Matching and Generalization Modulo Proximity and Tolerance Relations[⋆]

Temur Kutsia and Cleo Pau

RISC, Johannes Kepler University, Linz, Austria
{kutsia,ipau}@risc.jku.at

Abstract. Proximity relations are fuzzy binary relations satisfying reflexivity and symmetry properties. Tolerance, which is a reflexive and symmetric (and not necessarily transitive) relation, can be also seen as a crisp version of proximity. We discuss two fundamental symbolic computation problems for proximity and tolerance relations: matching and anti-unification, present algorithms for solving them, and study properties of those algorithms.

Keywords: Fuzzy proximity relations · Matching · Anti-unification.

1 Introduction

Proximity relations are reflexive and symmetric fuzzy binary relations. They generalize similarity relations, which are a fuzzy version of equivalences. Proximity relations help to represent fuzzy information in situations where similarity is not adequate.

The crisp counterpart of proximity is tolerance, which generalizes the standard equivalence relation by dropping the transitivity property. In the literature, tolerance appears under other names as well, e.g., compatibility, similarity, or proximity relation. The term 'tolerance relation' is attributed to Zeeman [17].

A tolerance relation can be expressed as an undirected graph. The vertices of the graph form the set on which the relation is defined, and two elements are related if and only if there is an edge in the graph connecting them. A similar graph but with weighted edges can be associated to a proximity relation. This graph-based view helps to easily explain two important notions related to proximity and tolerance relations: proximity/tolerance blocks and proximity/tolerance classes (of a node). Blocks correspond to maximal cliques in the graph and the class of a node corresponds to its set of adjacent nodes, together with the node itself (see, e.g., [5, 8]).

Unification and anti-unification are two fundamental operations for many areas of symbolic computation. Unification aims at computing a most specific common instance of given logical expressions, while anti-unification, a technique

⋆ This work was supported by the Austrian Science Fund (FWF) under project 28789-N32 and by the strategic program "Innovatives OÖ 2020" by the Upper Austrian Government.

dual to unification, computes their least general generalization. Both techniques have been studied for equivalence relations both in crisp and fuzzy settings. Syntactic and equational unification is surveyed, e.g., in [4], for syntactic and equational anti-unification see, e.g., [2,6,14,15]. Unification and anti-unification modulo similarity have been investigated, e.g., in [1,16].

On the other hand, there are very few works on unification and anti-unification modulo proximity and tolerance. In [8], the authors introduced the notion of proximity-based unification (improved later in [9]) and used it in fuzzy logic programming. It can be characterized as a block-based approach, because two terms are treated as approximate in one computation when they have the same set of positions, symbols in their corresponding positions belong to the same block, and a certain symbol is always assigned to the same block. This approach imposes the restriction that the same symbol can not be close to two symbols at the same time, when those symbols are not close to each other. One of them should be chosen as the proximal candidate to the given symbol. For matching, it means that $f(x,x)$ does not match to $f(a,c)$ when a and c are not close to each other, even if there exists a b close both to a and c. In [11,12], we reported the first results related to block-based anti-unification with proximity relations.

In this paper, we consider the class-based notion of approximation for proximity (and tolerance) relations, which helps relax the mentioned restriction. Under this approach, $f(x,x)$ matches $f(a,c)$, when there is a b that is close to both a and c, even if a and c are not close to each other. It is justified by the fact that $f(b,b)$ and $f(a,c)$ belong to the same proximity/tolerance class, and it has a natural interpretation, e.g.: for two distant points a and c on a plane, find a point x that is close to each of them. As we have already shown in [13], it is nontrivial to solve proximity constraints in this setting. Here we develop a dedicated algorithm for matching. In general, matching problems with proximity or tolerance relations might have finitely many incomparable solutions, but one can represent them in a more compact way. We show that for each matching problem there is a single answer in such a compact form, and investigate time and space complexity to compute it.

We also study class-based anti-unification for proximity/tolerance relations. This problem is closely related to matching, as generalizations (whose computation is the goal of anti-unification) are supposed to match the original terms. Also here, we aim at computing a compact representation of the solution, but unlike matching, for anti-unification there can be finitely many different solutions in compact form. If we are interested in linear generalizations (i.e., those which do not contain multiple occurrences of the same variable) then the problem has a unique compact solution. A potential application of these techniques includes, e.g., an extension of software code clone detection methods by treating certain mismatches as approximations.

The paper is organized as follows. In Sect. 2, we introduce the basic notions. The problem statement can be found in Sect. 3. In Sect. 4, we develop our matching algorithm and study its properties. Section 5 is about anti-unification. Section 6 contains concluding remarks.

2 Preliminaries

Proximity and Tolerance Relations

We define basic notions about proximity relations following [8].

A binary *fuzzy relation* on a set S is a mapping from $S \times S$ to the real interval $[0, 1]$. If $\mathcal{R}$ is a fuzzy relation on S and λ is a number $0 < \lambda \leq 1$ (called *cut value*), then the λ-*cut* of $\mathcal{R}$ on S, denoted $\mathcal{R}_\lambda$, is an ordinary (crisp) relation on S defined as $\mathcal{R}_\lambda := \{(s_1, s_2) \mid \mathcal{R}(s_1, s_1) \geq \lambda\}$.

Each fuzzy relation is characterized by a finite set of cut values, which we call *approximation levels* of the relation.

A fuzzy relation $\mathcal{R}$ on a set S is called a *proximity relation* on S iff it is reflexive ($\mathcal{R}(s, s) = 1$ for all $s \in S$) and symmetric ($\mathcal{R}(s_1, s_2) = \mathcal{R}(s_2, s_1)$ for all $s_1, s_2 \in S$). *Tolerance relations* are crisp reflexive and symmetric binary relations. A λ-cut of a proximity relation on S is a tolerance relation on S.

The *proximity class of level* $\lambda \in (0, 1]$ *of* $s \in S$ *with respect to a proximity relation* $\mathcal{R}$ (an $(\mathcal{R}, \lambda)$-*class* of s) is the set $\mathbf{pc}(s, \mathcal{R}, \lambda) := \{s' \mid \mathcal{R}(s, s') \geq \lambda\}$.

A *triangular norm* (T-norm) $\wedge$ in $[0, 1]$ is a binary operation $\wedge : [0; 1] \times [0, 1] \to [0, 1]$, which is associative, commutative, nondecreasing in both arguments, and satisfying $x \wedge 1 = 1 \wedge x = x$ for any $x \in [0, 1]$. T-norms have been studied in detail in [10]. In this paper we assume that the t-norm is minimum.

Terms and Extended Terms

Given disjoint sets of variables $\mathcal{V}$ and fixed arity function symbols $\mathcal{F}$, *terms* over $\mathcal{F}$ and $\mathcal{V}$ are defined as usual, by the grammar $t := x \mid f(t_1, \ldots, t_n)$, where $x \in \mathcal{V}$ and $f \in \mathcal{F}$ is n-ary. The set of terms over $\mathcal{V}$ and $\mathcal{F}$ is denoted by $\mathcal{T}(\mathcal{F}, \mathcal{V})$. We denote variables by x, y, z, arbitrary function symbols by f, g, h, constants by a, b, c, and terms by s, t, r.

Below we will need a notation for finite sets of function symbols, whose all elements have the same arity. They will be denoted by lower case bold face letters: $\mathbf{f}, \mathbf{g}, \mathbf{h}$. When we talk about finite sets of constants, we use $\mathbf{a}, \mathbf{b}$, and $\mathbf{c}$.

Extended terms or, shortly, *X-terms* over $\mathcal{F}$ and $\mathcal{V}$ are defined by the grammar $\mathbf{t} := x \mid \mathbf{f}(\mathbf{t}_1, \ldots, \mathbf{t}_n)$, where $\mathbf{f} \neq \emptyset$ contains finitely many function symbols of arity n. Hence, X-terms differ from the standard ones by permitting *finite non-empty sets* of n-ary function symbols in place of n-ary function symbols. Variables in X-terms are used in the standard terms. We denote the set of X-terms over $\mathcal{F}$ and $\mathcal{V}$ by $\mathcal{T}_{\mathsf{ext}}(\mathcal{F}, \mathcal{V})$, and use also bold face letters for its elements.

The set of variables for a term t and for an X-term $\mathbf{t}$ is denoted by $\mathcal{V}(t)$ and $\mathcal{V}(\mathbf{t})$, respectively. A term (resp. X-term) is called *linear* if every variable occurs in it at most once. The *head* of a term and an X-term is defined as

$$\mathit{head}(x) := x, \quad \mathit{head}(f(t_1, \ldots, t_n)) := f, \quad \mathit{head}(\mathbf{f}(\mathbf{t}_1, \ldots, \mathbf{t}_n)) := \mathbf{f}.$$

The *set of terms represented by an X-term* $\mathbf{t}$, denoted by $\tau(\mathbf{t})$, is defined as

$$\tau(x) := \{x\}, \ \tau(\mathbf{f}(\mathbf{t}_1, \ldots, \mathbf{t}_n)) := \{f(t_1, \ldots, t_n) \mid f \in \mathbf{f}, t_i \in \tau(\mathbf{t}_i), 1 \leq i \leq n\}.$$

We also define the *intersection* operation for X-terms, denoted by $\mathbf{t} \sqcap \mathbf{s}$:

– $x \sqcap x = x$ for all $x \in \mathcal{V}$.
– $\mathbf{t} \sqcap \mathbf{s} = (\mathbf{f} \cap \mathbf{g})(\mathbf{t}_1 \sqcap \mathbf{s}_1, \ldots, \mathbf{t}_n \sqcap \mathbf{s}_n)$, $n \geq 0$, if $\mathbf{f} \cap \mathbf{g} \neq \emptyset$ and $\mathbf{t}_i \sqcap \mathbf{s}_i \neq \emptyset$ for all $1 \leq i \leq n$, where $\mathbf{t} = \mathbf{f}(\mathbf{t}_1, \ldots, \mathbf{t}_n)$ and $\mathbf{t} = \mathbf{g}(\mathbf{s}_1, \ldots, \mathbf{s}_n)$.
– $\mathbf{t} \sqcap \mathbf{s} = \emptyset$ in all other cases.

Positions in terms are defined with respect to their tree representation in the standard way, as string of integers, where the empty string is denoted by ϵ. We will need another standard notion, the *subterm of t at position p*, denoted by $t|_p$. (See, e.g., [3] for details.) These notions straightforwardly extend to X-terms. For instance, for an X-term $\mathbf{t} = \{f\}(\{g,h\}(x, \{a,b,c\}), \{b,c,d\})$, the set of positions is $\{\epsilon, 1, 1.1, 1.2, 2\}$ and we have the X-subterms of $\mathbf{t}$ at those position $\mathbf{t}|_\epsilon = \mathbf{t}$, $\mathbf{t}|_1 = \{g,h\}(x, \{a,b,c\})$, $\mathbf{t}|_{1.1} = x$, $\mathbf{t}|_{1.2} = \{a,b,c\}$, and $\mathbf{t}|_2 = \{b,c,d\}$.

Substitutions and Extended Substitutions

Substitutions over $\mathcal{T}(\mathcal{F},\mathcal{V})$ (resp. over $\mathcal{T}_{\mathsf{ext}}(\mathcal{F},\mathcal{V})$ are mappings from variables to terms (resp. to X-terms), where all but finitely many variables are mapped to themselves. The symbols $\sigma, \vartheta, \varphi$ are used for term substitutions, and $\boldsymbol{\sigma}, \boldsymbol{\vartheta}$, $\boldsymbol{\varphi}$ for X-term substitutions. The identity substitution is denoted by *Id*.

