Anti-Unification for Unranked Terms and Hedges

Temur Kutsia[1], Jordi Levy[2], and Mateu Villaret[3]

1 **Research Institute for Symbolic Computation (RISC)**
Johannes Kepler University, Linz, Austria
kutsia@risc.jku.at
2 **Artificial Intelligence Research Institute (IIIA)**
Spanish Council for Scientific Research (CSIC)
Barcelona, Spain
levy@iiia.csic.es
3 **Departament d'Informàtica i Matemàtica Aplicada (IMA)**
Universitat de Girona (UdG), Girona, Spain
villaret@ima.udg.edu

Abstract

We study anti-unification for unranked terms and hedges that may contain term and hedge variables. The anti-unification problem of two hedges $\tilde{s}_1$ and $\tilde{s}_2$ is concerned with finding their generalization, a hedge $\tilde{q}$ such that both $\tilde{s}_1$ and $\tilde{s}_2$ are instances of $\tilde{q}$ under some substitutions. Hedge variables help to fill in gaps in generalizations, while term variables abstract single (sub)terms with different top function symbols. First, we design a complete and minimal algorithm to compute least general generalizations. Then, we improve the efficiency of the algorithm by restricting possible alternatives permitted in the generalizations. The restrictions are imposed with the help of a rigidity function that is a parameter in the improved algorithm and selects certain common subsequences from the hedges to be generalized. Finally, we indicate a possible application of the algorithm in software engineering.

1998 ACM Subject Classification F.4.2 [Theory of Computation]: Mathematical Logic and Formal Languages—Grammars and Other Rewriting Systems, F.2.2 [Theory of Computation]: Analysis of Algorithms and Problem Complexity—Nonnumerical Algorithms and Problems, D.2.7 [Software]: Software Engineering—Distribution, Maintenance, and Enhancement.

Keywords and phrases Anti-unification, generalization, unranked terms, hedges, software clones.

Digital Object Identifier 10.4230/LIPIcs.RTA.2011.219

Category Regular Research Paper

1 Introduction

The anti-unification problem of two terms t_1 and t_2 is concerned with finding their generalization, a term t such that both t_1 and t_2 are instances of t under some substitutions. The problem has a trivial solution, a fresh variable, that is the most general generalization of the given terms. Interesting generalizations are the least general ones. The purpose of anti-unification algorithms is to compute such least general generalizations. Plotkin [27] and Reynolds [28] pioneered research on anti-unification, designing generalization algorithms for ranked terms (where function symbols have a fixed arity) in the syntactic case. Since then, a number of algorithms and their modifications have been developed, addressing the problem in various theories (e.g., [1, 2, 4, 9, 15, 26]) and from different application points of view (e.g., [3, 8, 12, 17, 25, 31]). Applications come from the areas such as reasoning by analogy,

22nd International Conference on Rewriting Techniques and Applications (RTA'11).
Editor: M. Schmidt-Schauß; pp. 219–234

Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany


machine learning, inductive logic programming, software engineering, program synthesis, analysis, transformation, verification, just to name a few.

Unranked terms differ from the ranked ones by not having fixed arity for function symbols. Hedges are finite sequences of such terms. They are flexible structures, popular in representing semistructured data. To take the advantage of variadicity, unranked terms and hedges use two kinds of variables: term variables that stand for a single term and hedge variables that stand for hedges. Solving techniques over unranked terms and hedges mostly address unification and matching problems, see, e.g., [11, 18, 19, 20, 21, 23, 24]. Anti-unification for these structures practically has not been studied. The only exceptions, to the best of our knowledge, are [3, 33], where anti-unification of feature terms, and a special case of so called simple hedges are considered, respectively.

We address this shortcoming, presenting algorithms to compute least general generalizations for unranked terms/hedges. Hedge variables help to fill in gaps in generalizations, while term variables abstract single (sub)terms with different top function symbols. First, we develop a complete and minimal algorithm. Next, we improve its efficiency by restricting possible alternatives permitted in the generalizations. The restrictions are imposed with the help of a rigidity function that is a parameter in the improved algorithm. At each step, the algorithm decides which subsequence of terms of the given hedges is to be (structurally) retained in the generalization. This gives more efficient, yet pretty general algorithm for a generic rigidity function. Instantiating the parameter with specific rigidity functions, we obtain various special instances.

Finally, we discuss a possible application in software code clone detection. Our results open a possibility to address the problem of searching XML clones by means of anti-unification. Rigid hedge generalizations provide several advantages for this, combining fast textual and precise structural techniques.

2 Preliminaries

Given pairwise disjoint countable sets of unranked function symbols F (symbols without fixed arity), term variables $\mathcal{V}_\mathsf{T}$, and hedge variables $\mathcal{V}_\mathsf{H}$, we define *unranked terms* (terms in short) and *hedges* (sequences of terms or hedge variables) over F and $\mathsf{V} = \mathcal{V}_\mathsf{T} \cup \mathcal{V}_\mathsf{H}$ by the grammar: $t ::= x \mid f(\tilde{s}),\ s ::= t \mid X,\ \tilde{s} ::= s_1, \ldots, s_n$, where $x \in \mathcal{V}_\mathsf{T}$, $f \in \mathsf{F}$, $X \in \mathcal{V}_\mathsf{H}$, and $n \geq 0$. With this definition, terms are singleton hedges. Not all singleton hedges are terms: some may be hedge variables. If $\tilde{s} = s_1, \ldots, s_n$ and $\tilde{s}' = s'_1, \ldots, s'_m$, then we write $\tilde{s}, \tilde{s}'$ for $s_1, \ldots, s_n, s'_1, \ldots, s'_m$. We denote by $\tilde{s}|_i$ the ith element of $\tilde{s}$. We denote by $\tilde{s}|_i^j$, where $i < j$, the subsequence between positions i and j excluding them, i.e., the subsequence $\tilde{s}|_{i+1}, \ldots, \tilde{s}|_{j-1}$. The length of a sequence $\tilde{s}$, denoted $|\tilde{s}|$, is the number of elements in it.

The set of terms (resp., the set of hedges) over F and V is denoted by $\mathsf{T}(\mathsf{F}, \mathcal{V}_\mathsf{T}, \mathcal{V}_\mathsf{H})$ (resp., by $\mathsf{H}(\mathsf{F}, \mathcal{V}_\mathsf{T}, \mathcal{V}_\mathsf{H})$). We use the letters f, g, h, a, b, c, and d for function symbols, x, y, and z for term variables, X, Y, Z, U, and V for hedge variables, χ for a term variable or a hedge variable, t, l, and r for terms, s and q for a hedge variable or a term, and $\tilde{s}$ and $\tilde{q}$ for hedges. The empty hedge is denoted by ϵ. The terms of the form $a(\epsilon)$ are written as just a.

The *size* of a term t is the number of occurrences of symbols (from $\mathsf{F} \cup \mathsf{V}$) in it and is denoted by $\mathit{size}(t)$. We denote by $\mathit{var}(t)$ the *set of variables* of a term. These definitions are generalized for any syntactic object.

A *substitution* is a mapping from hedge variables to hedges and from term variables to terms, which is identity almost everywhere. We will use the traditional finite set representation of substitutions, writing, e.g., $\{x \mapsto f(a), X \mapsto \epsilon,\ Y \mapsto x, g(a, Z)\}$ for the substitution that

maps every variable to itself except x, X, and Y that are mapped respectively to $f(a)$, to ϵ, and to $x, g(a, Z)$. The lower case Greek letters are used to denote substitutions, with the exception of the identity substitution for which we write Id.

Substitutions can be applied to terms and hedges using the congruences

$$\sigma(f(s_1, \ldots, s_n)) = f(\sigma(s_1), \ldots, \sigma(s_n)), \qquad \sigma(s_1, \ldots, s_n) = \sigma(s_1), \ldots, \sigma(s_n).$$

We call $\sigma(s)$ and $\sigma(\tilde{s})$ the *instances* of respectively s and $\tilde{s}$ and use postfix notation to denote them, writing $s\sigma$ and $\tilde{s}\sigma$. We also say that $\tilde{s}$ is *more general* than $\tilde{q}$ if $\tilde{q}$ is an instance of $\tilde{s}$ and denote this fact by $\tilde{s} \preceq \tilde{q}$. If $\tilde{s} \preceq \tilde{q}$ and $\tilde{q} \preceq \tilde{s}$, then we write $\tilde{s} \simeq \tilde{q}$. If $\tilde{s} \preceq \tilde{q}$ and $\tilde{s} \not\simeq \tilde{q}$, then we say that $\tilde{s}$ is *strictly more general* than $\tilde{q}$ and write $\tilde{s} \prec \tilde{q}$. The set $dom(\sigma) = \{\chi \in \mathsf{V} \mid \chi\sigma \neq \chi\}$ is called the *domain* of σ.

The *composition* of two substitutions σ and ϑ, written as $\sigma\vartheta$, is defined as the composition of two mappings: We have $s(\sigma\vartheta) = (s\sigma)\vartheta$ for all s. A substitution σ_1 is *more general* than σ_2 with respect to a set of variables $\mathcal{X} \subseteq \mathsf{V}$, written $\sigma_1 \preceq_{\mathcal{X}} \sigma_2$, if there exists ϑ such that $\chi\sigma_1\vartheta = \chi\sigma_2$, for each $\chi \in \mathcal{X}$. The relations $\simeq$ and $\prec$ are extended to substitutions: $\sigma_1 \simeq_{\mathcal{X}} \sigma_2$ means $\sigma_1 \preceq_{\mathcal{X}} \sigma_2$ and $\sigma_2 \preceq_{\mathcal{X}} \sigma_1$, and $\sigma_1 \prec_{\mathcal{X}} \sigma_2$ means $\sigma_1 \preceq_{\mathcal{X}} \sigma_2$ and $\sigma_1 \not\simeq_{\mathcal{X}} \sigma_2$.

