Matching and Generalization Modulo Proximity and Tolerance

Temur Kutsia and Cleo Pau

RISC, Johannes Kepler University Linz, Austria

Abstract. Proximity relations are fuzzy binary relations satisfying fuzzy 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 study two fundamental symbolic computation algorithms for proximity and tolerance relations: matching and anti-unification, and briefly characterize their properties.

1 Preliminaries

Background. Proximity relations are reflexive and symmetric fuzzy binary relations. They generalize similarity relations, which are a fuzzy version of equivalences. The crisp counterpart of proximity is tolerance, which generalizes the standard equivalence relation by dropping the transitivity property. Proximity relations help to represent fuzzy information in situations where similarity is not adequate.

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 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 [3], for syntactic and equational anti-unification see, e.g., [2,4,7,8]. Unification and anti-unification modulo similarity have been investigated, e.g., in [1,9].

On the other hand, there are very few works on unification and anti-unification modulo proximity and tolerance. In [5], the authors introduced the notion of proximity-based unification and used it in fuzzy logic programming under a certain restriction imposed on the proximity relation. In [6], we reported the first results on proximity-based anti-unification with the same restriction.

In this paper we consider the general case: anti-unification for a proximity relation without any restriction. We develop an algorithm which computes a compact representation of the set of least general generalizations. The algorithm directly extends to tolerance relations. A potential application includes, e.g., an extension of software code clone detection methods by treating certain mismatches as similar.

Anti-unification is closely related to matching, which is a special case of unification. Generalizations (whose computation is the goal of anti-unification) are supposed to match the original terms. We show that, in general, matching problems with proximity or tolerance relations might have finitely many incomparable solutions, but we also come up with an algorithm which computes a compact representation of the set of solutions. Unlike anti-unification, where the solution set is represented by several incomparable representations, for matching problems we always have a single answer.

Proximity and tolerance relations. We define basic notions about proximity relations following [5]. 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 $0 \leq \lambda \leq 1$, then the λ-*cut* of $\mathcal{R}$ on S, denoted $\mathcal{R}_\lambda$, is an ordinary (crisp) relation on S defined as $\mathcal{R}_\lambda := \{(s_1, s_2) \mid \mathcal{R}(s_1, s_1) \geq \lambda\}$. In the role of T-norm $\wedge$ we take the minimum.

A fuzzy relation $\mathcal{R}$ on a set S is called a *proximity relation* on S iff it is reflexive and symmetric. The λ-cut of a proximity relation on S is a *tolerance* (i.e., a reflexive and symmetric) relation on S. The *proximity class of level* λ *of* $s \in S$ (a λ-*class* of s) is a set $\mathbf{pc}(s, \mathcal{R}, \lambda) = \{s' \mid \mathcal{R}(s, s') \geq \lambda\}$. When $\mathcal{R}$ and λ are fixed, we simply write $\mathbf{pc}(s)$. a

Terms. Given two 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 the notation for finite sets of function symbols, whose all elements have the same arity. They will be denoted by bold face letters.

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 funitely 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 are used in X-terms in the same way as in 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 *head* of an (X)-term is defined as $head(x) := x$, $head(f(t_1, \ldots, t_n)) := f$, and $head(\mathbf{f}(\mathbf{t}_1, \ldots, \mathbf{t}_n)) := \mathbf{f}$. 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 $\mathbf{t} \sqcap \mathbf{s}$:

- $\mathbf{t} \sqcap \mathbf{s} = \emptyset$, if $\tau(head(\mathbf{t})) \cap \tau(head(\mathbf{s})) = \emptyset$. $\quad x \sqcap x = x$ 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 $\mathbf{t} = \mathbf{g}(\mathbf{s}_1, \ldots, \mathbf{s}_n)$. Otherwise $\mathbf{t} \sqcap \mathbf{s} = \emptyset$.