The *domain* of a substitution σ is defined as $dom(\sigma) = \{x \mid \sigma(x) \neq x\}$. We use the usual set notation for substitutions, writing, e.g., σ as $\sigma = \{x \mapsto \sigma(x) \mid x \in dom(\sigma)\}$. *Substitution application* to terms is written in the postfix notation such as $t\sigma$ and is defined recursively as $x\sigma = \sigma(x)$ and $f(t_1, \ldots, t_n)\sigma = f(t_1\sigma, \ldots, t_n\sigma)$. In the same way, one can define the domain of an X-substitution and application of an X-substitution to an X-term.[1]

The *set of substitutions represented by an X-term substitution* $\boldsymbol{\sigma}$ is the set $\tau(\boldsymbol{\sigma}) := \{\sigma \mid \sigma(x) \in \tau(\boldsymbol{\sigma}(x))$ for all $x \in \mathcal{V}\}$.

Relations over Terms and Substitutions

Each proximity relation $\mathcal{R}$ we consider in this paper is defined on $\mathcal{F}$ so that for all $f, g \in \mathcal{F}$, we have $\mathcal{R}(f,g) = 0$ if $arity(f) \neq arity(g)$. We extend such a relation $\mathcal{R}$ from $\mathcal{F}$ to $\mathcal{F} \cup \mathcal{T}(\mathcal{F},\mathcal{V})$:

– For function symbols $\mathcal{R}$ is already defined.
– For variables: $\mathcal{R}(x,x) = 1$.
– For nonvariable terms:
$$\mathcal{R}(f(t_1,\ldots,t_n), g(s_1,\ldots,s_n)) = \mathcal{R}(f,g) \wedge \mathcal{R}(t_1,s_1) \wedge \cdots \wedge \mathcal{R}(t_n,s_n),$$
when f and g are both n-ary.
– In all other cases, $\mathcal{R}(T_1,T_2) = 0$ for $T_1, T_2 \in \mathcal{F} \cup \mathcal{T}(\mathcal{F},\mathcal{V})$.

Two terms t and s are $(\mathcal{R},\lambda)$-*close* to each other, written $t \simeq_{\mathcal{R},\lambda} s$, if $\mathcal{R}(t,s) \geq \lambda$.

[1] Note that notions of application of a substitution to an X-term and application of an X-substitution to a term are not defined.

Definition 1 (Relations $\preceq_{\mathcal{R},\lambda}$ and $\leqslant$). *The relations $\preceq_{\mathcal{R},\lambda}$ and $\leqslant$ and the corresponding notions are defined as follows:*

$\preceq_{\mathcal{R},\lambda}$: *A term t is* $(\mathcal{R},\lambda)$-more general than *s (or t is* $(\mathcal{R},\lambda)$-generalization of *s, or s is an* $(\mathcal{R},\lambda)$-instance *of t), written $t \preceq_{\mathcal{R},\lambda} s$, if there exists a substitution σ such that $t\sigma \simeq_{\mathcal{R},\lambda} s$. We say that σ is an* $(\mathcal{R},\lambda)$-matcher *of t to s.*

$\leqslant$: *A term t is* syntactically more general than *s (or t is a* syntactic generalization *of s, or s is a* syntactic instance *of t), written $t \leqslant s$, if there exists a σ such that $t\sigma = s$. We say that σ is a syntactic matcher of t to s.*

An X-term $\mathbf{t}$ is an $(\mathcal{R},\lambda)$-X-generalization *of a term s, if every $t \in \tau(\mathbf{t})$ is an* $(\mathcal{R},\lambda)$-generalization *of s.*

An X-substitution $\boldsymbol{\sigma}$ is an $(\mathcal{R},\lambda)$-X-matcher *of t to s, if every $\sigma \in \tau(\boldsymbol{\sigma})$ is an* $(\mathcal{R},\lambda)$-matcher *of t to s.*

A substitution σ that matches t to s is called a relevant $(\mathcal{R},\lambda)$-matcher *(resp.* relevant syntactic matcher*) of t to s if $dom(\sigma) \subseteq \mathcal{V}(t)$. A relevant $(\mathcal{R},\lambda)$-X-matcher is defined analogously.*

The strict part of $\preceq_{\mathcal{R},\lambda}$ and $\leqslant$ are denoted respectively by $\prec_{\mathcal{R},\lambda}$ and $<$.

The relation $\preceq_{\mathcal{R},\lambda}$ is not transitive. If $a \simeq_{\mathcal{R},\lambda} b$, $b \simeq_{\mathcal{R},\lambda} c$, and $a \not\simeq_{\mathcal{R},\lambda} c$, then we have $a \preceq_{\mathcal{R},\lambda} b$, $b \preceq_{\mathcal{R},\lambda} c$, and $a \not\preceq_{\mathcal{R},\lambda} c$. Unlike $\preceq_{\mathcal{R},\lambda}$, $\leqslant$ is transitive. (In fact, $\leqslant$ is a quasi-ordering, called instantiation quasi-ordering.) We also have $\leqslant \ \subseteq \ \preceq_{\mathcal{R},\lambda}$ for any $\mathcal{R}$ and λ.

Definition 2 ($(\mathcal{R},\lambda)$-lgg). *A term r is called an* $(\mathcal{R},\lambda)$-least general generalization *(an* $(\mathcal{R},\lambda)$-lgg*) of t and s iff*

- *r is $(\mathcal{R},\lambda)$-more general than both t and s, i.e., $r \preceq_{\mathcal{R},\lambda} t$ and $r \preceq_{\mathcal{R},\lambda} s$, and*
- *there is no r' such that $r \prec_{\mathcal{R},\lambda} r'$, $r' \preceq_{\mathcal{R},\lambda} t$, and $r' \preceq_{\mathcal{R},\lambda} s$.*

An X-term $\mathbf{r}$ is an $(\mathcal{R},\lambda)$-X-lgg *of t and s, if every $r \in \tau(\mathbf{r})$ is an* $(\mathcal{R},\lambda)$-lgg *of t and s.*

Theorem 1. *If r is an $(\mathcal{R},\lambda)$-generalization of t, then any syntactic generalization of r is also an $(\mathcal{R},\lambda)$-generalization of t.*

Proof. From $r \preceq_{\mathcal{R},\lambda} t$, by definition of $\preceq_{\mathcal{R},\lambda}$, there exists ϑ such that $r\vartheta \simeq_{\mathcal{R},\lambda} t$. From $r' \leqslant r$, by definition of $\leqslant$, there exists φ such that $r'\varphi = r$. Then we have $r'\varphi\vartheta = r\vartheta \simeq_{\mathcal{R},\lambda} t$, which implies $r' \preceq_{\mathcal{R},\lambda} t$. $\square$

Corollary 1. *Any syntactic generalization of an $(\mathcal{R},\lambda)$-lgg of t and s is an $(\mathcal{R},\lambda)$-generalization of both t and s.*

The notion of syntactic lgg can be defined analogously to $(\mathcal{R},\lambda)$-lgg, using the relation $\leqslant$. The syntactic lgg of two terms is unique modulo variable renaming, see, e.g., [14, 15]. In general, it is not difficult to show that for any terms t and s, if r and r' are their syntactic lgg and $(\mathcal{R},\lambda)$-lgg, respectively, then $r \leqslant r'$.

Example 1. Let $\mathcal{R}$ and λ be such that $a \simeq_{\mathcal{R},\lambda} b$, $b \simeq_{\mathcal{R},\lambda} c$, and $a \not\simeq_{\mathcal{R},\lambda} c$. Then $(\mathcal{R},\lambda)$-lgg of a and c is b, while their syntactic lgg is x.

Given a term t, a proximity relation $\mathcal{R}$, and a cut value λ, the $(\mathcal{R}, \lambda)$-*proximity class* of t is an X-term $\mathbf{pc}(t, \mathcal{R}, \lambda)$, defined as

$$\begin{aligned}&\mathbf{pc}(x, \mathcal{R}, \lambda) := \{x\},\\&\mathbf{pc}(f(t_1, \ldots, t_n), \mathcal{R}, \lambda) := \mathbf{pc}(f, \mathcal{R}, \lambda)(\mathbf{pc}(t_1, \mathcal{R}, \lambda), \ldots, \mathbf{pc}(t_n, \mathcal{R}, \lambda)).\end{aligned}$$

Theorem 2. *Given a proximity relation $\mathcal{R}$, a cut value λ, and two terms t and s, each $r \in \mathbf{pc}(t, \mathcal{R}, \lambda) \sqcap \mathbf{pc}(s, \mathcal{R}, \lambda)$ is $(\mathcal{R}, \lambda)$-close both to t and to s.*

Proof. Follows directly from the definition of proximity class of a term. □

The examples below illustrate some of the notions introduced in this section.

Example 2. Let the proximity relation $\mathcal{R}$ be defined as

$$\begin{aligned}&\mathcal{R}(g_1, g_2) = \mathcal{R}(a_1, a_2) = 0.5, &&\mathcal{R}(g_1, h_1) = \mathcal{R}(g_2, h_1) = 0.6,\\&\mathcal{R}(g_1, h_2) = \mathcal{R}(a_1, b) = 0.7, &&\mathcal{R}(g_2, h_2) = \mathcal{R}(a_2, b) = 0.8.\end{aligned}$$

The set of approximation levels of $\mathcal{R}$ is $\{0.5, 0.6, 0.7, 0.8\}$.

Let t be the term $f(g_1(a_1), g_2(a_2))$. Then the proximity class $\mathbf{pc}(t, \mathcal{R}, \lambda)$ for different values of λ is:

$$\begin{aligned}0 < \lambda \le 0.5:&\ \{f\}(\{g_1, g_2, h_1, h_2\}(\{a_1, a_2, b\}), \{g_1, g_2, h_1, h_2\}(\{a_1, a_2, b\})).\\0.5 < \lambda \le 0.6:&\ \{f\}(\{g_1, h_1, h_2\}(\{a_1, b\}), \{g_2, h_1, h_2\}(\{a_2, b\})).\\0.6 < \lambda \le 0.7:&\ \{f\}(\{g_1, h_2\}(\{a_1, b\}), \{g_2, h_2\}(\{a_2, b\})).\\0.7 < \lambda \le 0.8:&\ \{f\}(\{g_1\}(\{a_1\}), \{g_2, h_2\}(\{a_2, b\})).\\0.8 < \lambda \le 1:&\ \{f\}(\{g_1\}(\{a_1\}), \{g_2\}(\{a_2\})).\end{aligned}$$

Example 3. Let $\mathcal{R}$ be defined as in Example 2. Let $t = f(x, x)$ and $s = f(g_1(a_1), g_2(a_2))$. Then for each of the following X-substitution $\boldsymbol{\sigma}$, the set $\tau(\boldsymbol{\sigma})$ contains all relevant $(\mathcal{R}, \lambda)$-matchers of t to s for different values of λ:

$$\begin{aligned}0 < \lambda \le 0.5:&\quad \boldsymbol{\sigma} = \{x \mapsto \{g_1, g_2, h_1, h_2\}(\{a_1, a_2, b\})\}.\\&\quad \tau(\boldsymbol{\sigma}) \text{ contains 12 substitutions.}\\0.5 < \lambda \le 0.6:&\quad \boldsymbol{\sigma} = \{x \mapsto \{h_1, h_2\}(\{b\})\}.\\&\quad \tau(\boldsymbol{\sigma}) = \{\{x \mapsto h_1(b)\},\ \{x \mapsto h_2(b)\}\}.\\0.6 < \lambda \le 0.7:&\quad \boldsymbol{\sigma} = \{x \mapsto \{h_2\}(\{b\})\}. \qquad \tau(\boldsymbol{\sigma}) = \{\{x \mapsto h_2(b)\}\}.\\0.7 < \lambda \le 1:&\quad \text{No substitution matches } t \text{ to } s.\end{aligned}$$

Example 4. Let $\mathcal{R}$ be a proximity relation defined as

$$\begin{aligned}&\mathcal{R}(a_1, a) = \mathcal{R}(a_2, a) = \mathcal{R}(b_1, b) = \mathcal{R}(b_2, b) = 0.5,\\&\mathcal{R}(a_2, a') = \mathcal{R}(a_3, a') = \mathcal{R}(b_2, b') = \mathcal{R}(b_3, b') = 0.6, \qquad \mathcal{R}(f, g) = 0.7.\end{aligned}$$

Its set of approximation levels is $\{0.5, 0.6, 0.7\}$.