The *top symbol* of a term is defined as $top(x) = x$ and $top(f(\tilde{s})) = f$. We extend this notion to hedges, defining it as the sequence of symbols as follows: $top(\epsilon) = \epsilon$, $top(X, \tilde{s}) = X\,top(\tilde{s})$, and $top(t, \tilde{s}) = top(t)top(\tilde{s})$. Notice that we write these sequences as words, e.g., $top(f(a), a, X, x) = faXx$. The letter w will be used for those words.

A hedge $\tilde{s}$ is called a *generalization* or an *anti-instance* of two hedges $\tilde{s}_1$ and $\tilde{s}_2$ if $\tilde{s} \preceq \tilde{s}_1$ and $\tilde{s} \preceq \tilde{s}_2$. That means, there exist substitutions σ_1 and σ_2 such that $\tilde{s}_1 = \tilde{s}\sigma_1$ and $\tilde{s}_2 = \tilde{s}\sigma_2$. We say that a hedge $\tilde{s}$ is a *least general generalization* (lgg in short), aka a *most specific anti-instance*, of $\tilde{s}_1$ and $\tilde{s}_2$ if $\tilde{s}$ is a generalization of $\tilde{s}_1$ and $\tilde{s}_2$ and there is no generalization $\tilde{q}$ of $\tilde{s}_1$ and $\tilde{s}_2$ that satisfies $\tilde{s} \prec \tilde{q}$. That means, there are no generalizations of $\tilde{s}_1$ and $\tilde{s}_2$ that are strictly less general than their least general generalization.

An *anti-unification problem* (or equation), AUP in short, is a triple $\chi : \tilde{s}_1 \triangleq \tilde{s}_2$, where χ does not occur in $\tilde{s}_1$ and $\tilde{s}_2$. Intuitively, χ is a variable that stands for the most general generalization of $\tilde{s}_1$ and $\tilde{s}_2$. An *anti-unifier* of $\chi : \tilde{s}_1 \triangleq \tilde{s}_2$ is a substitution σ such that $dom(\sigma) \subseteq \{\chi\}$ and $\chi\sigma$ is a generalization of both $\tilde{s}_1$ and $\tilde{s}_2$. An anti-unifier σ of an AUP $\chi : \tilde{s}_1 \triangleq \tilde{s}_2$ is *least general (or most specific)* if there is no anti-unifier of ϑ of the same problem that satisfies $\sigma \prec_{\mathcal{X}} \vartheta$. Obviously, if σ is a least general anti-unifier of an AUP $\chi : \tilde{s}_1 \triangleq \tilde{s}_2$, then $\chi\sigma$ is a least general generalization of $\tilde{s}_1$ and $\tilde{s}_2$.

A *complete set of generalizations* of two hedges $\tilde{s}_1$ and $\tilde{s}_2$ is a set G of hedges that satisfies the properties:
Soundness: Each $\tilde{q} \in G$ is a generalization of both $\tilde{s}_1$ and $\tilde{s}_2$.
Completeness: For each generalization $\tilde{s}$ of $\tilde{s}_1$ and $\tilde{s}_2$, there exists $\tilde{q} \in G$ such that $\tilde{s} \preceq \tilde{q}$.

G is a *minimal complete set of generalizations* of $\tilde{s}_1$ and $\tilde{s}_2$ if it, in addition to soundness and completeness, satisfies also the following property:
Minimality: For each $\tilde{q}_1, \tilde{q}_2 \in G$, if $\tilde{q}_1 \preceq \tilde{q}_2$ then $\tilde{q}_1 = \tilde{q}_2$.

▶ **Lemma 2.1.** *For any three hedges $\tilde{s}_1$, $\tilde{s}_2$ and $\tilde{q}$, and any pair of substitutions σ_1 and σ_2 satisfying $\tilde{s}_1 = \tilde{q}\sigma_1$ and $\tilde{s}_2 = \tilde{q}\sigma_2$, if $size(\tilde{q}) \geq size(\tilde{s}_1) + size(\tilde{s}_2)$ then there exists a hedge variable X occurring in $\tilde{q}$ such that $X\sigma_1 = X\sigma_2 = \epsilon$.*

Proof. The hedge $\tilde{q}$ can not contain more function symbols than $\tilde{s}_1$ and $\tilde{s}_2$ do. It also can not contain more term variables than there are subterms in $\tilde{s}_1$ or $\tilde{s}_2$. Violation of any of these conditions would forbid $\tilde{s}_1$ or $\tilde{s}_2$ to be an instance of $\tilde{q}$. Hence, the only reason why $size(\tilde{q}) \geq size(\tilde{s}_1) + size(\tilde{s}_2)$ is that $\tilde{q}$ may contain extra hedge variables that are mapped to ϵ by both σ_1 and σ_2. ◀

▶ **Lemma 2.2.** *For any hedges $\tilde{s}_1$ and $\tilde{s}_2$ there exists their minimal complete set of generalizations that, modulo $\simeq$, is unique and finite.*

Proof. For classical first-order anti-unification this property is trivial, because instantiation does not decrease the size of terms. This means that anti-unifiers of two terms are smaller than each of those terms, hence finite modulo variable renaming. For hedges the property is not so simple to prove because instantiating a hedge variable by ϵ, the size of a term may decrease. However, by Lemma 2.1 we have that for any anti-unifier $\tilde{q}$ of $\tilde{s}_1$ and $\tilde{s}_2$ with $size(\tilde{q}) \geq size(\tilde{s}_1) + size(\tilde{s}_2)$ there exists another anti-unifier less general than $\tilde{q}$ (that we can obtain by replacing those extra hedge variables in $\tilde{q}$ by ϵ). The set of anti-unifiers smaller than the sum of the sizes of both hedges is a complete set of anti-unifiers, and it is finite and unique modulo $\simeq$. ◀

We denote the minimal complete set of generalizations of $\tilde{s}_1$ and $\tilde{s}_2$ by $mcg(\tilde{s}_1, \tilde{s}_2)$. Its elements are lggs of $\tilde{s}_1$ and $\tilde{s}_2$.

Like unification problems, anti-unification problems may be classified as unitary (if minimal complete sets of generalizations always exist and are singletons), finitary (if they always exist, are finite, and the problem is not unitary), infinitary (if they always exist and may be infinite), and nullary (if they may not exist). Hence, Lemma 2.2 implies that hedge anti-unification is finitary.

An anti-unification problem always has an anti-unifier. The empty substitution is a trivial example that represents the most general generalization. Our goal is to compute less general generalizations. In the next section, we design an algorithm that computes (the set of anti-unifiers that represents) the *mcg* of a given AUP. It requires some care to properly address the issues that arise because of hedge variables in generalizations.

Quiz 1: *Given two hedges $\tilde{s} = f(a), f(a)$ and $\tilde{q} = f(a), f$, what is the set $mcg(\tilde{s}, \tilde{q})$? Hint: There are three elements in $mcg(\tilde{s}, \tilde{q})$.*

Below we assume that the hedges to be generalized are variable disjoint.

3 Complete and Minimal Algorithm

We present our anti-unification algorithm as a rule-based algorithm that works on triples $A; S; \sigma$. Here A is a set of AUPs of the form $\{X_1 : \tilde{s}_1 \triangleq \tilde{q}_1, \ldots, X_n : \tilde{s}_n \triangleq \tilde{q}_n\}$ where each X_i occurs in the problem only once, S is a set of already solved anti-unification equations (the store), and σ is a substitution (computed so far)[1]. We call such a triple a *system*. The rules transform systems into systems:

T-H: **Trivial Hedge**

$$\{X : \epsilon \triangleq \epsilon\} \cup A;\ S;\ \sigma \Longrightarrow A;\ S;\ \sigma\{X \mapsto \epsilon\}.$$

Dec-T: **Decomposition for Terms**

$$\{X : f(\tilde{s}) \triangleq f(\tilde{q})\} \cup A;\ S;\ \sigma \Longrightarrow \{Y : \tilde{s} \triangleq \tilde{q}\} \cup A;\ S;\ \sigma\{X \mapsto f(Y)\}$$

where Y is a fresh variable.

Dec1-H: **Decomposition 1 for Hedges**

$$\{X : s, \tilde{s} \triangleq q, \tilde{q}\} \cup A;\ S;\ \sigma \Longrightarrow \{Y : s \triangleq q, Z : \tilde{s} \triangleq \tilde{q}\} \cup A;\ S;\ \sigma\{X \mapsto Y, Z\},$$

where $U : s, \tilde{s} \triangleq q, \tilde{q} \notin S$ for all $U \in \mathcal{V}_{\mathsf{H}}$, the variables Y and Z are fresh, and $\tilde{s} \neq \epsilon$ or $\tilde{q} \neq \epsilon$.

[1] Such a representation was first proposed in [1] for equational anti-unification.

Dec2-H: **Decomposition 2 for Hedges**

$$\{X : s, \tilde{s} \triangleq \tilde{q}\} \cup A;\ S;\ \sigma \Longrightarrow \{Y : s \triangleq \epsilon, Z : \tilde{s} \triangleq \tilde{q}\} \cup A;\ S;\ \sigma\{X \mapsto Y, Z\},$$

where $\chi : s, \tilde{s} \triangleq \tilde{q} \notin S$ for all χ, the variables Y and Z are fresh, and $\tilde{s} \neq \epsilon$ or $\tilde{q} \neq \epsilon$.

