1, \ldots, x_n) = f(s_1, \ldots, s_n) \land C \mid S \mid \theta) \to \\
& \{x_1 \vdash s_1, \ldots, x_n \vdash s_n \mid s_n \land C \mid S \mid \theta\sigma\}
\end{align*}
\]

where \( \sigma = \{x_1 : [s] \mapsto f(x_1 : [s_1], \ldots, x_n : [s_n])\}, x_1 : [s_1], \ldots, x_n : [s_n] \) are fresh variables, and \( n \geq 0 \)

Solve

\[
\begin{align*}
\text{root}(t) \not\equiv \text{root}(t') \land s' \in \text{LUBS}(LS(t), LS(t')) \land \exists y : \exists z : t \not\equiv t' \in S
\end{align*}
\]

where \( \sigma = \{x : [s] \mapsto z : s'\} \) and \( z : s' \) is a fresh variable.

Recover

\[
\begin{align*}
\text{root}(t) \not\equiv \text{root}(t')
\end{align*}
\]

where \( \sigma = \{x : [s] \mapsto y : s'\} \)

Figure 3. Rules for order-sorted least general generalizations.

\[
\begin{align*}
& \text{lgg}(f(xA), f(yB)) \\
& \text{Initial Configuration} \\
& \{f(xA) \not\equiv f(yB) \mid \emptyset \mid \text{id}\} \\
& \text{Decompose} \\
& \{x : A \vdash y : B \mid \{z : E \mapsto f(z_1 : E)\}\} \\
& \text{Solve} \\
& \{\emptyset \mid x : A \vdash z : C \mid \{z : E \mapsto f(z_2 : C), z_1 : E \mapsto z_2 : C\}\} \\
& \{\emptyset \mid x : A \vdash y : B \mid \{z : E \mapsto f(z_3 : D), z_1 : E \mapsto z_3 : D\}\}
\end{align*}
\]

Figure 4. Computation trace for order-sorted generalization of terms \( f(x) \) and \( f(y) \)

\[LS(v_i)\]. We define the term \( \text{lgg}^\bullet(t, t') = (\{t(x_1 : [s_1], \ldots, x_k : [s_k])\}_{i=1}^k) \mid \{x_k : [s_k]\}_{i=1}^k\}\), where \( x_1 : [s_1], \ldots, x_k : [s_k] \) are fresh variables. Furthermore, we define

\[
\text{Spec}(t, t') = \{\rho \mid \text{Dom}(\rho) = \{x_1 : [s_1], \ldots, x_k : [s_k]\}\land \forall 1 \leq i \leq k, \rho(x_i : [s_i]) = x'_i \mid x'_i \in \text{LUBS}(LS(u_i), LS(v_i))\}
\]

where all the \( x'_i \) are fresh variables, and, finally, \( \text{lgg}(t, t') = \{\text{lgg}^\bullet(t, t') \mid \rho \in \text{Spec}(t, t')\}\).

**Lemma 4.3** Given terms \( t \) and \( t' \) such that \( [LS(t)] = [LS(t')] \), \( \text{lgg}^\bullet(t, t') \) is a generalization of \( t \) and \( t' \) and \( \text{lgg}(t, t') \) provides a complete minimal set of lgg.

9
We provide some auxiliary notions and lemmas.

**Lemma 4.4** Given terms \( t \) and \( t' \) such that \([s] = [LS(t)] = [LS(t')]\), and a fresh variable \( x:[s] \) such that \((t \not\lessdot t' \mid \emptyset \mid \text{id}) \rightarrow^* \langle \emptyset \mid S \mid \theta \rangle\), and a constraint \( u \not\lessdot v \) is in \( S \) iff there exists a conflict position \( p \) of \( t \) and \( t' \) such that \( t\mid_p = u \) and \( t'\mid_p = v \), and there exist a variable name \( y \) and a sort \( s \in \text{LUBS}(LS(u), LS(v)) \) such that \( z = y:s \).

A substitution \( \delta \) is called **downgrading** if each binding is of the form \( x:s \mapsto x':s' \), where \( x \) and \( x' \) are variables and \( s' \leq s \).