Let $t = f(a_1, a_2, a_3)$ and $s = g(b_1, b_2, b_3)$. Then x is the syntactic lgg of t and s. As for proximity-based generalizations, for each of the following X-term $\mathbf{r}$, the set $\tau(\mathbf{r})$ contains all $(\mathcal{R}, \lambda)$-lggs of t and s for different values of λ:

$0 < \lambda \leq 0.5$:

$$\mathbf{r}_1 = \{f, g\}(x_1, x_1, x_3). \quad \tau(\mathbf{r}_1) = \{f(x_1, x_1, x_3),\ g(x_1, x_1, x_3)\}.$$
$$\mathbf{r}_2 = \{f, g\}(x_1, x_2, x_2). \quad \tau(\mathbf{r}_2) = \{f(x_1, x_2, x_2),\ g(x_1, x_2, x_2)\}.$$

$0.5 < \lambda \leq 0.6$:

$$\mathbf{r} = \{f, g\}(x_1, x_2, x_2). \quad \tau(\mathbf{r}) = \{f(x_1, x_2, x_2),\ g(x_1, x_2, x_2)\}.$$

$0.6 < \lambda \leq 0.7$:

$$\mathbf{r} = \{f, g\}(x_1, x_2, x_3). \quad \tau(\mathbf{r}) = \{f(x_1, x_2, x_3),\ g(x_1, x_2, x_3)\}.$$

$0.7 < \lambda \leq 1$: $\quad \mathbf{r} = x. \quad \tau(\mathbf{r}) = \{x\}.$

If we are interested only in linear generalizations, we will get a single X-term $(\mathcal{R}, \lambda)$-lgg for each fixed λ:

$0 < \lambda \leq 0.7$: $\quad \mathbf{r} = \{f, g\}(x_1, x_2, x_3). \quad \tau(\mathbf{r}) = \{f(x_1, x_2, x_3),\ g(x_1, x_2, x_3)\}.$

$0.7 < \lambda \leq 1$: $\quad \mathbf{r} = x. \quad \tau(\mathbf{r}) = \{x\}.$

3 Matching and Anti-Unification: Problem Statement

Matching and anti-unification problems for terms are formulated as follows: Given a proximity relation $\mathcal{R}$, a cut value λ, and two terms t and s, find

- an $(\mathcal{R}, \lambda)$-matcher of t to s (the matching problem) or
- an $(\mathcal{R}, \lambda)$-lgg of t and s (the anti-unification problem).

Below we develop algorithms to solve these problems. as we will see, each of them has finitely many solutions. It is important to mention that instead of computing all the solutions to the problems, we will be aiming at computing their compact representations in the form of X-substitutions (for matching) and X-terms (for generalization). Hence, our algorithms will solve the following reformulated version of the problems:

Matching problem

Given: a proximity relation $\mathcal{R}$, a cut value λ, and two terms t and s.
Find: an X-substitution $\boldsymbol{\sigma}$ s.t. each $\sigma \in \tau(\boldsymbol{\sigma})$ is an $(\mathcal{R}, \lambda)$-matcher of t to s.

Anti-unification problem

Given: a proximity relation $\mathcal{R}$, a cut value λ, and two terms t and s.
Find: an X-term $\mathbf{r}$ such that each $r \in \tau(\mathbf{r})$ is an $(\mathcal{R}, \lambda)$-lgg of t and s.

Such a reformulation will help us compute a single X-substitution instead of multiple matchers, and fewer X-lggs than lggs. Moreover, if we restrict ourselves to linear lggs (i.e., those with a single occurrence of generalization variables), then also here we get a single answer.

4 Matching

Given $\mathcal{R}$, λ, t, and s (where s does not contain variables), to solve an $(\mathcal{R},\lambda)$-matching problem $t \ll s$, we create the initial pair $\{t \ll s\}; \emptyset$ and apply the rules given below. They work on pairs $M; S$, where M is a set of matching problems, and S is the set of equations of the form $x \approx \mathbf{s}$. The rules are as follows ($\uplus$ stands for disjoint union):

Dec-M: **Decomposition**

$\{f(t_1,\dots,t_n) \ll g(s_1,\dots,s_n)\} \uplus M; S \Longrightarrow M \cup \{t_i \ll s_i \mid 1 \le i \le n\}; S,$
if $n \geq 0$, $\mathcal{R}(f,g) \geq \lambda$.

VE-M: **Variable elimination**

$\{x \ll s\} \uplus M;\ S \Longrightarrow M;\ S \cup \{x \approx \mathbf{pc}(s,\mathcal{R},\lambda)\}.$

Mer-M: **Merging**

$M;\ \{x \approx \mathbf{s}_1, x \approx \mathbf{s}_2\} \uplus S \Longrightarrow M;\ S \cup \{x \approx \mathbf{s}_1 \sqcap \mathbf{s}_2\}, \qquad \text{if } \mathbf{s}_1 \sqcap \mathbf{s}_2 \neq \emptyset.$

Cla-M: **Clash**

$\{f(t_1,\dots,t_n) \ll g(s_1,\dots,s_m)\} \uplus M;\ S \Longrightarrow \bot, \qquad \text{where } \mathcal{R}(f,g) < \lambda.$

Inc-M: **Inconsistency**

$M;\ \{x \approx \mathbf{s}_1,\ x \approx \mathbf{s}_2\} \uplus S \Longrightarrow \bot, \qquad \text{if } \mathbf{s}_1 \sqcap \mathbf{s}_2 = \emptyset.$

The matching algorithm $\mathfrak{M}$ uses the rules to transform pairs as long as possible, returning either $\bot$ (indicating failure), or the pair $\emptyset; S$ (indicating success). In the latter case, each variable occurs in S at most once and from S one can obtain an X-substitution $\{x \mapsto \mathbf{s} \mid x \approx \mathbf{s} \in S\}$. We call it the *computed X-substitution.*

We call a substitution σ an $(\mathcal{R},\lambda)$-solution of an $M; S$ pair, iff σ is an $(\mathcal{R},\lambda)$-matcher of M and for all $x \approx \mathbf{t} \in S$, we have $x\sigma \in \tau(\mathbf{t})$. We also assume that $\bot$ has no solution.

Lemma 1. *If $M_1; S_1 \Longrightarrow M_2; S_2$ is a step made by $\mathfrak{M}$, then σ is an $(\mathcal{R},\lambda)$-solution of $M_1; S_1$ iff it is an $(\mathcal{R},\lambda)$-solution of $M_2; S_2$.*

Proof. For the rules Dec-M and Cla-M, the lemma follows by definition of matcher. For Mer-M and Inc-M it is implied by definition of $\sqcap$. For VE-M, by definition of $\mathbf{pc}$, we have $x\sigma \in \mathbf{pc}(s,\mathcal{R},\lambda)$ iff $\mathcal{R}(x\sigma, s) \geq \lambda$, which is equivalent to the fact that σ is an $(\mathcal{R},\lambda)$-matcher of $x \ll s$. □

In the theorems below the *size* of a syntactic object (term, matching problem, set of matching problems, a set of equations) is the number of alphabet symbols in it: $size(x) = 1$, $size(f(t_1,\dots,t_n)) = 1 + \sum_{i=1}^{n} size(t_i)$, $size(t \ll s) = size(t \approx s) = size(t) + size(s)$, and $size(S) = \sum_{p \in S} size(p)$, where S is a set of matching problems or equations.

Theorem 3. *Given an $(\mathcal{R},\lambda)$-matching problem $t \ll s$, the matching algorithm $\mathfrak{M}$ terminates and computes an X-substitution $\boldsymbol{\sigma}$ such that $\tau(\boldsymbol{\sigma})$ consists of all relevant $(\mathcal{R},\lambda)$-matchers of t to s.*

Proof. The theorem consists of three parts: termination, soundness, and completeness. We prove each of them separately.

Termination. The rules Dec-M and VE-M strictly reduce the number of symbols in M. The rule Mer-M does the same for S, without changing M. Cla-M and Inc-M stop the algorithm immediately. Hence, the algorithm strictly reduces the lexicographic combination $\langle \mathit{size}(M), \mathit{size}(S)\rangle$ of sizes of M and S, which implies termination.

Soundness. If $\sigma \in \tau(\boldsymbol{\sigma})$, then σ is a relevant $(\mathcal{R},\lambda)$-matcher of t to s.
Let $\{t \ll s\}; \emptyset \Longrightarrow^+ \emptyset; S$ be the derivation in $\mathfrak{M}$ that computes $\boldsymbol{\sigma}$. By definition of computed X-substitution, we can conclude that $\sigma \in \tau(\boldsymbol{\sigma})$ iff σ is a solution of $\emptyset; S$. By induction on the length of the given derivation, using Lemma 1, we can prove that σ is an $(\mathcal{R},\lambda)$-matcher of t to s. In $\mathfrak{M}$, no new variables are created and put in S. All variables there come from the original problem. It implies that σ is a relevant matcher of t to s.

Completeness. If σ is a relevant $(\mathcal{R},\lambda)$-matcher of t to s, then $\sigma \in \tau(\boldsymbol{\sigma})$.
Since $t \ll s$ is solvable, we can construct a derivation $\{t \ll s\}; \emptyset \Longrightarrow^+ \emptyset; S$ in $\mathfrak{M}$. This follows from the fact that for each form of matching equation we have a rule in $\mathfrak{M}$, and if we have two equations with the same variable in S we can also transform it. Moreover, by Lemma 1, we would never apply Cla-M and Inc-M rules, because it would contradict the solvability of $t \ll s$. Hence, we can construct the mentioned derivation, for which, again by Lemma 1, we have that σ is a $(\mathcal{R},\lambda)$-solution of $\emptyset; S$. By definitions of computed X-substitution $\boldsymbol{\sigma}$ and τ, it implies that $\sigma \in \tau(\boldsymbol{\sigma})$. □

Hence, $\mathfrak{M}$ computes all relevant $(\mathcal{R},\lambda)$-X-matchers for matching problems.

Example 5. We illustrate the steps of the algorithm $\mathfrak{M}$ for the matching problem in Example 3 for $\lambda = 0.6$ and $\lambda = 0.8$.