Dec3-H: **Decomposition 3 for Hedges**

$$\{X : \tilde{s} \triangleq q, \tilde{q}\} \cup A;\ S;\ \sigma \Longrightarrow \{Y : \epsilon \triangleq q, Z : \tilde{s} \triangleq \tilde{q}\} \cup A;\ S;\ \sigma\{X \mapsto Y, Z\},$$

where $\chi : \tilde{s} \triangleq q, \tilde{q} \notin S$ for all χ, the variables Y and Z are fresh, and $\tilde{s} \neq \epsilon$ or $\tilde{q} \neq \epsilon$.

Sol1-H: **Solve 1 for Hedges**

$$\{X : s \triangleq \epsilon\} \cup A;\ S;\ \sigma \Longrightarrow A;\ \{X : s \triangleq \epsilon\} \cup S;\ \sigma, \quad \text{if } Y : s \triangleq \epsilon \notin S \text{ for all } Y.$$

Sol2-H: **Solve 2 for Hedges**

$$\{X : \epsilon \triangleq q\} \cup A;\ S;\ \sigma \Longrightarrow A;\ \{X : \epsilon \triangleq q\} \cup S;\ \sigma, \quad \text{if } Y : \epsilon \triangleq q \notin S \text{ for all } Y.$$

Sol3-H: **Solve 3 for Hedges**

$$\{X : s \triangleq q\} \cup A;\ S;\ \sigma \Longrightarrow A;\ \{X : s \triangleq q\} \cup S;\ \sigma,$$

if $s \neq q$, $s \in \mathcal{V}_{\mathsf{H}}$ or $q \in \mathcal{V}_{\mathsf{H}}$, and $Y : s \triangleq q \notin S$ for all Y.

Sol-T: **Solve for Terms**

$$\{X : l \triangleq r\} \cup A;\ S;\ \sigma \Longrightarrow A;\ \{y : l \triangleq r\} \cup S;\ \sigma\{X \mapsto y\},$$

if $top(l) \neq top(r)$, $\chi : l \triangleq r \notin S$ for all χ, and y is fresh.

Rec: **Recover**

$$\{X : \tilde{s} \triangleq \tilde{q}\} \cup A;\ \{\chi : \tilde{s} \triangleq \tilde{q}\} \cup S;\ \sigma \Longrightarrow A;\ \{\chi : \tilde{s} \triangleq \tilde{q}\} \cup S;\ \sigma\{X \mapsto \chi\}.$$

The idea of the store is to keep track of already solved AUPs in order to generalize the same pair of hedges with the same variable, as it is illustrated in the Rec rule: The already solved AUP $\chi : \tilde{s} \triangleq \tilde{q}$ from the store helps to reuse χ instead of X as a generalization of $\tilde{s}$ and $\tilde{q}$. This is important, since we aim at computing lggs.

In the condition of Dec1-H we use a hedge variable U while in Dec2-H and Dec3-H in the same role χ appears. The reason is that in Dec1-H, the hedge $s, \tilde{s}$ or the hedge $q, \tilde{q}$ is not a term and, hence, we can not have a term variable in place of U. On the other hand, in Dec2-H and Dec3-H it can happen that the AUP in the condition is between terms with χ being a term variable.

Notice that there is no rule for AUPs of the form $X : x \triangleq x$. This is because we assume the hedges to be generalized are variable disjoint and, hence, such problems do not appear.

To compute generalizations for hedges $\tilde{s}$ and $\tilde{q}$, the procedure starts with $\{X : \tilde{s} \triangleq \tilde{q}\}; \emptyset; Id$ where X is a fresh hedge variable and applies the rules on each selected anti-unification equation in all possible ways. We denote this procedure by $\mathfrak{G}$. To show that the process terminates, we define a complexity measure of the triple $A; S; \sigma$ as a multiset $M(A) := \{size(\tilde{s} \triangleq \tilde{q}) + 1 \mid X : \tilde{s} \triangleq \tilde{q} \in A\}$. We order complexity measures by the multiset extension $>_m$ of the standard ordering on natural numbers. It is easy to check that the theorem below holds, which immediately implies termination:

▶ **Theorem 3.1.** *If* $A_1; S_1; \sigma_1 \Longrightarrow A_2; S_2; \sigma_2$ *in* $\mathfrak{G}$, *then* $M(A_1) >_m M(A_2)$.

Hence, starting from $\{X : \tilde{s} \triangleq \tilde{q}\}; \emptyset; Id$, each sequence of transformations by $\mathfrak{G}$ necessarily terminates with a triple of the form $\emptyset; S; \sigma$.

▶ **Theorem 3.2** (Soundness of $\mathfrak{G}$). *If* $\{X : \tilde{s} \triangleq \tilde{q}\}; \emptyset; Id \Longrightarrow^* \emptyset; S; \sigma$ *is a derivation in* $\mathfrak{G}$, *then* $X\sigma \preceq \tilde{s}$ *and* $X\sigma \preceq \tilde{q}$.

Proof. The theorem follows from the straightforward fact that if $X\sigma \preceq \tilde{s}$ and $X\sigma \preceq \tilde{q}$ and $\{X : \tilde{s} \triangleq \tilde{q}\} \cup A; S; \sigma \Longrightarrow A', S', \sigma'$ is a transformation step in $\mathfrak{G}$, then $X\sigma' \preceq \tilde{s}$ and $X\sigma' \preceq \tilde{q}$. ◀

If $\{X : \tilde{s} \triangleq \tilde{q}\}; \emptyset; Id \Longrightarrow^* \emptyset; S; \sigma$ is a derivation in $\mathfrak{G}$, then we say that

- σ is a *substitution computed by* $\mathfrak{G}$ *for* $X : \tilde{s} \triangleq \tilde{q}$*;*
- the restriction of σ on X, denoted by $\sigma|_X$, is a *anti-unifier of* $X : \tilde{s} \triangleq \tilde{q}$ *computed by* $\mathfrak{G}$*;*
- the hedge $X\sigma$ is a *generalization of* $\tilde{s}$ *and* $\tilde{q}$ *computed by* $\mathfrak{G}$*.*

The proof of completeness of the algorithm requires auxiliary definitions and lemmas. We start generalizing the notion of anti-unifier for sets of equations.

▶ **Definition 3.3.** A set of AUPs is a set $A = \{\chi_1 : \tilde{s}_1 \triangleq \tilde{q}_1, \ldots, \chi_n : \tilde{s}_n \triangleq \tilde{q}_n\}$, where each of the variables $\chi_1, \ldots, \chi_n$ does not occur more than once. We define the set of generalization variables $gvar(A) = \{\chi_1, \ldots, \chi_n\}$.

▶ **Definition 3.4.** A substitution σ is called an anti-unifier of a set of AUPs A, if $dom(\sigma) \subseteq gvar(A)$ and for each $(\chi : \tilde{s} \triangleq \tilde{q}) \in A$, $\chi\sigma$ is a generalization of both $\tilde{s}$ and $\tilde{q}$.

Similarly, least general anti-unifiers are also generalized for sets of AUPs.

▶ **Definition 3.5.** We say that a set of AUPs A is unsimplifiable if any anti-unifier of A is equal to Id modulo variable renaming.

Notice that if A is unsimplifiable then A cannot contain equations with pairs of terms with the same top symbol $x : f(\tilde{s}) \triangleq f(\tilde{q})$, equations between equal sequences $\chi : \tilde{s} \triangleq \tilde{s}$, equations between terms $X : f(\tilde{s}) \triangleq g(\tilde{q})$ where $X \in \mathcal{V}_\mathsf{H}$, nor pairs of identical equations $\chi : \tilde{s} \triangleq \tilde{q}, \chi' : \tilde{s} \triangleq \tilde{q}$.

▶ **Lemma 3.6.** *Let A be a set of AUPs satisfying $gvar(A) \subseteq \mathcal{V}_\mathsf{H}$. Let S be an unsimplifiable set of AUPs. Let ϑ be an anti-unifier of A. Then, there exists a sequence of transformations $A; S; Id \Longrightarrow^* \emptyset; S'; \sigma$ where $\vartheta \preceq_{gvar(A)} \sigma$.*

This lemma is crucial for showing completeness of $\mathfrak{G}$. Its proof is quite long and proceeds by structural induction on A and by detailed case analysis on the form of a selected AUP in transformations. The interested reader can find it in the technical report [22].

▶ **Theorem 3.7** (Completeness of $\mathfrak{G}$)**.** *Let ϑ be an anti-unifier of $X : \tilde{s} \triangleq \tilde{q}$. Then $\mathfrak{G}$ computes a substitution σ such that $X\vartheta \preceq X\sigma$.*

Proof. Immediate consequence of Lemma 3.6 with $A = \{X : \tilde{s} \triangleq \tilde{q}\}$ and $S = \emptyset$. ◀

Hence, collecting all the hedges $X\sigma$ such that $\{X : \tilde{s} \triangleq \tilde{q}\}; \emptyset; Id \Longrightarrow^* \emptyset; S; \sigma$, we obtain a finite complete set of generalizations of $\tilde{s}$ and $\tilde{q}$. In general, this set is not minimal. Even for such a simple input as $\{X : f(a) \triangleq f(b)\}; \emptyset; Id$, the algorithm $\mathfrak{G}$ produces five generalizations: two hedges Y_1, Y_2 and Z_1, Z_2 and three terms $f(U_1, U_2)$, $f(V_1, V_2)$, and $f(x)$. The last term is an instance of the other four generalizations.