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. We use the usual set notation for substitutions, writing, e.g., σ as $\sigma = \{x \mapsto \sigma(x) \mid x \neq \sigma(x)\}$. The *set of substitutions represented by an X-term substitution* $\boldsymbol{\sigma}$ is the set $\tau(\boldsymbol{\sigma}) := \{\sigma \mid \sigma(x) \in \boldsymbol{\sigma}(x) \text{ for all } x \in \mathcal{V}\}$.

Proximity relations over terms and substitutions. Each proximity relation $\mathcal{R}$ we consider in this paper obeys the following restrictions: (a) It is defined on $\mathcal{F} \cup \mathcal{V}$; (b) $\mathcal{R}(f, g) = 0$ if $arity(f) \neq arity(g)$; (c) $\mathcal{R}(f, x) = 0$ if $f \in \mathcal{F}$ and $x \in \mathcal{V}$, (d) $\mathcal{R}(x, y) = 0$ for $x \neq y$ and $\mathcal{R}(x, x) = 1$ for all $x, y \in \mathcal{V}$. We extend such a relation $\mathcal{R}$ to terms: (i) $\mathcal{R}(s, t) := 0$ if $\mathcal{R}(head(s), head(t)) = 0$. (ii) $\mathcal{R}(s, t) := 1$ if $s = t$ and $s, t \in \mathcal{V}$. (iii) $\mathcal{R}(s, t) := \mathcal{R}(f, g) \wedge \mathcal{R}(s_1, t_1) \wedge \cdots \wedge \mathcal{R}(s_n, t_n)$, if $s = f(s_1, \ldots, s_n)$, $t = g(t_1, \ldots, t_n)$.

Two terms s and t are $(\mathcal{R}, \lambda)$-*close* to each other, written $s \simeq_{\mathcal{R},\lambda} t$, if $\mathcal{R}(s, t) \geq \lambda$. We say that s is $(\mathcal{R}, \lambda)$-*more general than* t (or t is an $(\mathcal{R}, \lambda)$-*instance* of s) and write $s \preceq_{\mathcal{R},\lambda} t$, if there exists a term substitution σ such that $s\sigma \simeq_{\mathcal{R},\lambda} t$. (We say that σ is an $(\mathcal{R}, \lambda)$-matcher of s to t.) This relation can be defined for term substitutions as well: $\sigma \preceq_{\mathcal{R},\lambda} \vartheta$ iff there exists φ such that $x\sigma\varphi \simeq_{\mathcal{R},\lambda} x\vartheta$ for all x. The strict part of $\preceq_{\mathcal{R},\lambda}$ is denoted by $\prec_{\mathcal{R},\lambda}$.

A term r is called an $(\mathcal{R}, \lambda)$-*least general generalization* ($(\mathcal{R}, \lambda)$-*lgg*) of t and s iff r is $(\mathcal{R}, \lambda)$-more general than t and s and there is no r' such that $r \prec_{\mathcal{R},\lambda} r'$ and r' is $(\mathcal{R}, \lambda)$-more general than t and s.

Given a term t, $\mathcal{R}$, and λ, the $(\mathcal{R}, \lambda)$-*proximity class* of t is an X-term $\mathbf{pc}(t, \mathcal{R}, \lambda)$, defined as $\mathbf{pc}(x, \mathcal{R}, \lambda) := \{x\}$ and $\mathbf{pc}(f(t_1, \ldots, t_n), \mathcal{R}, \lambda) := \mathbf{pc}(f, \mathcal{R}, \lambda)(\mathbf{pc}(t_1, \mathcal{R}, \lambda), \ldots, \mathbf{pc}(t_n, \mathcal{R}, \lambda))$

Theorem 1. *Given $\mathcal{R}, \lambda, t$ 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.*