$\lambda = 0.6:$

$$\begin{aligned}
&\{f(x,x) \ll f(g_1(a_1), g_2(a_2))\}; \emptyset \Longrightarrow_{\text{Dec-M}}\\
&\{x \ll g_1(a_1),\ x \ll g_2(a_2)\}; \emptyset \Longrightarrow_{\text{VE-M}}\\
&\{x \ll g_2(a_2)\}; \{x \approx \{g_1,h_1,h_2\}(\{a_1,b\})\} \Longrightarrow_{\text{VE-M}}\\
&\emptyset; \{x \approx \{g_1,h_1,h_2\}(\{a_1,b\}),\ x \approx \{g_2,h_1,h_2\}(\{a_2,b\})\} \Longrightarrow_{\text{Mer-M}}\\
&\emptyset; \{x \approx \{h_1,h_2\}(\{b\})\}.
\end{aligned}$$

$\lambda = 0.8:$

$$\begin{aligned}
&\{f(x,x) \ll f(g_1(a_1), g_2(a_2))\}; \emptyset \Longrightarrow_{\text{Dec-M}}\\
&\{x \ll g_1(a_1),\ x \ll g_2(a_2)\}; \emptyset \Longrightarrow_{\text{VE-M}}\\
&\{x \ll g_2(a_2)\}; \{x \approx \{g_1\}(\{a_1\})\}; \Longrightarrow_{\text{VE-M}}\\
&\emptyset; \{x \approx \{g_1\}(\{a_1\}),\ x \approx \{g_2,h_2\}(\{a_2,b\})\} \Longrightarrow_{\text{Inc-M}} \bot.
\end{aligned}$$

The proximity relation $\mathcal{R}$ can be represented as a weighted undirected graph, whose vertices form a (finite) subset of $\mathcal{F}$ and if $\mathcal{R}(f, g) = \mathfrak{d} > 0$ for two vertices f and g, then there is an edge of weight $\mathfrak{d}$ between them. When we consider $\mathcal{R}$ as a graph, we represent it as a pair $(V_{\mathcal{R}}, E_{\mathcal{R}})$ of the sets of vertices $V_{\mathcal{R}}$ and edges $E_{\mathcal{R}}$. We denote by $|S|$ the number of elements in the (finite) set S.

Graphs induced by proximity relations are sparse, since symbols of different arities are not close to each other. Therefore, in the proofs of complexity results below, we choose to represent the graphs by adjacency lists rather than by adjacency matrices.

Theorem 4. *Let $\mathcal{R} = (V_{\mathcal{R}}, E_{\mathcal{R}})$ be a proximity relation and M be a matching problem with size$(M) = n$. Then the algorithm $\mathfrak{M}$ needs $O(n|V_{\mathcal{R}}| + n|E_{\mathcal{R}}|)$ time and $O(n|V_{\mathcal{R}}| + |E_{\mathcal{R}}|)$ space to compute the $(\mathcal{R}, \lambda)$-solution to M for a given λ.*

Proof. We represent the graph for $\mathcal{R}$ as adjacency lists, in which proximity degrees are weights of edges. Such a weight of an edge (v_1, v_2) is stored at the vertex v_2 in the adjacency list of v_1 and vice versa [7]. Further, from the given matching problem $t \ll s$ we can construct its directed acyclic graph (dag) representation with shared variables (see, e.g., [4]). At each node g of s, we add a pointer to the entry in the adjacency list of $\mathcal{R}$ for the symbol g. The nodes in the representation of t are labeled by function symbols and variables occurring in t. In fact, we have a graph representation $\mathit{dag}(t)$ of t and a tree representation $\mathit{tree}(s)$ of s, since there are no variables to share in s.

During the run of the algorithm, we follow the structures top-down both in $\mathit{dag}(t)$ and $\mathit{tree}(s)$, comparing the node labels pairwise. Assume the label of a nonvariable node f in $\mathit{dag}(t)$ is an element of the adjacency list of a node g in $\mathit{tree}(s)$, and $\mathfrak{d} \geq \lambda$ for the degree $\mathfrak{d}$ stored together with f in the adjacency list. Then the Dec-M rule is applied and we proceed to the successor nodes of f and g (pairwise), as usual. Otherwise we stop (Cla-M rule).

When we reach a variable node x in $\mathit{dag}(t)$ and a node g in $\mathit{tree}(s)$, we check whether there already exists a pointer from x to the root h of some tree tree_h. If not, we make a copy tree_g of the subtree $\mathit{subtree}(s, g)$ of $\mathit{tree}(s)$ rooted at g. It means that the adjacency lists of the nodes of this subtree are also copied, not shared. We call the copies of those lists the class labels. After that, we make a pointer from x to g in tree_g, and continue with the next unvisited node-pairs in $\mathit{dag}(t)$ and $\mathit{tree}(s)$ (VE-M rule). If tree_h to which x points already exists, we go top-down to the trees tree_h and $\mathit{subtree}(s, g)$, updating the class label at each node of tree_h: if the class label at some node in this tree is L_1, and the adjacency list of the corresponding node in $\mathit{subtree}(s, g)$ is L_2, we replace L_1 in tree_h by $L_1 \cap L_2$, provided that $L_1 \cap L_2 \neq \emptyset$, and continue with the next unvisited node-pair. This process corresponds to the Mer-M rule, eagerly applied immediately after VE-M. If either the intersection is empty, or one tree is deeper than the other, then we stop with failure (the Inc-M rule).

First we make the space analysis. The adjacency list representation of $\mathcal{R}$ needs $O(|V_{\mathcal{R}}| + |E_{\mathcal{R}}|)$ space [7]. The graph/tree representation of the matching problem requires $O(n)$ space. All the copies of trees generated by the VE-M rule may contain in total at most n nodes, each labeled with at most $|V_{\mathcal{R}}|$ symbols, i.e., to store them we need $O(n|V_{\mathcal{R}}|)$ space. Hence, the total amount of required memory is $O(n|V_{\mathcal{R}}| + |E_{\mathcal{R}}|)$.

For the runtime complexity, constructing the adjacency list needs $O(|V_{\mathcal{R}}| + |E_{\mathcal{R}}|)$ time. Construction of the dag/tree representation of the matching problem can be done in $O(n)$ time [4]. Each node in $dag(t)$ and $tree(s)$ is visited once. Hence, the structure traversal is done in linear time with respect to n. Checking the membership of some vertex f from $dag(t)$ in the adjacency list of some vertex g in $tree(s)$, needed in the Dec-M rule, requires $O(degree(g))$ time. Since this check is performed $O(n)$ times, and a (rough) upper bound for vertex degrees is $|E_{\mathcal{R}}|$, we can say that the total time needed for the adjacency list membership operation during the run of $\mathfrak{M}$ is $O(n|E_{\mathcal{R}}|)$. Creating the copies of trees by the VE-M rule is constant for each symbol, thus needing $O(n|V_{\mathcal{R}}|)$ time. Computation of intersections between two proximity classes needs $O(|V_{\mathcal{R}}|)$ time. We may need to perform $O(n)$ such intersections, hence for them we need $O(n|V_{\mathcal{R}}|)$ time. It implies that the runtime complexity of the matching algorithm is $O(n|V_{\mathcal{R}}| + n|E_{\mathcal{R}}|)$. □

4.1 Computing Approximation Degrees for Matching

The algorithm above does not compute approximation degrees for the returned matchers. We can add this feature with a small modification of the notions.

A *graded set of function symbols* is a finite set of pairs $\{\langle f_1, \alpha_1\rangle, \dots, \langle f_n, \alpha_n\rangle\}$, where $\alpha_i \in (0, 1]$ and all f_i's have the same arity. *Graded X-terms* are constructed from graded function symbol sets and variables in the same way X-terms were constructed from non-graded symbol sets and variables. We reuse the bold face letters $\mathbf{f}, \mathbf{a}, \mathbf{t}$, etc. that denote the non-graded counterparts of these notions.

The intersection of graded function symbol sets is defined as $\mathbf{f}_1 \cap \mathbf{f}_2 := \{\langle f, \alpha_1 \wedge \alpha_2\rangle \mid \langle f, \alpha_1\rangle \in \mathbf{f}_1, \langle f, \alpha_2\rangle \in \mathbf{f}_2\}$. Then the intersection of graded X-terms $\mathbf{t}_1 \sqcap \mathbf{t}_2$ is defined as it was done for non-graded X-terms, using the intersection of graded sets of functions symbols.

The *graded* $(\mathcal{R}, \lambda)$*-proximity class* for a symbol f is a set $\{\langle g, \alpha\rangle \mid \mathcal{R}(f, g) = \alpha \geq \lambda\}$. Also here, we reuse the notation from its non-graded version: $\mathbf{pc}(f, \mathcal{R}, \lambda)$. The proximity class for a term is defined and denoted similarly.

Example 6. Let $\mathcal{R}$ be the proximity relation defined in example 2.

Let t be the term $f(g_1(a_1), g_2(a_2))$. Then the graded proximity class for it, $\mathbf{pc}(t, \mathcal{R}, \lambda)$, for different values of λ is:

$$\begin{aligned}
0 < \lambda \leq 0.5 :&\\
&\{\langle f,1\rangle\}(\{\langle g_1,1\rangle, \langle g_2,0.5\rangle, \langle h_1,0.6\rangle, \langle h_2,0.7\rangle\}(\{\langle a_1,1\rangle, \langle a_2,0.5\rangle, \langle b,0.7\rangle\}),\\
&\qquad \{\langle g_1,0.5\rangle, \langle g_2,1\rangle, \langle h_1,0.6\rangle, \langle h_2,0.8\rangle\}(\{\langle a_1,0.5\rangle, \langle a_2,1\rangle, \langle b,0.8\rangle\})).
\end{aligned}$$

$$\begin{aligned}
0.5 < \lambda \leq 0.6 :\ & \{\langle f,1\rangle\}(\{\langle g_1,1\rangle, \langle h_1,0.6\rangle, \langle h_2,0.7\rangle\}(\{\langle a_1,1\rangle, \langle b,0.7\rangle\}),\\
&\qquad \{\langle g_2,1\rangle, \langle h_1,0.6\rangle, \langle h_2,0.8\rangle\}(\{\langle a_2,1\rangle, \langle b,0.8\rangle\})).
\end{aligned}$$

$$0.6 < \lambda \leq 0.7 : \{\langle f,1\rangle\}(\{\langle g_1,1\rangle, \langle h_2,0.7\rangle\}(\{\langle a_1,1\rangle, \langle b,0.7\rangle\}),$$

$$\{\langle g_2, 1\rangle, \langle h_2, 0.8\rangle\}(\{\langle a_2, 1\rangle, \langle b, 0.8\rangle\})).$$

$$0.7 < \lambda \leq 0.8 : \{\langle f, 1\rangle\}(\{\langle g_1, 1\rangle\}(\{\langle a_1, 1\rangle\}),$$
$$\{\langle g_2, 1\rangle, \langle h_2, 0.8\rangle\}(\{\langle a_2, 1\rangle, \langle b, 0.8\rangle\})).$$

$$0.8 < \lambda \leq 1 \quad : \{\langle f, 1\rangle\}(\{\langle g_1, 1\rangle\}(\{\langle a_1, 1\rangle\}), \{\langle g_2, 1\rangle\}(\{\langle a_2, 1\rangle\})).$$

The modified version of the matching algorithm works on triples $M; S; \alpha$, where S is a set of equations between variables and graded X-terms, and α is the approximation degree between function symbols, initialized with 1. The only rule we need to change is Dec-M, which should update the approximation degree α:

Dec-M: **Decomposition**

$$\{f(t_1, \ldots, t_n) \ll g(s_1, \ldots, s_n)\} \uplus M;\ S;\ \alpha \Longrightarrow$$
$$M \cup \{t_i \ll s_i \mid 1 \leq i \leq n\};\ S;\ \alpha \wedge \mathcal{R}(f, g),$$

if $n \geq 0$, $\mathcal{R}(f, g) \geq \lambda$.