Nevertheless, this redundancy is not trivially avoidable because rules allowing apparently useless alignments are needed for completeness:

Answer to Quiz 1. *Besides the "expected" lgg $f(a), f(X)$, the set $mcg(\tilde{s}, \tilde{q})$ for $\tilde{s} = f(a), f(a)$ and $\tilde{q} = f(a), f$ also contains two less obvious ones: $f(X, Y), f(X)$ and $f(X, Y), f(Y)$.*

We need a minimization step to keep only least general generalizations. Minimization involves a matchability test between two hedges. If two hedges $\tilde{s}$ and $\tilde{q}$ are in the set we are going to minimize, then we proceed as follows:

- If $\tilde{s} \simeq \tilde{q}$, then we delete one of them and keep the other (e.g., with the smaller size).
- If one of them is strictly more general than the other one, we delete the more general hedge and keep the more specific one.

For matchability, one could, in principle, use the hedge matching algorithm from [18], but there is a subtlety one should take into account: The hedges that are to be matched, in general, are not ground. Therefore, when trying to match, e.g., $\tilde{s} = X, X$ to $\tilde{q} = X, a$, we should rename X in $\tilde{q}$ into a new constant. Furthermore, we should introduce a restriction that no term variable matches such new constants. Thus, the matchability test should fail for the problems like $X, X \ll X, a$ and $x \ll X$.

Hence, combining $\mathfrak{G}$ with minimization, we can compute $mcg(\tilde{s}_1, \tilde{s}_2)$ for each $\tilde{s}_1$ and $\tilde{s}_2$.

▶ **Example 3.8.** For the terms $f(g(a, X), a, X, b)$ and $f(g(b), b)$, $\mathfrak{G}$ computes the *mcg*: $\{f(g(x, Y), Z, Y, b),\ f(g(x, Y), x, Y, Z),\ f(g(U, Y, Z), Y, Z, b),\ f(g(U, Y, Z), U, Y, b)\}$. These four lggs are selected from 169 generalizations computed in the first step of the algorithm.

The drawback of the algorithm $\mathfrak{G}$ is that it is highly nondeterministic. It computes $O(3^n)$ generalizations[2], where n is the size of the input. (For instance, for $f(a_1, a_2, a_3, a_4, a_5)$ and $f(b_1, b_2, b_3, b_4, b_5)$ it computes 11685 generalizations, most of them several times, until it selects a single one, e.g., $f(x_1, x_2, x_3, x_4, x_5)$, on the minimization step.) The minimization step involves NP-complete hedge matching algorithm (see [18, 21]) performed on the pairs of elements of the generalization set. Hence, this algorithm is only of theoretical interest and falls short of being practically useful. Our goal is to impose requirements on the set of generalizations such that, on the one hand, it is still "interesting", on the other hand, it can be computed faster in many cases. This leads us to the notion of rigid generalization, described in the next section.

4 Computing Rigid Generalizations

The main intuition behind rigid generalizations is to capture the structure (modulo a given rigidity property) of as many nonvariable terms in the input hedges as possible. It is parameterized by a binary *rigidity function* $\mathcal{R}$ that computes a finite set of *alignments* for strings, defined as follows:

▶ **Definition 4.1** (Alignment and Rigidity Function)**.** Let w_1 and w_2 be strings of symbols. Then the sequence $a_1[i_1, j_1] \cdots a_n[i_n, j_n]$, for $n \geq 0$, is an alignment if

- i's and j's are positive integers such that $i_1 < \cdots < i_n$ and $j_1 < \cdots < j_n$, and
- $a_k = w_1|_{i_k} = w_2|_{j_k}$, for all $1 \leq k \leq n$.

A rigidity function $\mathcal{R}$ is a function that returns, for every pair of strings of symbols w_1 and w_2, a set of alignments of w_1 and w_2.

▶ **Example 4.2.** We give some examples of rigidity functions. Here and below, instead of saying that the rigidity function $\mathcal{R}$ returns "the set of alignments of ...", we just say that it returns "the set of ...".

- $\mathcal{R}$ returns the set of all longest common subsequences of its arguments: $\mathcal{R}(abc, dd) = \{\epsilon\}$, $\mathcal{R}(abcda, bcad) = \{b[2, 1]c[3, 2]a[5, 3], b[2, 1]c[3, 2]d[4, 4]\}$.
- $\mathcal{R}$ returns the set of all those longest common subsequences whose length is at least 4: $\mathcal{R}(abcda, bca) = \emptyset$, $\mathcal{R}(abcda, bcacda) = \{a[1, 3]c[3, 4]d[4, 5]a[5, 6], b[2, 1]c[3, 4]d[4, 5]a[5, 6]\}$.

[2] Notice that the hedge decomposition rule has three non-deterministic choices.

- $\mathcal{R}$ returns the set of all longest common substrings of its arguments: $\mathcal{R}(abcda, bcad) = \{b[2,1]c[3,2]\}$, $\mathcal{R}(abcda,\ bcada) = \{b[2,1]c[3,2], d[4,4]a[5,5]\}$, $\mathcal{R}(abc, dd) = \{\epsilon\}$.

▶ **Definition 4.3** ($\mathcal{R}$-Generalization)**.** Given two (variable disjoint) hedges $\tilde{s}_1$ and $\tilde{s}_2$ and the rigidity function $\mathcal{R}$, we say that a hedge $\tilde{s}$ that generalizes both $\tilde{s}_1$ and $\tilde{s}_2$ is their *generalization with respect to* $\mathcal{R}$, or, in short, an $\mathcal{R}$-generalization, if either $\mathcal{R}(top(\tilde{s}_1), top(\tilde{s}_2)) = \emptyset$ and $\tilde{s}$ is a hedge variable, or there exists an alignment $f_1[i_1, j_1] \cdots f_n[i_n, j_n] \in \mathcal{R}(top(\tilde{s}_1), top(\tilde{s}_2))$, such that the following conditions are fulfilled:

1. The sequence $\tilde{s}$ does not contain pairs of consecutive hedge variables.
2. If we remove all hedge variables that occur as elements of $\tilde{s}$, we get a sequence of the form $f_1(\tilde{q}_1), \ldots, f_n(\tilde{q}_n)$.
3. For every $1 \leq k \leq n$, there exists a pair of sequences $\tilde{s}'_1$ and $\tilde{s}'_2$ such that $\tilde{s}_1|_{i_k} = f_k(\tilde{s}'_1)$, $\tilde{s}_2|_{j_k} = f_k(\tilde{s}'_2)$ and $\tilde{q}_k$ is an $\mathcal{R}$-generalization of $\tilde{s}'_1$ and $\tilde{s}'_2$.

Under this definition, $\mathcal{R}$-generalizations do not contain term variables. The minimal complete set of $\mathcal{R}$-generalizations of $\tilde{s}_1$ and $\tilde{s}_2$ is denoted by $mcg_{\mathcal{R}}(\tilde{s}_1, \tilde{s}_2)$. An *$\mathcal{R}$-anti-unifier* of $X : \tilde{s}_1 \triangleq \tilde{s}_2$ is a substitution σ such that $X\sigma$ is an $\mathcal{R}$-generalization of $\tilde{s}_1$ and $\tilde{s}_2$.

▶ **Example 4.4.** Let $\mathcal{R}(w_1, w_2)$ be the set of all longest common subsequences of w_1 and w_2.

- The terms $t_1 = f(g(a, X), a, X, b)$ and $t_2 = f(g(b), b)$ have a single least general $\mathcal{R}$-generalization $f(g(Y), Z, b)$. Note that this term does not belong to $mcg(t_1, t_2)$ computed in Example 3.8.
- $f(g(a, a), a, X, b)$ and $f(g(b, b), g(Y), b)$ have two $\mathcal{R}$-generalizations: $f(g(U), Z, b)$ and $f(V, g(U), Z, b)$. The first one is less general than the second one.
- The hedges a, b and b, c have a single $\mathcal{R}$-generalization: X, b, Y.

▶ **Example 4.5.** Let $\mathcal{R}(w_1, w_2)$ be the set of all longest common substrings of w_1 and w_2.

- The least general $\mathcal{R}$-generalization of $a, a, b, f, f, f(a, a, b)$ and $a, a, c, f, f, f(a, a, c)$ is the hedge $X, f, f, f(a, a, Y)$.
- The least general $\mathcal{R}$-generalization of $a, a, b, b, f, f, f(a, a, b, b)$ and $a, a, c, f, f, f(a, a, c)$ is the hedge $X, f, f, f(a, a, Y)$.

Quiz 2: *What is the $mcg_{\mathcal{R}}(\tilde{s}, \tilde{q})$ for two identical hedges $\tilde{s} = f(a, b, c), g(a), h(a)$ and $\tilde{q} = f(a, b, c), g(a), h(a)$, where $\mathcal{R}$ is a function that computes the set of all common subsequences of the minimal length 3 of its arguments?*

Our goal is to compute a minimal complete set of $\mathcal{R}$-generalizations. For this, we design a new set of transformation rules. It consists of only four rules shown below:

$\mathcal{R}$-Dec-H: **$\mathcal{R}$-Rigid Decomposition for Hedges**

$$\{X : \tilde{s} \triangleq \tilde{q}\} \cup A;\ S;\ \sigma \Longrightarrow$$
$$\{Z_k : \tilde{s}_k \triangleq \tilde{q}_k \mid 1 \leq k \leq n\} \cup A;$$
$$\{Y_0 : \tilde{s}|_0^{i_1} \triangleq \tilde{q}|_0^{j_1}\} \cup \{Y_k : \tilde{s}|_{i_k}^{i_{k+1}} \triangleq \tilde{q}|_{j_k}^{j_{k+1}} \mid 1 \leq k \leq n-1\} \cup \{Y_n : \tilde{s}|_{i_n}^{|\tilde{s}|+1} \triangleq \tilde{q}|_{j_n}^{|\tilde{q}|+1}\} \cup S;$$
$$\sigma\{X \mapsto Y_0, f_1(Z_1), Y_1, \ldots, Y_{n-1}, f_n(Z_n), Y_n\},$$

if $\mathcal{R}(top(\tilde{s}), top(\tilde{q}))$ contains a sequence $f_1[i_1, j_1] \cdots f_n[i_n, j_n]$ such that for all $1 \leq k \leq n$, $\tilde{s}|_{i_k} = f_k(\tilde{s}_k)$, $\tilde{q}|_{j_k} = f_k(\tilde{q}_k)$, and Y_0, Y_k's and Z_k's are fresh.