**Lemma 4.5** Given terms \( t \) and \( t' \) such that \([s] = [LS(t)] = [LS(t')]\), and let \( \text{lgg}^*(t, t') \). Then, for all \( S \) and \( \theta \) such that \((t \not\lessdot t' \mid \emptyset \mid \text{id}) \rightarrow^* \langle \emptyset \mid S \mid \theta \rangle\), there exists a downgrading substitution \( \delta \) such that \( \text{lgg}^*(t, t')\delta = (x:[s])\theta \).

**Theorem 4.6** Given terms \( t \) and \( t' \) such that \([s] = [LS(t)] = [LS(t')]\), and a fresh variable \( x:[s] \), \( u \in \text{lgg}(t, t') \) is a lgg of \( t \) and \( t' \) if and only if \((t \not\lessdot t' \mid \emptyset \mid \text{id}) \rightarrow^* \langle \emptyset \mid S \mid \theta \rangle\) for some \( S \) and \( \theta \) and there is a renaming \( \rho \) s.t. \( u\rho = (x:[s])\theta \).

**Proof** We reason by contradiction. Both cases if and only if are similar and we provide only the proof for the if case.

Let us assume some \( S \) and \( \theta \) such that there are no \( u \in \text{lgg}(t, t') \) and renaming \( \rho \) s.t. \( u\rho = (x:[s])\theta \). For all \( u \in \text{lgg}(t, t') \), by Definition 4.2, \( \text{lgg}^*(t, t') \leq u \) with a downgrading substitution. By Lemma 4.5, \( \text{lgg}^*(t, t') \leq (x:[s])\theta \) with a downgrading substitution. Let \( \delta \) be the downgrading substitution such that \( \text{lgg}^*(t, t')\delta = (x:[s])\theta \). For all \( u \in \text{lgg}(t, t') \), let \( \delta_u \) be the downgrading substitution such that \( \text{lgg}^*(t, t')\delta_u = u \). Since there is no renaming between \((x:[s])\theta\) and \( u \) and both have a downgrading substitution with \( \text{lgg}^*(t, t') \), there must be a binding \( x:s \mapsto x':s' \) in \( \delta \) and a binding \( x:s \mapsto x'':s'' \) in \( \delta_u \) s.t. either \( s' < s'' \), \( s'' < s' \), or \([s'] \neq [s'']\). But the three possibilities are impossible by definition, since \( s' < s'' \) contradicts the idea that \( u \) is a lgg, \( s'' < s' \) contradicts Lemma 4.4, and \([s'] \neq [s'']\) contradicts both that \( u \) is a lgg and Lemma 4.4.

\( \square \)

5 Conclusions and Future Work

We have presented an order–sorted generalization algorithm that computes a minimal and complete set of least general generalizations for two terms. Our algorithm is directly applicable to any many-sorted, and order-sorted declarative language and reasoning system (and also, a fortiori, to untyped languages and systems which have only one sort). However, several such languages – such as ASF+SDF, OBJ, Cafe-OBJ, Elan, and Maude –, as well as various theorem proving systems, also support built-in reasoning modulo frequently occurring equational axioms such as associativity, commutativity and identity. It would therefore be highly desirable to support order–sorted generalization modulo such equational theories. In [2], we have developed a modular algorithm for a parametric family of commonly occurring equational theories, namely, for all theories \((\Sigma, E)\) such that each binary function symbol \( f \in \Sigma \) can have any combination of associativity, commutativity, and identity axioms. It would be very useful to combine the order–sorted and the \( E \)–generalization
inference systems into a single generalization calculus supporting both types and equational axioms. However, this combination seems to us non–trivial and is left for future work.

In our own work, we plan to extend the current order-sorted, syntactic generalization algorithm presented here to an order–sorted, equational one as a key component of a narrowing-based partial evaluator (PE) for programs in order-sorted rule-based languages such as OBJ, Cafe-OBJ, and Maude. This will make available for such languages useful narrowing–driven PE techniques developed for the syntactic setting in, e.g., [3,4]. We are also considering adding this generalization mechanism to an inductive theorem prover such a Maude’s ITP [10] to support automatic conjecture of lemmas. This will provide a first–order typed analogue of similar automatic lemma conjecture mechanisms in first–order untyped inductive theorem provers such as Nqthm [8] and its ACL2 successor [16].

References