The graded counterpart of τ, denoted by τ_{gr}, takes a graded X-term and returns a set of pairs $\langle t, \alpha\rangle$ where t is a term and $\alpha \in (0, 1]$. It is defined as

$$\tau_{\mathrm{gr}}(x) := \{\langle x, 1\rangle\},$$
$$\tau_{\mathrm{gr}}(\mathbf{f}(\mathbf{t}_1, \ldots, \mathbf{t}_n) := \{\langle f(t_1, \ldots, t_n), \alpha\rangle \mid \langle f, \alpha_0\rangle \in \mathbf{f}, \langle t_i, \alpha_i\rangle \in \tau_{\mathrm{gr}}(\mathbf{t}_i),$$
$$1 \leq i \leq n,\ \alpha = \alpha_0 \wedge \cdots \wedge \alpha_n\}.$$

Similarly, we define τ_{gr} for graded X-substitutions: $\tau_{\mathrm{gr}}(\boldsymbol{\sigma})$ is a set of pairs $\langle \sigma, \alpha\rangle$, where σ is a substitution and $\alpha \in (0, 1]$, defined as

$$\tau_{\mathrm{gr}}(\{x_1 \mapsto \mathbf{t}_1, \ldots, x_n \mapsto \mathbf{t}_n\}) :=$$
$$\{\langle \{x_1 \mapsto t_1, \ldots, x_n \mapsto t_n\}, \alpha_1 \wedge \cdots \wedge \alpha_n\rangle \mid \langle t_i, \alpha_i\rangle \in \tau_{\mathrm{gr}}(\mathbf{t}_i),\ 1 \leq i \leq n\}.$$

When it succeeds, the matching algorithm stops with the triple of the form $\emptyset;\ \{x_1 \approx \mathbf{s}_1, \ldots, x_n \approx \mathbf{s}_n\};\ \alpha$. From this representation, we take $\boldsymbol{\sigma} = \{x_1 \mapsto \mathbf{s}_1, \ldots, x_n \mapsto \mathbf{s}_n\}$ as the computed graded X-substitution and α as the upper bound of approximation degrees of all matchers. We can use $\boldsymbol{\sigma}$ and α as the basis from which various concrete matchers of the original $(\mathcal{R}, \lambda)$-matching problem M and their approximation degrees can be obtained. For instance:

– Extract each solution and its approximation degree from $\tau_{\mathrm{gr}}(\boldsymbol{\sigma})$: For each $\langle \sigma, \alpha_\sigma\rangle \in \tau_{\mathrm{gr}}(\boldsymbol{\sigma})$ we get $\langle \sigma, \alpha_\sigma \wedge \alpha\rangle$.
– If we are interested only in matchers with the maximal approximation degree, we select $\langle \sigma, \alpha_\sigma\rangle \in \tau_{\mathrm{gr}}(\boldsymbol{\sigma})$ so that $\alpha_\sigma \wedge \alpha$ is maximal (there can be several of them), without unpacking the whole set of solutions.

Example 7. Let the proximity relation $\mathcal{R}$ be obtained by adding $\mathcal{R}(f_1, f_2) = 0.8$ to the one defined in Example 2. Let $t = f_1(x, x)$ and $s = f_2(g_1(a_1), g_2(a_2))$.

We illustrate the steps of the algorithm $\mathfrak{M}$ for the matching problem $f_1(x,x) \ll f_2(g_1(a_1), g_2(a_2))$ for $\lambda = 0.6$.

$$\begin{aligned}
&\{f_1(x,x) \ll f_2(g_1(a_1), g_2(a_2))\}; \emptyset; 1 \Longrightarrow_{\textsc{Dec-M}}\\
&\{x \ll g_1(a_1),\ x \ll g_2(a_2)\}; \emptyset; 0.8 \Longrightarrow_{\text{VE-M}}\\
&\{x \ll g_2(a_2)\}; \{x \approx \{\langle g_1, 1\rangle, \langle h_1, 0.6\rangle, \langle h_2, 0.7\rangle\}(\{\langle a_1, 1\rangle, \langle b, 0.7\rangle\})\}; 0.8\\
&\qquad \Longrightarrow_{\text{VE-M}}\\
&\emptyset; \{x \approx \{\langle g_1, 1\rangle, \langle h_1, 0.6\rangle, \langle h_2, 0.7\rangle\}(\{\langle a_1, 1\rangle, \langle b, 0.7\rangle\}),\\
&\qquad x \approx \{\langle g_2, 1\rangle, \langle h_1, 0.6\rangle, \langle h_2, 0.8\rangle\}(\{\langle a_2, 1\rangle, \langle b, 0.8\rangle\})\}; 0.8 \Longrightarrow_{\textsc{Mer-M}}\\
&\emptyset; \{x \approx \{\langle h_1, 0.6\rangle, \langle h_2, 0.7\rangle\}(\{\langle b, 0.7\rangle\})\}; 0.8.
\end{aligned}$$

If we want to extract all $(\mathcal{R}, \lambda)$-matchers, we would return $\langle\{x \mapsto h_1(b)\}, 0.6\rangle$ and $\langle\{x \mapsto h_2(b)\}, 0.7\rangle$. The maximal solution would be only $\langle\{x \mapsto h_2(b)\}, 0.7\rangle$.

If we had $\mathcal{R}(f_1, f_2) = 0.6$, then the sets of all $(\mathcal{R}, \lambda)$-matchers and all maximal $(\mathcal{R}, \lambda)$-matchers would coincide. They both would be $\{\langle\{x \mapsto h_1(b)\}, 0.6\rangle, \langle\{x \mapsto h_2(b)\}, 0.6\rangle\}$.

5 Anti-Unification

Given $\mathcal{R}$ and λ, for solving an $(\mathcal{R}, \lambda)$-anti-unification problem between two terms t and s, we create the anti-unification triple (AUT) $x : \mathbf{pc}(t, \mathcal{R}, \lambda) \triangleq \mathbf{pc}(s, \mathcal{R}, \lambda)$ where x is a fresh variable. Then we put it in the initial tuple $\{x : \mathbf{pc}(t, \mathcal{R}, \lambda) \triangleq \mathbf{pc}(s, \mathcal{R}, \lambda)\}; \emptyset; x$, and apply the rules given below. They work on tuples $A; S; \mathbf{r}$, where A is a set of AUTs to be solved (called the AU-problem set), S is the set consisting of AUTs already solved (called the store), and $\mathbf{r}$ is the generalization X-term computed so far. The rules transform such tuples in all possible ways as long as possible, returning $\emptyset; S; \mathbf{r}$. In this case, we call $\mathbf{r}$ the *computed X-term*. We denote the algorithm by $\mathfrak{G}$. The rules are as follows:

Dec-AU: **Decomposition**

$$\begin{aligned}
&\{x : \mathbf{f}(\mathbf{t}_1, \ldots, \mathbf{t}_n) \triangleq \mathbf{g}(\mathbf{s}_1, \ldots, \mathbf{s}_n)\} \uplus A;\ S;\ \mathbf{r} \Longrightarrow\\
&\quad \{y_1 : \mathbf{t}_1 \triangleq \mathbf{s}_1, \ldots, y_n : \mathbf{t}_n \triangleq \mathbf{s}_n\} \cup A;\ S;\ \mathbf{r}\{x \mapsto (\mathbf{f} \cap \mathbf{g})(y_1, \ldots, y_n)\},
\end{aligned}$$

where $n \geq 0$, $\mathbf{f} \cap \mathbf{g} \neq \emptyset$.

Sol-AU: **Solving**

$$\{x : \mathbf{t} \triangleq \mathbf{s}\} \uplus A;\ S;\ \mathbf{r} \Longrightarrow A;\ \{x : \mathbf{t} \triangleq \mathbf{s}\} \cup S;\ \mathbf{r},$$

if $\mathit{head}(\mathbf{t}) \cap \mathit{head}(\mathbf{s}) = \emptyset$.

Mer-AU: **Merging**

$$\emptyset;\ \{x_1 : \mathbf{t}_1 \triangleq \mathbf{s}_1,\ x_2 : \mathbf{t}_2 \triangleq \mathbf{s}_2\} \uplus S;\ \mathbf{r} \Longrightarrow \emptyset;\ \{x_1 : \mathbf{t} \triangleq \mathbf{s}\} \cup S;\ \mathbf{r}\{x_2 \mapsto x_1\},$$

if $\mathbf{t} = \mathbf{t}_1 \sqcap \mathbf{t}_2 \neq \emptyset$ and $\mathbf{s} = \mathbf{s}_1 \sqcap \mathbf{s}_2 \neq \emptyset$.

Mer-AU can be applied in different ways, which might lead to multiple X-lggs. One may notice that we do not have a rule for AUTs containing variables.

This is because one can treat the input variables as constants. Then AUTs such as $x : y \triangleq y$ are dealt by the Dec-AU rule, and AUTs of the form $x : y \triangleq z$ with $y \neq z$ are processed by Sol-AU.

Theorem 5. *Given a proximity relation $\mathcal{R}$, a cut value λ, and two terms t and s, the anti-unification algorithm $\mathfrak{G}$ terminates and computes X-terms $\mathbf{r}_1, \dots, \mathbf{r}_n$, $n \geq 1$, such that $\cup_{i=1}^{n} \tau(\mathbf{r}_i)$ contains all $(\mathcal{R}, \lambda)$-least general generalizations of t and s (modulo variable renaming).*

Proof. Like Theorem 3, here also we have three parts: termination, soundness and completeness.

Termination. The algorithm obviously terminates, since the rules Dec-AU and Sol-AU strictly reduce the number of symbols in the AU-problem set A, and Mer-AU strictly reduces the number of symbols in the store S.

Soundness. We will prove that if $r \in \cup_{i=1}^{n} \tau(\mathbf{r}_i)$, then r is an $(\mathcal{R}, \lambda)$-generalization of t and s.

If $r \in \cup_{i=1}^{n} \tau(\mathbf{r}_i)$, then $r \in \tau(\mathbf{r}_j)$ for some $1 \leq j \leq n$. It means that there exists a derivation

$$\{x : \mathbf{pc}(t, \mathcal{R}, \lambda) \triangleq \mathbf{pc}(s, \mathcal{R}, \lambda)\}; \emptyset; x\boldsymbol{\vartheta}_0 \Longrightarrow^k \emptyset; S; x\boldsymbol{\vartheta}_0\boldsymbol{\vartheta}_1 \cdots \boldsymbol{\vartheta}_k, \tag{1}$$

where $\boldsymbol{\vartheta}_0 = Id$, $k \geq 1$ and $\mathbf{r}_j = x\boldsymbol{\vartheta}_0\boldsymbol{\vartheta}_1 \cdots \boldsymbol{\vartheta}_k$. For the reference, we denote the tuple at step l in this derivation by $A_l; S_l; \mathbf{r}_l$. Observe that:

– by Dec-AU rule, whenever an AUT $x' : \mathbf{t}' \triangleq \mathbf{s}'$ appears in some A_l in this derivation ($0 \leq l \leq k$), then we have $x' \in \mathcal{V}(x\boldsymbol{\vartheta}_0 \cdots \boldsymbol{\vartheta}_l)$, $\mathbf{t}' = \mathbf{pc}(t, \mathcal{R}, \lambda)|_{p'}$ for some position p' in $\mathbf{pc}(t, \mathcal{R}, \lambda)$, and $\mathbf{s}' = \mathbf{pc}(s, \mathcal{R}, \lambda)|_{p'}$ for the same position p' in $\mathbf{pc}(s, \mathcal{R}, \lambda)$;

– by Sol-AU rule, the same is true for any $x' : \mathbf{t}' \triangleq \mathbf{s}'$, which appears in some S_l in this derivation ($0 \leq l \leq k$) with $A_l \neq \emptyset$;

– by Mer-AU rule, for any AUT $x' : \mathbf{t}' \triangleq \mathbf{s}'$, which appears in some S_l in this derivation ($0 \leq l \leq k$) with $A_l = \emptyset$, we have $x' \in \mathcal{V}(x\boldsymbol{\vartheta}_0 \cdots \boldsymbol{\vartheta}_l)$, $\tau(\mathbf{t}') \subseteq \tau(\mathbf{pc}(t, \mathcal{R}, \lambda)|_{p'})$ for some position p' in $\mathbf{pc}(t, \mathcal{R}, \lambda)$, and $\tau(\mathbf{s}') \subseteq \tau(\mathbf{pc}(s, \mathcal{R}, \lambda)|_{p'})$ for the same position p' in $\mathbf{pc}(s, \mathcal{R}, \lambda)$.