$\mathcal{R}$-S-H: **$\mathcal{R}$-Rigid Solve for Hedges**

$$\{X : \tilde{s} \triangleq \tilde{q}\} \cup A;\ S;\ \sigma \Longrightarrow A;\ \{X : \tilde{s} \triangleq \tilde{q}\} \cup S;\ \sigma,$$

if $\mathcal{R}(top(\tilde{s}), top(\tilde{q})) = \emptyset$. (Notice that this transformation is equivalent to rule $\mathcal{R}$-Dec-H where $\mathcal{R}(top(\tilde{s}), top(\tilde{q})) = \{\epsilon\}$).

$\mathcal{R}$-CS1: **$\mathcal{R}$-Rigid Clean Store 1**

$$A; \{X_1 : \tilde{s} \triangleq \tilde{q}, X_2 : \tilde{s} \triangleq \tilde{q}\} \cup S; \sigma \Longrightarrow A; \{X_1 : \tilde{s} \triangleq \tilde{q}\} \cup S; \sigma\{X_2 \mapsto X_1\}, \quad \text{if } X_1 \neq X_2.$$

$\mathcal{R}$-CS2: **$\mathcal{R}$-Rigid Clean Store 2**

$$A;\ \{X : \epsilon \triangleq \epsilon\} \cup S;\ \sigma \Longrightarrow A;\ S;\ \sigma\{X \mapsto \epsilon\}$$

To compute $\mathcal{R}$-generalizations of $\tilde{s}$ and $\tilde{q}$, we start with $\{X : \tilde{s} \triangleq \tilde{q}\}; \emptyset; Id$ and apply the rules on the selected anti-unification equation(s) in all possible ways. The obtained procedure is denoted by $\mathfrak{G}(\mathcal{R})$. To show that it terminates, we define the complexity measure for $A; S; \sigma$ as a pair $(M(A), M(S))$, where M is defined as in the termination proof of $\mathfrak{G}$. The measures are compared lexicographically. Each rule strictly reduces it, therefore there can be no infinite transformation chains. All the rules, except $\mathcal{R}$-Dec-H, transform the selected equation(s) uniquely. $\mathcal{R}$-Dec-H can introduce only finitely many branchings, because $\mathcal{R}$ returns a finite set. Hence, the following theorem holds:

▶ **Theorem 4.6.** *The procedure $\mathfrak{G}(\mathcal{R})$ terminates on any input and produces a system $\emptyset; S; \sigma$ where S is irreducible with respect to the store cleaning rules.*

The intuition behind the $\mathcal{R}$-Dec-H rule is that, once $\mathcal{R}$ gives the set of alignments of the strings $top(\tilde{s})$ and $top(\tilde{q})$, we choose one alignment from it, and rigid decomposition is not permitted to be performed on the equations formed by the remaining subsequences of $\tilde{s}$ and $\tilde{q}$ (i.e, the ones that are generalized by Y's in $\mathcal{R}$-Dec-H). Otherwise, the generalization might violate the restrictions of Definition 4.3. Therefore, we move these equations to the store where the decomposition and solve rules do not apply. However, it may introduce certain redundancies in the store. These redundancies are dealt with the store cleaning rules. Another interesting observation is that $\mathfrak{G}(\mathcal{R})$ never introduces in the set A or S equations of the form $x : l \triangleq r$ where x is a term variable.

Since we generalize variable disjoint hedges, the strings in $\mathcal{R}(top(\tilde{s}), top(\tilde{q}))$ (that are common subsequences of $top(\tilde{s})$ and $top(\tilde{q})$) do not contain variables. After application of the rule $\mathcal{R}$-Dec-H, each hedge variable in the anti-unifier gets separated from the other variables by a nonvariable term, to obey the restriction 1 of Definition 4.3.

We did not have the store cleaning rules in our previous algorithm $\mathfrak{G}$, because the AUPs they are dealing with would never appear in the store the rules in $\mathfrak{G}$ are operating on.

Proving soundness of $\mathfrak{G}(\mathcal{R})$ is quite involved, because we should show that the output of $\mathfrak{G}(\mathcal{R})$ satisfies properties of $\mathcal{R}$-generalizations. We need a couple of lemmas for that:

▶ **Lemma 4.7.** *Let $A; S; \vartheta \Longrightarrow_{\mathsf{R}_1} A_1; S_1; \vartheta\sigma_1 \Longrightarrow_{\mathsf{R}_2} A_2; S_2; \vartheta\sigma_1\sigma_2$ be a sequence of transformations where $\mathsf{R}_1 \in \{\mathcal{R}\text{-CS1}, \mathcal{R}\text{-CS2}\}$ and $\mathsf{R}_2 \in \{\mathcal{R}\text{-Dec-H}, \mathcal{R}\text{-S-H}\}$. Then there exists a transformation sequence $A; S; \vartheta \Longrightarrow_{\mathsf{R}_2} A'_1; S'_1; \vartheta\sigma_2 \Longrightarrow_{\mathsf{R}_1} A'_2; S'_2; \vartheta\sigma_2\sigma_1$ such that $A'_2 = A_2$, $S'_2 = S_2$, and $\vartheta\sigma_1\sigma_2 = \vartheta\sigma_2\sigma_1$.*

Proof. Since R_1 does not affect the first component in the system, we have $A_1 = A$ and $A'_2 = A'_1$. We perform the step R_2 in the second transformation sequence exactly in the same way as in the first one, choosing the same rule, the same AUP in A, the same alignment, and the same fresh variables. Then $A'_1 = A_2$ and, hence, $A'_2 = A_2$. As for the stores, S_2 consists of all the AUPs in S except those deleted by R_1 and R_2 and, in addition, it contains the AUPs introduced by R_2. In the second sequence, S'_1 consists of all the AUPs in S except the one deleted by R_2 and the ones introduced by R_2. In the last step, we delete from S'_1 exactly the same AUP that was deleted from S_1 by R_1. Therefore, we get $S'_2 = S_2$. Finally, σ_1 and σ_2 commute, because their domains and ranges are disjoint. Hence, $\vartheta\sigma_1\sigma_2 = \vartheta\sigma_2\sigma_1$. ◀

▶ **Lemma 4.8.** *If $A; S_1; \vartheta \Longrightarrow^* \emptyset; S_2; \vartheta\sigma$ is a derivation in $\mathfrak{G}(\mathcal{R})$ using only $\mathcal{R}$-Dec-H and $\mathcal{R}$-S-H, then for all $(X : \tilde{s} \triangleq \tilde{q}) \in A$, the hedge $X\sigma$ is an $\mathcal{R}$-generalization of $\tilde{s}$ and $\tilde{q}$.*

Proof. We proceed by induction on the length of the derivation. If it is 1, then the derivation has the form $\{X : \tilde{s} \triangleq \tilde{q}\}; S; \vartheta \Longrightarrow^* \emptyset; \{X : \tilde{s} \triangleq \tilde{q}\} \cup S; \vartheta\sigma$, where $\sigma = \{X \mapsto Y_0\}$ for a fresh Y_0 if the used rule is $\mathcal{R}$-Dec-H, and $\sigma = Id$ if the used rule is $\mathcal{R}$-S-H. Since $\mathcal{R}(top(\tilde{s}), top(\tilde{q})) \subseteq \{\epsilon\}$, $X\sigma$ is an $\mathcal{R}$-generalization of $\tilde{s}$ and $\tilde{q}$.

Now we assume that the lemma holds for all derivations with the length less than $m > 1$ and prove it for m. Let the system to be transformed be $\{X : \tilde{s} \triangleq \tilde{q}\} \cup A'; S; \vartheta$. If it is transformed by the rule $\mathcal{R}$-S-H then $\mathcal{R}(top(\tilde{s}), top(\tilde{q})) = \emptyset$, $\sigma = Id$, and we obtain a new system $A'; \{X : \tilde{s} \triangleq \tilde{q}\} \cup S; \vartheta$. By the induction hypothesis, $X'\sigma$ is an $\mathcal{R}$-generalization of $\tilde{s}'$ and $\tilde{q}'$ for all $(X' : \tilde{s}' \triangleq \tilde{q}') \in A'$. By the definition of $\mathcal{R}$-generalization, the same holds for $X\sigma$, $\tilde{s}$, and $\tilde{q}$ because $\mathcal{R}(top(\tilde{s}), top(\tilde{q})) = \emptyset$.