2 Matching and anti-unification problems and algorithms

Matching and anti-unification problem for terms are formulated as follows: Given $\mathcal{R}, \lambda, s, t$, find $(\mathcal{R}, \lambda)$-matcher of s to t (the matching problem) or an $(\mathcal{R}, \lambda)$-lgg of s and t (the anti-unification problem). Below we develop algorithms to solve these problems. Each of them have 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	anti-unification
Given:	$\mathcal{R}, \lambda, s, t$	$\mathcal{R}, \lambda, s, t$
Find:	an X-substitution $\boldsymbol{\sigma}$ such that each $\sigma \in \tau(\boldsymbol{\sigma})$ is an $(\mathcal{R}, \lambda)$-matcher of s to t	an X-term $\mathbf{r}$ such that each $r \in \tau(\mathbf{r})$ is an $(\mathcal{R}, \lambda)$-lgg of s and t

Such a reformulation will help us to compute a single X-substitution instead of multiple matchers, and fewer X-lggs compared to 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.

Given $\mathcal{R}$, λ, s, and t (where t does not contain variables), to solve an $(\mathcal{R}, \lambda)$-matching problem $s \ll t$, we create the initial pair $\{s \ll t\}; \emptyset$ 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 semi-solved equations of the form $x \approx \mathbf{t}$. The rules 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 $\{x \mapsto \mathbf{t} \mid x \approx \mathbf{t} \in S\}$. We call it the *computed X-substitution*. The rules are as follows (the notation for overlined expressions $\overline{exp_n}$ abbreviates the sequence $exp_1, \ldots, exp_n$):

Decomposition: $\{f(\overline{t_n}) \ll g(\overline{s_n})\} \uplus M; S \Longrightarrow M \cup \{\overline{t_i \ll s_i}\}; S$, where $n \geq 0$ and $\mathcal{R}(f, g) \geq \lambda$.

Clash: $\{f(\overline{t_i}) \ll g(\overline{s_i}\} \uplus M;\ S \Longrightarrow \bot$, where $\mathcal{R}(f, g) < \lambda$.

Elimination: $\{x \ll t\} \uplus M; S \Longrightarrow M; S \cup \{x \approx \mathbf{pc}(t)\}$.

Combination: $\emptyset; \{x \approx \mathbf{t}_1, x \approx \mathbf{t}_2\} \uplus S \Longrightarrow \emptyset; S \cup \{x \approx \mathbf{t}_1 \sqcap \mathbf{t}_2\}$, if $\mathbf{t}_1 \sqcap \mathbf{t}_2 \neq \emptyset$.

Inconsistency: $\emptyset; \{x \approx \mathbf{t}_1, x \approx \mathbf{t}_2\} \uplus S \Longrightarrow \bot$, if $\mathbf{t}_1 \sqcap \mathbf{t}_2 = \emptyset$.

Theorem 2. *Given an $(\mathcal{R}, \lambda)$-matching problem $s \ll t$, the matching algorithm terminates and computes an X-substitution $\boldsymbol{\sigma}$ such that $\tau(\boldsymbol{\sigma})$ contains all $(\mathcal{R}, \lambda)$-matchers of s to t.*

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

Decompose: $\{x : \mathbf{f}(\overline{\mathbf{t}_n}) \triangleq \mathbf{g}(\overline{\mathbf{s}_n})\} \uplus A;\ S;\ \mathbf{r} \Longrightarrow \{\overline{y_n : \mathbf{t}_n \triangleq \mathbf{s}_n}\} \cup A;\ S;\ \{x \mapsto (\mathbf{f} \sqcap \mathbf{g})(\overline{y_n})\}(\mathbf{r})$, where $n \geq 0$ and $\mathbf{f} \sqcap \mathbf{g} \neq \emptyset$.

Solve: $\{x : \mathbf{t} \triangleq \mathbf{s}\} \uplus A;\ S;\ \mathbf{r} \Longrightarrow A;\ \{x : \mathbf{t} \triangleq \mathbf{s}\} \cup S;\ \mathbf{r}$, if $head(\mathbf{t}) \cap head(\mathbf{s}) = \emptyset$.