Coming back to the derivation in (1), we prove that for all $0 \leq i < k$, if $x\boldsymbol{\vartheta}_0\boldsymbol{\vartheta}_1 \cdots \boldsymbol{\vartheta}_i$ is an $(\mathcal{R}, \lambda)$-X-generalization of t and s, then $x\boldsymbol{\vartheta}_0\boldsymbol{\vartheta}_1 \cdots \boldsymbol{\vartheta}_{i+1}$ is an $(\mathcal{R}, \lambda)$-X-generalization of t and s. For $i = 0$ it is obvious. We assume that this statement is true for some $0 \leq i < k$ and show it for $i + 1$. We should look at all possible ways to make the step

$$A_i; S_i; x\boldsymbol{\vartheta}_0\boldsymbol{\vartheta}_1 \cdots \boldsymbol{\vartheta}_i \Longrightarrow A_{i+1}; S_{i+1}; x\boldsymbol{\vartheta}_0\boldsymbol{\vartheta}_1 \cdots \boldsymbol{\vartheta}_{i+1}.$$

– The step is made by Dec-AU. It means that the problem set A_i contains an AUT of the form $x_i : \mathbf{f}_i(\mathbf{t}_{i_1}, \dots, \mathbf{t}_{i_{n_i}}) \triangleq \mathbf{g}_i(\mathbf{s}_{i_1}, \dots, \mathbf{s}_{i_{n_i}})$ with $\mathbf{f}_i \cap \mathbf{g}_i \neq \emptyset$, which is replaced in A_{i+1} by new AUTs $y_1 : \mathbf{t}_{i_1} \triangleq \mathbf{s}_{i_1}, \dots, y_{n_i} : \mathbf{t}_{i_{n_i}} \triangleq \mathbf{s}_{i_{n_i}}$, and $\boldsymbol{\vartheta}_{i+1} = \{x_i \mapsto (\mathbf{f}_i \cap \mathbf{g}_i)(y_1, \dots, y_{n_i})\}$. There is a position p in both $\mathbf{pc}(t, \mathcal{R}, \lambda)$ and $\mathbf{pc}(s, \mathcal{R}, \lambda)$ such that $\mathbf{pc}(t, \mathcal{R}, \lambda)|_p = \mathbf{f}_i(\mathbf{t}_{i_1}, \dots, \mathbf{t}_{i_{n_i}})$

and $\mathbf{pc}(s, \mathcal{R}, \lambda)|_p = \mathbf{g}_i(\mathbf{s}_{i_1}, \dots, \mathbf{s}_{i_{n_i}})$. In the same position p in t and s, we have respectively $t_p = f_i(t_{i_1}, \dots, t_{i_{n_i}}) \in \tau(\mathbf{f}_i(\mathbf{t}_{i_1}, \dots, \mathbf{t}_{i_{n_i}}))$ and $s_p = g_i(s_{i_1}, \dots, s_{i_{n_i}}) \in \tau(\mathbf{s}_i(\mathbf{s}_{i_1}, \dots, \mathbf{s}_{i_{n_i}}))$. Moreover, in the same p in the X-term $x\boldsymbol{\vartheta}_0\boldsymbol{\vartheta}_1 \cdots \boldsymbol{\vartheta}_i$ we have x_i and we know (by the assumption) that $x\boldsymbol{\vartheta}_0\boldsymbol{\vartheta}_1 \cdots \boldsymbol{\vartheta}_i$ is an $(\mathcal{R}, \lambda)$-generalization of t and s. Besides, by definition of X-generalization, its is obvious that $x_i\boldsymbol{\vartheta}_{i+1} = (\mathbf{f}_i \sqcap \mathbf{g}_i)(y_1, \dots, y_{n_i})$ is an $(\mathcal{R}, \lambda)$-generalization of $f_i(t_{i_1}, \dots, t_{i_{n_i}})$ and $g_i(s_{i_1}, \dots, s_{i_{n_i}})$. By replacing x_i with $x_i\boldsymbol{\vartheta}_{i+1}$, we obtain that $x\boldsymbol{\vartheta}_0\boldsymbol{\vartheta}_1 \cdots \boldsymbol{\vartheta}_i\boldsymbol{\vartheta}_{i+1}$ is an $(\mathcal{R}, \lambda)$-generalization of t and s.

– The step is made by Sol-AU. In this case, $x\boldsymbol{\vartheta}_0\boldsymbol{\vartheta}_1 \cdots \boldsymbol{\vartheta}_{i+1} = x\boldsymbol{\vartheta}_0\boldsymbol{\vartheta}_1 \cdots \boldsymbol{\vartheta}_i$ and the statement holds.

– The step is made by Mer-AU. In this case, S_i contains two AUTs $x_{i_1} : \mathbf{t}_{i_1} \triangleq \mathbf{s}_{i_1}$, $x_{i_2} : \mathbf{t}_{i_2} \triangleq \mathbf{s}_{i_2}$ with $\mathbf{t}_{i_1} \sqcap \mathbf{t}_{i_2} \neq \emptyset$ and $\mathbf{s}_{i_1} \sqcap \mathbf{s}_{i_2} \neq \emptyset$. In S_{i+1} these AUTs are replaced by a single AUT $x_{i_1} : \mathbf{t}_{i_1} \sqcap \mathbf{t}_{i_2} \triangleq \mathbf{s}_{i_1} \sqcap \mathbf{s}_{i_2}$, and $\boldsymbol{\vartheta}_{i+1} = \{x_{i_2} \mapsto x_{i_1}\}$. There are two positions p_1 and p_2 in $\mathbf{pc}(t, \mathcal{R}, \lambda)$ such that $\tau(\mathbf{t}_{i_j}) \subseteq \tau(\mathbf{pc}(t, \mathcal{R}, \lambda)|_{p_j})$, $j = 1, 2$. From $\mathbf{t}_{i_1} \sqcap \mathbf{t}_{i_2} \neq \emptyset$ we have $\tau(\mathbf{t}_{i_1}) \cap \tau(\mathbf{t}_{i_2}) \neq \emptyset$ and, as a consequence, $\tau(\mathbf{pc}(t, \mathcal{R}, \lambda)|_{p_1}) \cap \tau(\mathbf{pc}(t, \mathcal{R}, \lambda)|_{p_2}) \neq \emptyset$. Similarly, we get $\tau(\mathbf{s}_{i_1}) \cap \tau(\mathbf{s}_{i_2}) \neq \emptyset$.

Since $x\boldsymbol{\vartheta}_0\boldsymbol{\vartheta}_1 \cdots \boldsymbol{\vartheta}_i$ is an $(\mathcal{R}, \lambda)$-X-generalization of t and s, for any $q \in \tau(x\boldsymbol{\vartheta}_0\boldsymbol{\vartheta}_1 \cdots \boldsymbol{\vartheta}_i)$ there exist substitutions σ_t and σ_s such that $q\sigma_t \simeq_{\mathcal{R},\lambda} t$ and $q\sigma_s \simeq_{\mathcal{R},\lambda} s$. For σ_t, we have $x_{i_1}\sigma_t \simeq_{\mathcal{R},\lambda} t|_{p_1}$ and $x_{i_2}\sigma_t \simeq_{\mathcal{R},\lambda} t|_{p_2}$. For σ_s, we have $x_{i_1}\sigma_s \simeq_{\mathcal{R},\lambda} s|_{p_1}$ and $x_{i_2}\sigma_s \simeq_{\mathcal{R},\lambda} s|_{p_2}$.

In $x\boldsymbol{\vartheta}_0\boldsymbol{\vartheta}_1 \cdots \boldsymbol{\vartheta}_{i+1}$, we have x_{i_1} both in position p_1 and in position p_2. Let φ_t be a substitution such that $dom(\varphi_t) = dom(\sigma_t) \setminus \{x_{i_2}\}$, $x_{i_1}\varphi_t \in \tau(\mathbf{pc}(t, \mathcal{R}, \lambda)|_{p_1}) \cap \tau(\mathbf{pc}(t, \mathcal{R}, \lambda)|_{p_2})$, and $y\varphi_t = y\sigma_t$ for all $y \in dom(\varphi_t) \setminus \{x_{i_1}\}$. Such a φ_t exists, since we have shown that $\tau(\mathbf{pc}(t, \mathcal{R}, \lambda)|_{p_1}) \cap \tau(\mathbf{pc}(t, \mathcal{R}, \lambda)|_{p_2}) \neq \emptyset$. By definition, we know that every element of the set $\tau(\mathbf{pc}(t, \mathcal{R}, \lambda)|_{p_1}) \cap \tau(\mathbf{pc}(t, \mathcal{R}, \lambda)|_{p_2})$ is $(\mathcal{R}, \lambda)$-close to both $t|_{p_1}$ and $t|_{p_2}$. Hence, $x_{i_1}\varphi_t \simeq_{\mathcal{R},\lambda} t|_{p_1}$ and $x_{i_1}\varphi_t \simeq_{\mathcal{R},\lambda} t|_{p_2}$. We can define φ_s analogously, and by a similar reasoning conclude that $x_{i_1}\varphi_s \simeq_{\mathcal{R},\lambda} s|_{p_1}$ and $x_{i_1}\varphi_s \simeq_{\mathcal{R},\lambda} s|_{p_2}$. For every position other than those where x_{i_2} appeared in $x\boldsymbol{\vartheta}_0\boldsymbol{\vartheta}_1 \cdots \boldsymbol{\vartheta}_i$, the X-terms $x\boldsymbol{\vartheta}_0\boldsymbol{\vartheta}_1 \cdots \boldsymbol{\vartheta}_{i+1}$ and $x\boldsymbol{\vartheta}_0\boldsymbol{\vartheta}_1 \cdots \boldsymbol{\vartheta}_i$ coincide. Hence, for every $q \in \tau(x\boldsymbol{\vartheta}_0\boldsymbol{\vartheta}_1 \cdots \boldsymbol{\vartheta}_{i+1})$, we get $q\varphi_t \simeq_{\mathcal{R},\lambda} t$ and $q\varphi_s \simeq_{\mathcal{R},\lambda} s$, implying that $x\boldsymbol{\vartheta}_0\boldsymbol{\vartheta}_1 \cdots \boldsymbol{\vartheta}_{i+1}$ is an $(\mathcal{R}, \lambda)$-X-generalization of t and s.