If the rule $\mathcal{R}$-Dec-H is used to transform $\{X : \tilde{s} \triangleq \tilde{q}\} \cup A'; S; \vartheta$, then the new system is $\{Z_k : \tilde{s}_k \triangleq \tilde{q}_k \mid 1 \leq k \leq n\} \cup A'; S'; \vartheta\sigma'$, where $\sigma' = \{X \mapsto Y_0, f_1(Z_1), Y_1, \ldots, Y_{n-1}, f_n(Z_n), Y_n\}$ and the conditions of $\mathcal{R}$-Dec-H are satisfied. By the induction hypothesis, We have a derivation $\{Z_k : \tilde{s}_k \triangleq \tilde{q}_k \mid 1 \leq k \leq n\} \cup A'; S'; \vartheta\sigma' \Longrightarrow^* \emptyset; S''; \vartheta\sigma'\sigma''$ using only the rules $\mathcal{R}$-Dec-H and $\mathcal{R}$-S-H such that for all $(X' : \tilde{s}' \triangleq \tilde{q}') \in \{Z_k : \tilde{s}_k \triangleq \tilde{q}_k \mid 1 \leq k \leq n\} \cup A'$, the hedge $X'\sigma''$ is an $\mathcal{R}$-generalization of $\tilde{s}'$ and $\tilde{q}'$. In particular, this holds for Z's. Therefore, $X\sigma'\sigma''$ is an $\mathcal{R}$-generalization of $\tilde{s}$ and $\tilde{q}$. This finishes the proof. ◀

▶ **Lemma 4.9.** *If $\{X : \tilde{s}_1 \triangleq \tilde{s}_2\}; \emptyset; Id \Longrightarrow^* \emptyset; S_1; \vartheta \Longrightarrow_{\mathsf{R}} \emptyset; S_2; \vartheta\sigma$ is a derivation in $\mathfrak{G}(\mathcal{R})$ such that $X\vartheta$ is an $\mathcal{R}$-generalization of $\tilde{s}_1$ and $\tilde{s}_2$ and $\mathsf{R} \in \{\mathcal{R}\text{-CS1}, \mathcal{R}\text{-CS2}\}$. Then $X\vartheta\sigma$ is an $\mathcal{R}$-generalization of $\tilde{s}_1$ and $\tilde{s}_2$.*

Proof. Let R be $\mathcal{R}$-CS1, transforming $\{X_1 : \tilde{s}'_1 \triangleq \tilde{s}'_2, X_2 : \tilde{s}'_1 \triangleq \tilde{s}'_2\} \subseteq S_1$ into $\{X_1 : \tilde{s}'_1 \triangleq \tilde{s}'_2\} \subseteq S_2$ with the substitution $\sigma = \{X_2 \mapsto X_1\}$. The hedges $\tilde{s}'_1$ and $\tilde{s}'_2$ occur in $\tilde{s}_1$ and $\tilde{s}_2$, respectively, so that the corresponding occurrences are abstracted by the same variable in $X\vartheta$. This variable for some pairs of occurrences of $\tilde{s}'_1$ and $\tilde{s}'_2$ is X_1 and for some others X_2. Hence, if we replace X_2 with X_1 in $X\vartheta$, the obtained hedge $X\vartheta\sigma$ will be a generalization of $\tilde{s}_1$ and $\tilde{s}_2$.

To prove that after this replacement we still have an $\mathcal{R}$-generalization of $\tilde{s}_1$ and $\tilde{s}_2$, we proceed by induction on the maximal depth d of the occurrences of X_2 in $X\vartheta$. It is enough to show that replacing only one occurrence of X_2 with X_1 retains the $\mathcal{R}$-generalization property.

Let first $\mathsf{d} = 0$. Then $X\vartheta$ has a form $\tilde{q}_1, X_2, \tilde{q}_2$. Replacing X_2 with X_1 gives $\tilde{q}_1, X_1, \tilde{q}_2$, that keeps the same alignment from $\mathcal{R}(top(\tilde{s}_1), top(\tilde{s}_2))$ that was in $X\vartheta$ and satisfies all three conditions of the definition of $\mathcal{R}$-generalization. Hence, $\tilde{q}_1, X_1, \tilde{q}_2$ is an $\mathcal{R}$-generalization of $\tilde{s}_1$ and $\tilde{s}_2$.

Now assume that $\mathsf{d} > 0$. It means that there exists a term $f(\tilde{q})$ in $X\vartheta$, such that X_2 occurs at depth $\mathsf{d} - 1$ in $\tilde{q}$. Then there are terms $f(\tilde{s}''_1)$ in $\tilde{s}_1$ and $f(\tilde{s}''_2)$ in $\tilde{s}_2$ such that $\tilde{q}$ is an $\mathcal{R}$-generalization of $\tilde{s}''_1$ and $\tilde{s}''_2$. By the induction hypothesis, replacing an occurrence of X_2 in $\tilde{q}$ with X_1 gives a hedge that is again an $\mathcal{R}$-generalization of $\tilde{s}''_1$ and $\tilde{s}''_2$. Hence, the hedge obtained from $X\vartheta$ by replacing one occurrence of X_2 with X_1 is an $\mathcal{R}$-generalization of $\tilde{s}_1$ and $\tilde{s}_2$, because we just showed that the third condition of the definition of $\mathcal{R}$-generalization is satisfied, while the other two conditions were not affected.

Repeating the process of replacement of one occurrence of X_2 by X_1 iteratively until there are no more X_2's in $X\vartheta$, we prove that $X\vartheta\sigma$ is an $\mathcal{R}$-generalization of $\tilde{s}_1$ and $\tilde{s}_2$.

Proof for $\mathsf{R} = \mathcal{R}$-CS2 is straightforward. ◀

Now we can prove the soundness theorem for $\mathfrak{G}(\mathcal{R})$:

▶ **Theorem 4.10** (Soundness of $\mathfrak{G}(\mathcal{R})$). *If* $\{X : \tilde{s}_1 \triangleq \tilde{s}_2\}; \emptyset; Id \Longrightarrow^* \emptyset; S; \sigma$ *is a derivation in* $\mathfrak{G}(\mathcal{R})$*, then* $X\sigma$ *is an* $\mathcal{R}$*-generalization of* $\tilde{s}_1$ *and* $\tilde{s}_2$*.*

Proof. By Lemma 4.7, every derivation in $\mathfrak{G}(\mathcal{R})$ can be reordered so that first only the rules $\mathcal{R}$-Dec-H and $\mathcal{R}$-S-H are applied until the set of AUPs becomes empty, and then the store is cleaned. The substitutions computed by the original derivation and by the reordered derivation coincide. Let σ' be the substitution obtained at the end of the subderivation with $\mathcal{R}$-Dec-H and $\mathcal{R}$-S-H. By Lemma 4.8, $X\sigma'$ is an $\mathcal{R}$-generalization of $\tilde{s}_1$ and $\tilde{s}_2$. By Lemma 4.9, substitutions introduced by the store cleaning rules keep the $\mathcal{R}$-generalization property. Hence, $X\sigma$ is an $\mathcal{R}$-generalization of $\tilde{s}_1$ and $\tilde{s}_2$. ◀

The algorithm $\mathfrak{G}(\mathcal{R})$ is complete, as the following theorem shows.

▶ **Theorem 4.11** (Completeness of $\mathfrak{G}(\mathcal{R})$). *Let* $\tilde{q}$ *be an* $\mathcal{R}$*-generalization of* $\tilde{s}_1$ *and* $\tilde{s}_2$*. Then* $\mathfrak{G}(\mathcal{R})$ *computes an* $\mathcal{R}$*-anti-unifier* σ *for* $X : \tilde{s}_1 \triangleq \tilde{s}_2$ *such that* $\tilde{q} \preceq X\sigma$*.*

The proof of this theorem is rather long, proceeding by induction on the size of $\tilde{q}$ and by case analysis on its form. It can be found in the technical report [22].

We may prune the search space of the algorithm $\mathfrak{G}(\mathcal{R})$, giving priority to the rules $\mathcal{R}$-CS1 and $\mathcal{R}$-CS2. If they are applicable to a system, no other rule should apply to it. It can prevent re-computing equivalent $\mathcal{R}$-generalizations on different branches without violating completeness. In addition, we may forbid the rule $\mathcal{R}$-Dec-H to add to the set A the AUPs of the form $Z_k : \epsilon \triangleq \epsilon$ for $1 \leq k \leq n$, and to the set S the AUPs of the form $Y_m : \epsilon \triangleq \epsilon$ for $0 \leq m \leq n$. Respectively, such Z_k's and Y_m's are replaced by ϵ in the substitution computed by $\mathcal{R}$-Dec-H. These simplifications can be justified by the fact that those AUPs, anyway, eventually will be eliminated by the $\mathcal{R}$-CS2 rule. Therefore, they do not affect completeness. In the examples below we assume $\mathfrak{G}(\mathcal{R})$ to be optimized in such ways. The length of each derivation under the optimized $\mathfrak{G}(\mathcal{R})$ does not exceed the size of the input problem.

To compute minimal complete set of $\mathcal{R}$-generalizations, we still need to perform the minimization step, unless the cardinality of the set that $\mathcal{R}$ computes is not greater than 1. In the latter case the $\mathfrak{G}(\mathcal{R})$ computes a single $\mathcal{R}$-generalization of the input hedges.

Hence, combining $\mathfrak{G}(\mathcal{R})$ with the minimization step, we can compute $mcg_{\mathcal{R}}(\tilde{s}_1, \tilde{s}_2)$ for any hedges $\tilde{s}_1, \tilde{s}_2$, and the rigidity function $\mathcal{R}$.