Merge: $\emptyset;\ \{x_1 : \mathbf{t}_1 \triangleq \mathbf{s}_1, x_2 : \mathbf{t}_2 \triangleq \mathbf{s}_2\} \uplus S;\ \mathbf{r} \Longrightarrow \emptyset;\ \{x_1 : \mathbf{t} \triangleq \mathbf{s}\} \cup S;\ \{x_2 \mapsto x_1\}(\mathbf{r})$, if $\mathbf{t} = \mathbf{t}_1 \sqcap \mathbf{t}_2 \neq \emptyset$ and $\mathbf{s} = \mathbf{s}_1 \sqcap \mathbf{s}_2 \neq \emptyset$.

Note that Merge can be applied in different ways, which might lead to multiple X-lggs.

Theorem 3. *Given $\mathcal{R}, \lambda, s$, and t, the described anti-unification algorithm terminates and computes X-terms $\mathbf{r}_1, \ldots, \mathbf{r}_n$, $n \geq 1$, such that $\cup_{i=1}^{n} \tau(\mathbf{r})$ contains all lggs of s and t.*

To compute linear generalizations, we do not need the Merge 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.

3 Examples

Example 1. Let $\mathcal{R}(g_1, g_2) = \mathcal{R}(a_1, a_2) = 0.5$, $\mathcal{R}(g_1, h_1) = \mathcal{R}(g_2, h_1) = 0.6$, $\mathcal{R}(g_1, h_2) = \mathcal{R}(a_1, b) = 0.7$, $\mathcal{R}(g_2, h_2) = \mathcal{R}(a_2, b) = 0.8$. Let $f(x, x) \ll f(g_1(a_1), g_2(a_2))$ be an $(\mathcal{R}, \lambda)$-matching problem. Then the matching algorithm returns the following answers for different values of λ:

$$0 < \lambda \leq 0.5: \quad \{x \mapsto \{g_1, g_2, h_1, h_2\}(\{a_1, a_2, b\})\}. \qquad 0.5 < \lambda \leq 0.6: \quad \{x \mapsto \{h_1, h_2\}(\{b\})\}.$$
$$0.6 < \lambda \leq 0.7: \quad \{x \mapsto \{h_2\}(\{b\})\}. \qquad 0.7 < \lambda \leq 1: \quad \bot.$$

Example 2. Let $\mathcal{R}(f, g) = 0.7$, $\mathcal{R}(a_1, a) = \mathcal{R}(a_2, a) = \mathcal{R}(b_1, b) = \mathcal{R}(b_2, b) = 0.5$, $\mathcal{R}(a_2, a') = \mathcal{R}(a_3, a') = \mathcal{R}(b_2, b') = \mathcal{R}(b_3, b') = 0.6$. Let $s = f(a_1, a_2, a_3)$ and $t = g(b_1, b_2, b_3)$. Then the anti-unification algorithm run ends with the following store-$(\mathcal{R}, \lambda)$-lgg pairs for different values of λ:

$$\begin{array}{rll}
0 < \lambda \leq 0.5: & \text{store}_1 = \{x_1 : \{a\} \triangleq \{b\},\ x_3 : \{a_3, a'\} \triangleq \{b_3, b'\}\}, & \text{X-lgg}_1 = \{f, g\}(x_1, x_1, x_3). \\
& \text{store}_2 = \{x_1 : \{a_1, a\} \triangleq \{b_1, b\},\ x_2 : \{a'\} \triangleq \{b'\}\}, & \text{X-lgg}_2 = \{f, g\}(x_1, x_2, x_2). \\
0.5 < \lambda \leq 0.6: & \text{store} = \{x_1 : \{a_1\} \triangleq \{b_1\},\ x_2 : \{a'\} \triangleq \{b'\}\}, & \text{X-lgg} = \{f, g\}(x_1, x_2, x_2). \\
0.6 < \lambda \leq 0.7: & \text{store} = \{x_1 : \{a_1\} \triangleq \{b_1\},\ x_2 : \{a_2\} \triangleq \{b_2\}, & \text{X-lgg} = \{f, g\}(x_1, x_2, x_3). \\
& \qquad x_3 : \{a_3\} \triangleq \{b_3\}\}, & \\
0.7 < \lambda \leq 1: & \text{store} = \{x : \{f(a_1, a_2, a_3)\} \triangleq g(b_1, b_2, b_3)\}, & \text{X-lgg} = x.
\end{array}$$