Hence, we proved that in derivation (1), $x\boldsymbol{\vartheta}_0\boldsymbol{\vartheta}_1 \cdots \boldsymbol{\vartheta}_k$ is an $(\mathcal{R}, \lambda)$-X-generalization of t and s. Since $x\boldsymbol{\vartheta}_0\boldsymbol{\vartheta}_1 \cdots \boldsymbol{\vartheta}_k = \mathbf{r}_j$ with $r \in \tau(\mathbf{r}_j)$, we get that r is an $(\mathcal{R}, \lambda)$-generalization of t and s, which proves soundness.

Completeness. If r is an $(\mathcal{R}, \lambda)$-lgg of t and s, then there exists $r' \in \cup_{i=1}^{n} \tau(\mathbf{r}_i)$ such that r and r' are equal modulo variable renaming.

We prove completeness by structural induction on r.

First, assume r is a variable. Since it is an $(\mathcal{R}, \lambda)$-lgg of t and s, we have $\tau(head(\mathbf{pc}(t, \mathcal{R}, \lambda))) \cap \tau(head(\mathbf{pc}(s, \mathcal{R}, \lambda))) = \emptyset$. But in this case we apply the rule Sol-AU and get also a variable as a computed X-generalization, which may differ from r only by the name.

Now assume $r = h(r_1, \ldots, r_m)$. Then we have that $t = f(t_1, \ldots, t_m)$, $s = g(s_1, \ldots, s_m)$, and $h \in \mathbf{f} \cap \mathbf{g}$, where $\mathbf{f} = \mathbf{pc}(f, \mathcal{R}, \lambda)$ and $\mathbf{g} = \mathbf{pc}(g, \mathcal{R}, \lambda)$. We apply the rule Dec-AU to $x : \mathbf{pc}(t, \mathcal{R}, \lambda) \triangleq \mathbf{pc}(s, \mathcal{R}, \lambda)$ and obtain new AUTs $y_k : \mathbf{pc}(t_k, \mathcal{R}, \lambda) \triangleq \mathbf{pc}(s_k, \mathcal{R}, \lambda)$, $1 \leq k \leq m$. Note that each r_k, $1 \leq k \leq m$, is an $(\mathcal{R}, \lambda)$-lgg of t_k and s_k. Then by the induction hypothesis, for each $1 \leq k \leq m$ we compute $\mathbf{r}'_k$ so that there exists $r'_k \in \tau(\mathbf{r}'_k)$ which is a renamed copy of r_k. We combine the initial step Dec-AU with the derivations that compute $\mathbf{r}'_i$ to obtain a derivation computing $(\mathbf{f} \cap \mathbf{g})(\mathbf{r}'_1, \ldots, \mathbf{r}'_m)$. However, this does not yet guarantee that $(\mathbf{f} \cap \mathbf{g})(\mathbf{r}'_1, \ldots, \mathbf{r}'_m)$ contains a renamed copy of r, since by being an $(\mathcal{R}, \lambda)$-lgg, r might contain the same variable in multiple positions (in different r_i and r_j), which we have not captured yet. Let p_i and p_j be such positions in r, containing the same variable y, but having different variables y_i and y_j in $(\mathbf{f} \cap \mathbf{g})(\mathbf{r}'_1, \ldots, \mathbf{r}'_m)$. Since r is a generalization of t and s, having the same variable in p_i and p_j implies that $\tau(\mathbf{pc}(t, \mathcal{R}, \lambda)|_{p_i}) \cap \tau(\mathbf{pc}(t, \mathcal{R}, \lambda)|_{p_j}) \neq \emptyset$. Therefore, $\mathbf{pc}(t, \mathcal{R}, \lambda)|_{p_i} \sqcap \mathbf{pc}(t, \mathcal{R}, \lambda)|_{p_j} \neq \emptyset$. Similarly, we have $\mathbf{pc}(s, \mathcal{R}, \lambda)|_{p_i} \sqcap \mathbf{pc}(s, \mathcal{R}, \lambda)|_{p_j} \neq \emptyset$. Having different y_i and y_j in positions p_i and p_j in $(\mathbf{f} \cap \mathbf{g})(\mathbf{r}'_1, \ldots, \mathbf{r}'_m)$ implies that we have $y_i : \mathbf{pc}(t, \mathcal{R}, \lambda)|_{p_i} \triangleq \mathbf{pc}(s, \mathcal{R}, \lambda)|_{p_i}$ and $y_j : \mathbf{pc}(t, \mathcal{R}, \lambda)|_{p_j} \triangleq \mathbf{pc}(s, \mathcal{R}, \lambda)|_{p_j}$ in the store in the derivation we just constructed. But then we can extend this derivation by applying Mer-AU rule for y_i and y_j obtaining $(\mathbf{f} \cap \mathbf{g})(\mathbf{r}'_1, \ldots, \mathbf{r}'_i, \ldots, \mathbf{r}''_j, \ldots, \mathbf{r}'_m)$ which reduces the difference with r in distinct variables. We can repeat these steps as long as there are positions which contain different variables in the generalization computed by us, and the same variable in r. In this way, we obtain an X-generalization $\mathbf{r}'$ of t and s such that there exists $r' \in \tau(\mathbf{r}')$ which is a renamed copy of r. □

Hence, the algorithm computes $(\mathcal{R}, \lambda)$-X-lggs of the given terms. To compute linear generalizations, we do not need the Mer-AU rule. In this case the anti-unification algorithm returns a single X-term $\mathbf{r}$ such that $\tau(\mathbf{r})$ contains all linear lggs of s and t (modulo variable renaming).

Example 8. Let $\mathcal{R}$ be a proximity relation defined as

$$\mathcal{R}(a_1, a) = \mathcal{R}(a_2, a) = \mathcal{R}(b_1, b) = \mathcal{R}(b_2, b) = 0.5,$$
$$\mathcal{R}(a_2, a') = \mathcal{R}(a_3, a') = \mathcal{R}(b_2, b') = \mathcal{R}(b_3, b') = 0.6, \qquad \mathcal{R}(f, g) = 0.7.$$

Let $t = f(a_1, a_2, a_3)$ and $s = g(b_1, b_2, b_3)$. Then the anti-unification algorithm run ends with the following pairs consisting of the store and an $(\mathcal{R}, \lambda)$-lgg, for different values of λ:

$0 < \lambda \leq 0.5$: $\text{store}_1 = \{x_1 : \{a\} \triangleq \{b\},\ x_3 : \{a_3, a'\} \triangleq \{b_3, b'\}\}$,
X-$\text{lgg}_1 = \{f, g\}(x_1, x_1, x_3)$.

$\text{store}_2 = \{x_1 : \{a_1, a\} \triangleq \{b_1, b\},\ x_2 : \{a'\} \triangleq \{b'\}\}$,
X-$\text{lgg}_2 = \{f, g\}(x_1, x_2, x_2)$.

$0.5 < \lambda \leq 0.6$: $\text{store} = \{x_1 : \{a_1\} \triangleq \{b_1\},\ x_2 : \{a'\} \triangleq \{b'\}\}$,

$$\text{X-lgg} = \{f, g\}(x_1, x_2, x_2).$$

$$0.6 < \lambda \le 0.7: \quad \text{store} = \{x_1 : \{a_1\} \triangleq \{b_1\},\ x_2 : \{a_2\} \triangleq \{b_2\},\ x_3 : \{a_3\} \triangleq \{b_3\}\},$$
$$\text{X-lgg} = \{f, g\}(x_1, x_2, x_3).$$

$$0.7 < \lambda \le 1: \quad \text{store} = \{x : \{f(a_1, a_2, a_3)\} \triangleq g(b_1, b_2, b_3)\}, \quad \text{X-lgg} = x.$$

The store shows how to obtain terms which are $(\mathcal{R}, \lambda)$-close to the original terms. For instance, when $0 < \lambda \le 0.5$, store$_1$ tells us that for any substitution σ from the set $\tau(\{x_1 \mapsto \{a\}, x_3 \mapsto \{a_3, a'\}\})$, the instances of the generalizations, $f(x_1, x_1, x_3)\sigma$ and $g(x_1, x_1, x_3)\sigma$ are $(\mathcal{R}, \lambda)$-close to the original term t. We have two such σ's, $\sigma_1 = \{x_1 \mapsto a, x_3 \mapsto a_3\}$ and $\sigma_2 = \{x_1 \mapsto a, x_3 \mapsto a'\}$. They give, respectively, $f(x_1, x_1, x_3)\sigma_1 = f(a, a, a_3) \simeq_{\mathcal{R},\lambda} f(a_1, a_2, a_3)$, $g(x_1, x_1, x_3)\sigma_1 = g(a, a, a_3) \simeq_{\mathcal{R},\lambda} f(a_1, a_2, a_3)$, and $f(x_1, x_1, x_3)\sigma_2 = f(a, a, a') \simeq_{\mathcal{R},\lambda} f(a_1, a_2, a_3)$, $g(x_1, x_1, x_3)\sigma_2 = g(a, a, a') \simeq_{\mathcal{R},\lambda} f(a_1, a_2, a_3)$.

Similarly, for any substitution ϑ from the set $\tau(\{x_1 \mapsto \{b\}, x_3 \mapsto \{b_3, b'\}\})$ (which is also extracted from store$_1$), the instances of the generalizations $f(x_1, x_1, x_3)\vartheta$ and $g(x_1, x_1, x_3)\vartheta$ are $(\mathcal{R}, \lambda)$-close to the original term s.

Now we illustrate how the first two X-lggs have been computed. Let $\lambda = 0.5$. For the initial problem we take $\mathbf{pc}(t, \mathcal{R}, \lambda) = \{f, g\}(\{a_1, a\}, \{a_2, a, a'\}, \{a_3, a'\})$ and $\mathbf{pc}(s, \mathcal{R}, \lambda) = \{g, f\}(\{b_1, b\}, \{b_2, b, b'\}, \{b_3, b'\})$ and proceed as follows:

$$\{x : \{f, g\}(\{a_1, a\}, \{a_2, a, a'\}, \{a_3, a'\}) \triangleq \{g, f\}(\{b_1, b\}, \{b_2, b, b'\}, \{b_3, b'\})\};$$
$$\emptyset; x \Longrightarrow_{\text{Dec-AU}}$$
$$\{x_1 : \{a_1, a\} \triangleq \{b_1, b\}, x_2 : \{a_2, a, a'\} \triangleq \{b_2, b, b'\}, x_3 : \{a_3, a'\} \triangleq \{b_3, b'\}\};$$
$$\emptyset; \{f, g\}(x_1, x_2, x_3) \Longrightarrow_{\text{Sol-AU}\times 3}$$
$$\emptyset; \{x_1 : \{a_1, a\} \triangleq \{b_1, b\}, x_2 : \{a_2, a, a'\} \triangleq \{b_2, b, b'\}, x_3 : \{a_3, a'\} \triangleq \{b_3, b'\}\};$$
$$\{f, g\}(x_1, x_2, x_3).$$

Now there are two alternatives: to merge x_1 and x_2, or x_2 and x_3. They give:

$$\emptyset;\ \{x_1 : \{a\} \triangleq \{b\}, x_3 : \{a_3, a'\} \triangleq \{b_3, b'\}\};\ \{f, g\}(x_1, x_1, x_3), \quad \text{or}$$
$$\emptyset;\ \{x_1 : \{a_1, a\} \triangleq \{b_1, b\}, x_2 : \{a'\} \triangleq \{b'\}\};\ \{f, g\}(x_1, x_2, x_2).$$

These are exactly the stores and X-lggs we saw at the beginning of this example.