▶ **Example 4.12.** Given two terms $f(g(a,a), a, X, b)$ and $f(g(b,b), g(Y), b)$, and $\mathcal{R}$ being the function computing the set of all longest common subsequences, the algorithm $\mathfrak{G}(\mathcal{R})$ gives two $\mathcal{R}$-generalizations: $f(V, g(U), Z, b)$ and $f(g(U), Z, b)$. After the minimization step, only the last one is retained. We illustrate how $\mathfrak{G}(\mathcal{R})$ computes $f(g(U), Z, b)$:

$$\{X_0 : f(g(a,a), a, X, b) \triangleq f(g(b,b), g(Y), b)\}; \emptyset; Id \Longrightarrow_{\mathcal{R}\text{-Dec-H}}$$
$$\{X_1 : g(a,a), a, X, b \triangleq g(b,b), g(Y), b\}; \emptyset; \{X_0 \mapsto f(X_1)\}.$$

This problem is transformed by the $\mathcal{R}$-Dec-H rule with the alignment $g[1,1]b[4,3]$:

$$\{U : a, a \triangleq b, b,\ U' : \epsilon \triangleq \epsilon\}; \{Z : a, X \triangleq g(Y)\}; \{X_0 \mapsto f(g(U), Z, b(U')), \ldots\} \Longrightarrow_{\mathcal{R}\text{-S-H}}$$
$$\{U' : \epsilon \triangleq \epsilon\}; \{U : a, a \triangleq b, b,\ Z : a, X \triangleq g(Y)\}; \{X_0 \mapsto f(g(U), Z, b(U')), \ldots\} \Longrightarrow_{\mathcal{R}\text{-Dec-H}}$$
$$\emptyset; \{U : a, a \triangleq b, b,\ Z : a, X \triangleq g(Y)\}; \{X_0 \mapsto f(g(U), Z, b), \ldots\}.$$

From the final state one can get not only the $\mathcal{R}$-anti-unifier $\{X_0 \mapsto f(g(U), Z, b)\}$ and the corresponding $\mathcal{R}$-generalization $f(g(U), Z, b)$, but also the substitutions that show how the original terms are obtained from the $\mathcal{R}$-generalization. These substitutions can be extracted

from the store: $\sigma_1 = \{U \mapsto a, a,\ Z \mapsto a, X\}$ with $f(g(U), Z, b)\sigma_1 = f(g(a, a), a, X, b)$ and $\sigma_2 = \{U \mapsto b, b,\ Z \mapsto g(Y)\}$ with $f(g(U), Z, b)\sigma_2 = f(g(b, b), g(Y), b)$. In this way, we can also say that the store gives us the difference of the input terms.

▶ **Example 4.13.** Let $\mathcal{R}$ compute the set of all longest common substrings of its arguments and let $a, a, b, f, f, f(a, a, b)$ and $a, a, c, f, f, f(a, a, c)$ be the input hedges. Their only $\mathcal{R}$-generalization $X, f, f, f(a, a, Y)$ can be computed by $\mathfrak{G}(\mathcal{R})$ performing the following steps:

$$\{X_0 : a, a, b, f, f, f(a, a, b) \triangleq a, a, c, f, f, f(a, a, c)\}; \emptyset; Id \Longrightarrow_{\mathcal{R}\text{-Dec-H}}$$
$$\{Y' : a, a, b \triangleq a, a, c\}; \{X : a, a, b \triangleq a, a, c\}; \{X_0 \mapsto X, f, f, f(Y')\} \Longrightarrow_{\mathcal{R}\text{-Dec-H}}$$
$$\emptyset; \{X : a, a, b \triangleq a, a, c,\ Y : b \triangleq c\}; \{X_0 \mapsto X, f, f, f(a, a, Y), \ldots\}.$$

Answer to the Quiz 2: *Given two identical hedges $\tilde{s} = f(a, b, c), g(a), h(a)$ and $\tilde{q} = f(a, b, c), g(a), h(a)$ and $\mathcal{R}$ computing the set of all common subsequences of the minimal length 3 of its arguments, $mcg_{\mathcal{R}}(\tilde{s}, \tilde{q}) = \{f(a, b, c), g(X), h(X)\}$. One might expect the lgg to be the hedge $f(a, b, c), g(a), h(a)$ itself, but it violates the condition 3 of Definition 4.3.*

The example in the Quiz 2 makes it clear why among the $\mathcal{R}$-generalization rules, we do not have the one that would generalize two identical terms with the same term (the so called Trivial Terms rule). It would simply make the $\mathfrak{G}(\mathcal{R})$ algorithm unsound.

Our approach generalizes existing works on word anti-unification. To extend the word anti-unification algorithm from [10] to hedges, one can just take as $\mathcal{R}$ the function that generates the singleton set consisting of the maximal variable-free subsequence in the unique generalization of two words computed in [10]. Similarly, ϵ-free anti-unification for words [6] can be extended to hedges by taking $\mathcal{R}$ as the function that computes the set of all maximal variable-free subsequences of ϵ-free generalizations of the input words.

Precision of rigid anti-unification can be improved, permitting term variables to occur in rigid generalizations. The idea is to generalize AUPs between two term sequences of equal length not with a single hedge variable, but with a sequence of term variables of the same length. This refinement would give $f(x_1, x_2, x_3, x_4, x_5)$ (instead of $f(X)$) as a generalization of $f(a_1, a_2, a_3, a_4, a_5)$ and $f(b_1, b_2, b_3, b_4, b_5)$. This can be achieved by a relatively little computational overhead compared to the $\mathfrak{G}(\mathcal{R})$ algorithm. The details can be found in [22]. Here we only remark that the standard anti-unification over ranked terms [27, 28] can be modeled by such a refinement of $\mathcal{R}$-generalization, choosing $\mathcal{R}$ as the function that returns a singleton set $\mathcal{R}(w_1, w_2) = \{a_1[i_1, i_1] \cdots a_n[i_n, i_n]\}$, where $a_1 \cdots a_n$ is the longest common subsequence of w_1 and w_2 such that all a_is occur at the same positions in w_1 and w_2.

5 Application in Clone Detection

In this section we outline a possible application of $\mathcal{R}$-generalization in software code clone detection. Clone detection is an active research topic since clones are considered to be a significantly problematic issue for software maintenance. Studies show that from 5 to 20% of software systems can contain duplicated code. Due to various complications such duplicated pieces cause, it is widely agreed and the clones should be detected. The survey papers [29, 30] give a detailed characterization of code duplication reasons and drawbacks, introduce clone types, describe and evaluate clone detection process and techniques, and list open problems in clone detection research. The proposed classification distinguishes four types of clones:

Type I: Identical code fragments except for variations in whitespace, layout and comments.

Type II: Syntactically identical fragments except for variations in identifiers, literals, types, whitespace, layout and comments.

Type III: Copied fragments with further modifications such as changed, added or removed statements, in addition to variations in identifiers, literals, types, whitespace, layout and comments.

Type IV: Two or more code fragments that perform the same computation but are implemented by different syntactic variants.

Complexity and sophistication in detecting such clones increases from Type I through Type IV with Type IV being the highest. (Although it does not mean that Type IV contains other types as special cases.) $\mathcal{R}$-generalizations can help in detecting clones of types I-III. We illustrate the idea on an example of a clone of type III.

▶ **Example 5.1.** Type III clones from [29]:

```
if (a >= b)  {                    if (m >= n)
    c = d + b; // Comment1           { // Comment1'
    d = d + 1; }                       y = x + n;
else                                   z = 1; // Added statement
    c = d - a; //Comment2              x = x + 5; //Comment3 }
                                  else
                                     y = x - m; //Comment2'
```

Some clone detection techniques are based on tree representation of the code, like parse trees, abstract syntax trees, or an XML form of abstract syntax trees; see, e.g., [5, 13, 16, 34, 32], for some of the works that follow this approach. We assume that the code is represented in a structural form that can be encoded with unranked terms (or hedges). We keep the representation abstract, without specifying what exactly this structural form is.

Usually, clone detection tools first preprocess the code, then find potential clone candidates, and, finally, analyze them to detect actual clones. One can employ the $\mathcal{R}$-generalization algorithm in the process of finding potential code clones. Further analysis can be based on various measures, like, e.g., on the size of the generalization, or on the maximal length of a nonvariable hedge in the generalization, etc. Although we are not concerned with this part, by choosing appropriate $\mathcal{R}$s we can anticipate this last filtering process. The choice of the $\mathcal{R}$ depends on what is considered as interesting clone.

▶ **Example 5.2.** Unranked term form for the pieces of code in Example 5.1:

$$if(\geq(a,b), then(=(c,+(d,b)),=(d,+(d,1))), else(=(c,-(d,a))))$$
$$if(\geq(m,n), then(=(y,+(x,n)),=(z,1),=(x,+(x,5))), else(=(y,-(x,m))))$$

Let $\mathcal{R}$ be the relation of longest common subsequence. We choose it to capture the idea that the clones have a lot in common. Such an $\mathcal{R}$ is supposed to draw out from two pieces of code as much common statements as possible. Then (the refinement with term variables for) $\mathcal{R}$-generalization of these terms returns three generalizations as clone candidates:

$$if(\geq(x_1,x_2), then(X,=(x_3,x_4),=(x_5,+(x_5,x_6))), else(x_7,-(x_5,x_1))),$$
$$if(\geq(y_1,y_2), then(=(y_3,+(y_4,y_2)),Y,=(y_4,+(y_4,y_5))), else(y_3,-(y_4,y_1))),$$
$$if(\geq(z_1,z_2), then(=(z_3,+(z_4,z_2)),=(z_5,z_6),Z), else(z_3,-(z_4,z_1))).$$

Among them, we say that the second one is the best generalization of the clone pieces, because it preserves the common structure better that the other two (has a bigger size compared to them).

6 Discussion

The standard anti-unification [27, 28] has already been considered for computing software clones in [8, 7], detecting mostly clones of types I and II. However, we think that parameterized anti-unification over unranked terms offers more flexibility in finding clone candidates. First of all, it helps to detect inserted or deleted pieces of code, which is necessary for clones of type III. Besides, if we are interested in clones whose length (as a sequence of program statements) is greater than a predefined threshold, we can include this measure in the definition of the relation $\mathcal{R}$, considering only sequences that are longer than the threshold number. Another advantage of this approach is that it is modular, where most of the computations are performed on strings. It may combine advantages of fast textual and precise structural techniques. For many interesting string relations (e.g., longest common subsequence, longest common substring, sequence alignment, etc.), there exist efficient algorithms that also scale well for large data [14]. Hence, one can take advantage using these off-the-shelf methods when computing clone candidates by $\mathcal{R}$-generalization.