The store tells us how to obtain terms which are $(\mathcal{R}, \lambda)$-close to the original terms. For instance, when $0 < \lambda \leq 0.5$, the store tells us that for any substitution σ from the set $\tau(\{x_1 \mapsto \{a\}, x_3 \mapsto \{a_3, a'\}\})$, each of the instances of lggs $\sigma(f(x_1, x_1, x_3))$ and $\sigma(g(x_1, x_1, x_3))$ will be $(\mathcal{R}, \lambda)$-close to the original term s, i.e., $\sigma(f(x_1, x_1, x_3)) \simeq_{\mathcal{R},\lambda} s$ and $\sigma(g(x_1, x_1, x_3)) \simeq_{\mathcal{R},\lambda} s$. Similarly, from the same store we read that for any $\vartheta \in \tau(\{x_1 \mapsto \{b\}, x_3 \mapsto \{b_3, b'\}\})$, we have $\vartheta(f(x_1, x_1, x_3)) \simeq_{\mathcal{R},\lambda} t$ and $\vartheta(g(x_1, x_1, x_3)) \simeq_{\mathcal{R},\lambda} t$.

For linear generalizations, we will get one answer for each fixed λ, which happens to be the same for all $\lambda' s$ up to 0.7: $\{f, g\}(x_1, x_2, x_3)$. The corresponding stores and the remaining case are given below:

$$\begin{array}{ll}
0 < \lambda \leq 0.5: & \text{store} = \{x_1 : \{a_1, a\} \triangleq \{b_1, b\}, x_2 : \{a_2, a, a'\} \triangleq \{b_2, b, b'\}, x_3 : \{a_3, a'\} \triangleq \{b_3, b'\}\}. \\
0.5 < \lambda \leq 0.6: & \text{store} = \{x_1 : \{a_1\} \triangleq \{b_1\}, x_2 : \{a_2, a'\} \triangleq \{b_2, b'\}, x_3 : \{a_3, a'\} \triangleq \{b_3, b'\}\}. \\
0.6 < \lambda \leq 0.7: & \text{store} = \{x_1 : \{a_1\} \triangleq \{b_1\}, x_2 : \{a_2\} \triangleq \{b_2\}, x_3 : \{a_3\} \triangleq \{b_3\}\}. \\
0.7 < \lambda \leq 1: & \text{store} = \{x : \{f(a_1, a_2, a_3)\} \triangleq g(b_1, b_2, b_3)\}, \quad \text{X-lgg} = x.
\end{array}$$

Acknowledgments. This work was supported by the Austrian Science Fund (FWF) under project 28789-N32.

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 International Symposium, LOPSTR 2017, Revised Selected Papers*, volume 10855 of *Lecture Notes in Computer Science*, 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 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.
4. 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 *Lecture Notes in Computer Science*, pages 543–557. Springer, 2014.

5. P. Julián-Iranzo and C. Rubio-Manzano. Proximity-based unification theory. *Fuzzy Sets and Systems*, 262:21–43, 2015.
6. T. Kutsia and C. Pau. Proximity-based generalization. In M. Ayala Rincón and P. Balbiani, editors, *Proceedings of the 32nd International Workshop on Unification, UNIF 2018*, 2018.
7. G. D. Plotkin. A note on inductive generalization. *Machine Intel.*, 5(1):153–163, 1970.
8. J. C. Reynolds. Transformational systems and the algebraic structure of atomic formulas. *Machine Intel.*, 5(1):135–151, 1970.
9. M. I. Sessa. Approximate reasoning by similarity-based SLD resolution. *Theor. Comput. Sci.*, 275(1-2):389–426, 2002.