Example 9. Consider again the proximity relation and the terms from Example 8, but this times assume we are interested in linear generalizations. Then the stores and X-lggs are the following:

$$0 < \lambda \le 0.5:$$
$$\text{store} = \{x_1 : \{a_1, a\} \triangleq \{b_1, b\}, x_2 : \{a_2, a, a'\} \triangleq \{b_2, b, b'\},$$
$$x_3 : \{a_3, a'\} \triangleq \{b_3, b'\}\}.$$

$$\text{X-lgg} = \{f, g\}(x_1, x_2, x_3).$$

$0.5 < \lambda \leq 0.6$:

$$\text{store} = \{x_1 : \{a_1\} \triangleq \{b_1\}, x_2 : \{a_2, a'\} \triangleq \{b_2, b'\},$$
$$x_3 : \{a_3, a'\} \triangleq \{b_3, b'\}\}.$$
$$\text{X-lgg} = \{f, g\}(x_1, x_2, x_3).$$

$0.6 < \lambda \leq 0.7$:

$$\text{store} = \{x_1 : \{a_1\} \triangleq \{b_1\},\ x_2 : \{a_2\} \triangleq \{b_2\}, x_3 : \{a_3\} \triangleq \{b_3\}\}.$$
$$\text{X-lgg} = \{f, g\}(x_1, x_2, x_3).$$

$0.7 < \lambda \leq 1$: $\text{store} = \{x : \{f(a_1, a_2, a_3)\} \triangleq g(b_1, b_2, b_3)\}$. $\text{X-lgg} = x$.

Theorem 6. *Let $\mathcal{R} = (V_\mathcal{R}, E_\mathcal{R})$ be a proximity relation and λ be a cut value. Assume t and s are terms with size(s) + size$(t) = n$. Then*

- *$\mathfrak{G}$ needs $O(n^2|V_\mathcal{R}| + |E_\mathcal{R}|)$ time and $O(n|V_\mathcal{R}| + |E_\mathcal{R}|)$ space to compute a single $(\mathcal{R}, \lambda)$-X-lgg of t and s;*
- *$\mathfrak{G}$ needs $O(n|V_\mathcal{R}| + |E_\mathcal{R}|)$ time and space to compute a linear $(\mathcal{R}, \lambda)$-X-lgg of t and s.*

Proof. To represent the relation $\mathcal{R}$, we use adjacency lists in the same way as we did for the matching algorithm (see the proof of Theorem 4). For adjacency lists, the required amount of memory is $O(|V_\mathcal{R}| + |E_\mathcal{R}|)$. The input can be represented as trees in $O(n)$ space. The same amount is needed for the store. The generalization X-term contains $O(n)$ nodes, each labeled with at most $|V_\mathcal{R}|$ symbols. Hence, the total space requirement is $O(n|V_\mathcal{R}| + |E_\mathcal{R}|)$, and it is independent whether we compute a single $(\mathcal{R}, \lambda)$-X-lgg or a linear $(\mathcal{R}, \lambda)$-X-lgg.

As for the runtime complexity, constructing the adjacency list representation is done in $O(|V_\mathcal{R}| + |E_\mathcal{R}|)$ time. Besides, whenever Dec-AU or Sol-AU is applied, we need to compute the intersection between proximity classes of two function symbols, which needs $O(|V_\mathcal{R}|)$ time. Hence, applying these two rules as long as possible requires $O(n|V_\mathcal{R}|)$ time. It implies that the runtime complexity for computing linear $(\mathcal{R}, \lambda)$-X-lgg of t and s is $O(n|V_\mathcal{R}| + |E_\mathcal{R}|)$.

To compute an unrestricted $(\mathcal{R}, \lambda)$-X-lgg, we should further apply Mer-AU as long as possible. This may require $O(n^2)$ steps. At each step we perform the intersection of proximity classes which is done in $O(|V_\mathcal{R}|)$ time. Therefore, exhaustive application of Mer-AU for computing one $(\mathcal{R}, \lambda)$-X-lgg of t and s needs $O(n^2|V_\mathcal{R}|)$. Together with the complexity of maximal applications of the Dec-AU or Sol-AU rules considered above, it gives the $O(n^2|V_\mathcal{R}| + |E_\mathcal{R}|)$ runtime bound for computing a single $(\mathcal{R}, \lambda)$-X-lgg of t and s. $\square$

5.1 Computing Approximation Degrees for Anti-Unification

We can incorporate the approximation degree computation in anti-unification easier than we did for matching. To $(\mathcal{R}, \lambda)$-anti-unify t and s, we just take their graded proximity classes $\mathbf{pc}(t, \mathcal{R}, \lambda)$ and $\mathbf{pc}(s, \mathcal{R}, \lambda)$ and run the algorithm as described above. The operations $\cap$ and $\sqcap$ will be performed on graded sets of functions symbols and graded X-terms, respectively.

Example 10. Let $\mathcal{R}$ be a proximity relation from Example 8 with $\mathcal{R}(f, g) = 0.7$ replaced by $\mathcal{R}(f, h) = 0.7$ and $\mathcal{R}(h, g) = 0.8$. Let $t = f(a_1, a_2, a_3)$ and $s = g(b_1, b_2, b_3)$. Then for $0 < \lambda \leq 0.5$ we get

$$\begin{aligned}
\text{store}_1 = \{&x_1 : \{\langle a, 0.5\rangle\} \triangleq \{\langle b, 0.5\rangle\},\\
&x_3 : \{\langle a_3, 1\rangle, \langle a', 0.6\rangle\} \triangleq \{\langle b_3, 1\rangle, \langle b', 0.6\rangle\}\},\\
\text{X-lgg}_1 = \{&\langle h, 0.7\rangle\}(x_1, x_1, x_3).
\end{aligned}$$

$$\begin{aligned}
\text{store}_2 = \{&x_1 : \{\langle a_1, 1\rangle, \langle a, 0.5\rangle\} \triangleq \{\langle b_1, 1\rangle, \langle b, 0.5\rangle\},\\
&x_2 : \{\langle a', 0.6\rangle\} \triangleq \{\langle b', 0.6\rangle\}\},\\
\text{X-lgg}_2 = \{&\langle h, 0.7\rangle\}(x_1, x_2, x_2).
\end{aligned}$$

From the X-lgg's we get the actual generalizations. For instance, X-lgg$_2$ gives $r = h(x_1, x_2, x_2)$. From the generalizations, we can "get close" to the original terms by applying the substitutions composed from the store: $\mathcal{R}(r\{x_1 \mapsto a_1, x_2 \mapsto a'\}, t) = \mathcal{R}(h(a_1, a', a'), t) = 0.6$ and $\mathcal{R}(r\{x_1 \mapsto b_1, x_2 \mapsto b'\}, s) = \mathcal{R}(h(b_1, b', b'), s) = 0.6$. Another instance would be $\mathcal{R}(r\{x_1 \mapsto a, x_2 \mapsto a'\}, t) = \mathcal{R}(h(a, a', a'), t) = 0.5$ and $\mathcal{R}(r\{x_1 \mapsto b, x_2 \mapsto b'\}, s) = \mathcal{R}(h(b, b', b'), s) = 0.5$.

6 Conclusion

In this paper, we investigated two fundamental matching and anti-unification problems with fuzzy proximity relations. Fuzzy proximity (and its crisp counterpart, tolerance) is not a transitive relation, which makes these problems challenging. In general, there is no single solution to them.

We developed algorithms that solve the mentioned problems, aiming at computing a compact representation of solution sets. We use extended terms (X-terms) to represent term sets. In X-terms, instead of function symbols, finite sets of function symbols are permitted. X-substitutions map variables to X-terms.

Our matching algorithm computes a single X-substitution solution for solvable proximity (and tolerance) matching problems. We prove that it is sound and complete: every standard substitution obtained from the computed X-matcher is a matcher, and any relevant solution of the matching problem is contained in the set of substitutions induced by the computed X-matcher. Time and space complexities of the algorithm are analyzed.

Unlike matching, proximity/tolerance anti-unification problems, in general, do not have a single solution even if we restrict computed least-general generalizations to X-terms. Our anti-unification algorithm computes a finite complete set of X-lggs. If we consider the linear variant (i.e., if generalizations are not permitted to contain more than one occurrence of each generalization variable), then there exists a single linear X-lgg (which still represents a finite set of lggs as standard terms), and our algorithm computes it. We also analyze time and space complexities of our anti-unification algorithm and its linear variant.

References

1. H. Aït-Kaci and G. Pasi. Fuzzy unification and generalization of first-order terms over similar signatures. In F. Fioravanti and J. P. Gallagher, editors, *Logic-Based Program Synthesis and Transformation - 27th Int. Symposium, LOPSTR 2017, Revised Selected Papers*, volume 10855 of *LNCS*, pages 218–234. Springer, 2017.
2. M. Alpuente, S. Escobar, J. Espert, and J. Meseguer. A modular order-sorted equational generalization algorithm. *Inf. Comput.*, 235:98–136, 2014.
3. F. Baader and T. Nipkow. *Term rewriting and all that.* Cambridge University Press, 1998.
4. F. Baader and W. Snyder. Unification theory. In J. A. Robinson and A. Voronkov, editors, *Handbook of Automated Reasoning (in 2 volumes)*, pages 445–532. Elsevier and MIT Press, 2001.
5. W. Bartol, J. Miró, K. Pióro, and F. Rosselló. On the coverings by tolerance classes. *Inf. Sci.*, 166(1-4):193–211, 2004.
6. A. Baumgartner and T. Kutsia. A library of anti-unification algorithms. In E. Fermé and J. Leite, editors, *Logics in Artificial Intelligence - 14th European Conference, JELIA 2014*, volume 8761 of *LNCS*, pages 543–557. Springer, 2014.
7. T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms, 3rd Edition.* MIT Press, 2009.
8. P. Julián-Iranzo and C. Rubio-Manzano. Proximity-based unification theory. *Fuzzy Sets and Systems*, 262:21–43, 2015.
9. P. Julián-Iranzo and F. Sáenz-Pérez. An efficient proximity-based unification algorithm. In *2018 IEEE International Conference on Fuzzy Systems, FUZZ-IEEE 2018, Rio de Janeiro, Brazil, July 8-13, 2018*, pages 1–8. IEEE, 2018.
10. E. Klement, R. Mesiar, and E. Pap. *Triangular Norms*, volume 8 of *Trends in Logic*. Springer, 2000.
11. T. Kutsia and C. Pau. Proximity-based generalization. In M. Ayala Rincón and P. Balbiani, editors, *32nd International Workshop on Unification, UNIF 2018, Proceedings*, 2018.
12. T. Kutsia and C. Pau. Computing all maximal clique partitions in a graph. RISC Report 19-04, RISC, Johannes Kepler University Linz, 2019.
13. T. Kutsia and C. Pau. Solving proximity constraints. In M. Gabbrielli, editor, *Logic-Based Program Synthesis and Transformation - 29th International Symposium, LOPSTR 2019, Revised Selected Papers*, volume 12042 of *LNCS*, pages 107–122. Springer, 2019.
14. G. D. Plotkin. A note on inductive generalization. *Machine Intel.*, 5(1):153–163, 1970.
15. J. C. Reynolds. Transformational systems and the algebraic structure of atomic formulas. *Machine Intel.*, 5(1):135–151, 1970.
16. M. I. Sessa. Approximate reasoning by similarity-based SLD resolution. *Theor. Comput. Sci.*, 275(1-2):389–426, 2002.
17. E. C. Zeeman. Topology of the brain. In C. H. Waddington, editor, *Towards a Theoretical Biology*, volume 1, pages 140–151. Edinburgh University Press, 1965.