Yet another advantage of using $\mathcal{R}$-generalizations in clone detection is that it works on unranked terms that are natural abstractions of XML documents. How to detect clones well in generated XML/HTML is mentioned as one of the open problems in clone detection research in [29]. A detection technique that uses $\mathcal{R}$-generalization would be an interesting approach to this problem.

Moreover, from the clones computed by $\mathcal{R}$-generalization (anti-unification, in general) one can extract a procedure. This process has a use in code refactoring. The clones can be replaced by the procedure calls, properly instantiated by the substitution that gives from the computed $\mathcal{R}$-generalization the clone it generalizes. As we saw in Example 4.12, these substitutions are easily extracted from the store. In general, while anti-unifiers reflect similarities between two inputs, the data in the store can be used to identify differences between them (i.e., between inputs). This provides for unranked trees a functionality similar, for instance, to one of the well-known comparison utilities (e.g., diff, cmp, fc) that compare the contents of files, finding common contents and differences in them.

The emphasis of this paper is not, however, on clone detection by anti-unification. It can be a topic of separate research where (a) one shows that the $\mathcal{R}$-generalization approach can cover a wide range of currently existing techniques to find similarities between different pieces of code, and (b) presents a clone detection method (and its implementation) fully, starting from code preprocessing till returning the actual interesting clones, where $\mathcal{R}$-generalization performs the task of selecting clone candidates. In this paper we presented the $\mathcal{R}$-anti-unification itself from the theoretical point of view and just tried to indicate some possibilities of its application in clone detection.

We proved properties of $\mathcal{R}$-generalization for a generic $\mathcal{R}$, i.e., for the entire class of rigidity functions. Specializing $\mathcal{R}$ with a particular function, we obtain a specific instance of $\mathcal{R}$-generalization. We saw how certain known generalization problems fall into the class of specific instances of $\mathcal{R}$-generalization in this way.

$\mathcal{R}$-generalization can be made more precise by permitting to generalize term sequences of equal length with a sequence of term variables of the same length, instead of abstracting the original sequences by a single hedge variable. One could think of another extension, to allow a kind of recursive rigid generalization, extending the scope of rigid decomposition rule to the hedges that we currently move to the store, whenever possible. It would require an appropriate revision of the definition of rigid anti-unification.

7 Final Comments

We have presented anti-unification algorithms for unranked terms and hedges, starting from a minimal complete one and then designing a more efficient and flexible version for computing only rigid anti-unifiers. We indicated possible applications of this technique in software code clone detection.

There are a couple of possible directions in future work. One option is to bring in certain higher-order features that can help to further improve the precision of rigid generalizations. An example of such a higher-order extension would be the introduction of function variables. With their help, the algorithm can compute generalizations of the arguments of terms whose heads are distinct. Other interesting directions would be to perform unranked anti-unification in a sorted setting or on compressed terms.

Acknowledgments

This research has been partially supported by the CICyT projects SuRoS (ref. TIN2008-04547/TIN) and TASSAT (ref. TIN2010-20967-C04-01) and by the EC FP6 for Integrated Infrastructures Initiatives under the project SCIEnce (contract No. 026133). The authors thank the anonymous referees for helpful comments.

References

1 M. Alpuente, S. Escobar, J. Meseguer, and P. Ojeda. A modular equational generalization algorithm. In M. Hanus, editor, *LOPSTR*, volume 5438 of *Lecture Notes in Computer Science*, pages 24–39. Springer, 2008.

2 M. Alpuente, S. Escobar, J. Meseguer, and P. Ojeda. Order-sorted generalization. *Electr. Notes Theor. Comput. Sci.*, 246:27–38, 2009.

3 E. Armengol and E. Plaza. Bottom-up induction of feature terms. *Machine Learning*, 41(3):259–294, 2000.

4 F. Baader. Unification, weak unification, upper bound, lower bound, and generalization problems. In R. V. Book, editor, *RTA*, volume 488 of *Lecture Notes in Computer Science*, pages 86–97. Springer, 1991.

5 I. D. Baxter, A. Yahin, L. M. de Moura, M. Sant'Anna, and L. Bier. Clone detection using abstract syntax trees. In *ICSM*, pages 368–377, 1998.

6 A. Biere. Normalisation, unification and generalisation in free monoids. Master's thesis, University of Karlsruhe, 1993. (in German).

7 P. Bulychev and M. Minea. An evaluation of duplicate code detection using anti-unification. In *Proc. 3rd International Workshop on Software Clones*, 2009.

8 P. E. Bulychev, E. V. Kostylev, and V. A. Zakharov. Anti-unification algorithms and their applications in program analysis. In A. Pnueli, I. Virbitskaite, and A. Voronkov, editors, *Ershov Memorial Conference*, volume 5947 of *Lecture Notes in Computer Science*, pages 413–423. Springer, 2009.

9 J. Burghardt. E-generalization using grammars. *Artif. Intell.*, 165(1):1–35, 2005.

10 I. Cicekli and N. K. Cicekli. Generalizing predicates with string arguments. *Appl. Intell.*, 25(1):23–36, 2006.

11 H. Cirstea, C. Kirchner, R. Kopetz, and P.-E. Moreau. Anti-patterns for rule-based languages. *J. Symb. Comput.*, 45(5):523–550, 2010.

12 A. L. Delcher and S. Kasif. Efficient parallel term matching and anti-unification. *J. Autom. Reasoning*, 9(3):391–406, 1992.

13 W. S. Evans, C. W. Fraser, and F. Ma. Clone detection via structural abstraction. *Software Quality Journal*, 17(4):309–330, 2009.

14 D. Gusfield. *Algorithms on Strings, Trees, and Sequences - Computer Science and Computational Biology*. Cambridge University Press, 1997.

15 G. Huet. *Résolution d'équations dans des langages d'ordre* $1, 2, \ldots, \omega$. PhD thesis, Université Paris VII, September 1976.

16 R. Koschke, R. Falke, and P. Frenzel. Clone detection using abstract syntax suffix trees. In *WCRE*, pages 253–262. IEEE Computer Society, 2006.

17 U. Krumnack, A. Schwering, H. Gust, and K.-U. Kühnberger. Restricted higher-order anti-unification for analogy making. In M. A. Orgun and J. Thornton, editors, *Australian Conference on Artificial Intelligence*, volume 4830 of *Lecture Notes in Computer Science*, pages 273–282. Springer, 2007.

18 T. Kutsia. Solving equations with sequence variables and sequence functions. *J. Symb. Comput.*, 42(3):352–388, 2007.

19 T. Kutsia. Flat matching. *J. Symb. Comput.*, 43(12):858–873, 2008.

20 T. Kutsia, J. Levy, and M. Villaret. Sequence unification through currying. In F. Baader, editor, *RTA*, volume 4533 of *Lecture Notes in Computer Science*, pages 288–302. Springer, 2007.

21 T. Kutsia, J. Levy, and M. Villaret. On the relation between context and sequence unification. *J. Symb. Comput.*, 45(1):74–95, 2010.

22 T. Kutsia, J. Levy, and M. Villaret. Anti-Unification for Unranked Terms and Hedges. Technical Report 11-03, RISC Report Series, University of Linz, Austria, 2011.

23 T. Kutsia and M. Marin. Matching with regular constraints. In G. Sutcliffe and A. Voronkov, editors, *LPAR*, volume 3835 of *Lecture Notes in Computer Science*, pages 215–229. Springer, 2005.

24 T. Kutsia and M. Marin. Order-sorted unification with regular expression sorts. In C. Lynch, editor, *RTA*, volume 6 of *LIPIcs*, pages 193–208. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2010.

25 J. Lu, J. Mylopoulos, M. Harao, and M. Hagiya. Higher order generalization and its application in program verification. *Ann. Math. Artif. Intell.*, 28(1-4):107–126, 2000.

26 F. Pfenning. Unification and anti-unification in the calculus of constructions. In *LICS*, pages 74–85. IEEE Computer Society, 1991.

27 G. D. Plotkin. A note on inductive generalization. *Machine Intelligence*, 5(1):153–163, 1970.

28 J. C. Reynolds. Transformational systems and the algebraic structure of atomic formulas. *Machine Intelligence*, 5(1):135–151, 1970.

29 C. K. Roy and J. R. Cordy. A survey of software clone detection research. Technical report, School of Computing, Queen's University at Kingston, Ontario, Canada, 2007.

30 C. K. Roy, J. R. Cordy, and R. Koschke. Comparison and evaluation of code clone detection techniques and tools: A qualitative approach. *Sci. Comput. Program.*, 74(7):470–495, 2009.

31 U. Schmid. *Inductive Synthesis of Functional Programs, Universal Planning, Folding of Finite Programs, and Schema Abstraction by Analogical Reasoning*, volume 2654 of *Lecture Notes in Computer Science*. Springer, 2003.

32 V. Wahler, D. Seipel, J. W. von Gudenberg, and G. Fischer. Clone detection in source code by frequent itemset techniques. In *SCAM*, pages 128–135. IEEE Computer Society, 2004.

33 A. Yamamoto, K. Ito, A. Ishino, and H. Arimura. Modelling semi-structured documents with hedges for deduction and induction. In C. Rouveirol and M. Sebag, editors, *ILP*, volume 2157 of *Lecture Notes in Computer Science*, pages 240–247. Springer, 2001.

34 W. Yang. Identifying syntactic differences between two programs. *Softw., Pract. Exper.*, 21(7):739–755, 1991.