Extending the ρLog calculus with proximity relations

Besik Dundua, Temur Kutsia, Mircea Marin and Cleo Pau

Abstract ρLog-prox is a calculus for rule-based programming with strategies, which supports both exact and approximate computations. Rules are represented as conditional transformations of sequences of expressions, which are built from variadic function symbols and four kinds of variables: for terms, hedges, function symbols, and contexts. ρLog-prox extends ρLog by permitting in its programs fuzzy proximity relations, which are reflexive and symmetric, but not transitive. We introduce syntax and operational semantics of ρLog-prox, illustrate its work by examples, and present a terminating, sound, and complete algorithm for the ρLog-prox expression matching problem.

1 Introduction

ρLog [18] is a calculus for conditional transformation of sequences of expressions, controlled by strategies. It originated from experiments with extending the language of Mathematica [26] by a rule-based programming package [17, 19]. Meanwhile there are some tools based on or influenced by ρLog, such as its implementation in Mathematica [16], an extension of Prolog, called PρLog [7], or an extension of Maple, called symbtrans [3].

Besik Dundua
VIAM, Ivane Javakhishvili Tbilisi State University and International Black Sea University, Tbilisi, Georgia, e-mail: bdundua@egmail.com

Temur Kutsia
RISC, Johannes Kepler University Linz, Austria, e-mail: kutsia@risc.jku.at

Mircea Marin
West University of Timişoara, Romania, e-mail: mircea.marin@e-uvt.ro

Cleo Pau
RISC, Johannes Kepler University Linz, Austria, e-mail: ipau@risc.jku.at

ρLog objects are logic terms that are built from function symbols without fixed arity and four different kinds of variables: for individual terms, for finite sequences of terms (hedges), for function symbols, and for contexts (special unary higher-order functions). Rules transform finite sequences of terms, when the given conditions are satisfied. They are labeled by strategies, providing a flexible mechanism for combining and controlling their behavior. ρLog programs are sets of rules. The inference system is based on SLDNF-resolution [15]. Program meaning is characterized by logic programming semantics. Rules and strategies are formulated as clauses.

ρLog-based/inspired tools have been used in extraction of frequent patterns from data mining workflows [22], for automatic derivation of multiscale models of arrays of micro- and nanosystems [27], modeling rewriting strategies [6], etc.

The core of ρLog is a powerful pattern matching algorithm [13]. Matching with hedge and context variables is finitary: problems might have finitely many different solutions. In many situations, it can replace recursion, leading to pretty compact and intuitive code. Nondeterministic computations are modeled naturally by backtracking.

The computational mechanism of ρLog is based on the assumption that the provided information is precise and the problems can be solved exactly. However, in many cases, especially in the areas related to applications of artificial intelligence, one has to deal with vague information, which increases demand for the corresponding reasoning and computing techniques. Several approaches to this problem propose methods and tools that integrate fuzzy logic or probabilistic reasoning with declarative programming, see, e.g., [8–11, 14, 20, 21, 23, 24].

ρLog-prox, described in this paper, is an attempt to address this problem by combining approximate reasoning and strategic rule-based programming. It extends ρLog with the capabilities to process imprecise information represented by proximity relations. The latter are binary fuzzy relations, satisfying the properties of reflexivity and symmetry. We develop a matching algorithm that solves the problem of approximate equality between terms that may contain variables for terms, hedges, function symbols and contexts. A particular difficulty is related to the fact that proximity relations are not transitive. We prove that our matching algorithm is terminating, sound, and complete, and integrate it in the ρLog-prox calculus. The integration is transparent: approximate equality is expressed explicitly, no hidden fuzziness is assumed. Multiple solutions to matching problems are explored by nondeterministic computations in the inference mechanism.

The rest of the paper is organized as follows: In Section 2 we introduce the terminology, define our language, and discuss proximity relations. Section 3 is about the basics of ρLog-prox: its syntax, semantics, and an illustrative example are presented. In Section 4, we develop an algorithm for solving proximity matching problems and prove its properties. Section 5 is the conclusion.

2 Preliminaries

In this section, we introduce the basic notions needed in the rest of the paper.

Terms, hedges, contexts, substitutions

The alphabet $\mathscr{A}$ consists of the following pairwise disjoint sets of symbols:

- $\mathscr{V}_\mathsf{T}$: term variables, denoted by $x, y, z, \ldots$,
- $\mathscr{V}_\mathsf{S}$: hedge variables, denoted by $\overline{x}, \overline{y}, \overline{z}, \ldots$,
- $\mathscr{V}_\mathsf{F}$: function variables, denoted by $X, Y, Z, \ldots$,
- $\mathscr{V}_\mathsf{C}$: context variables, denoted by $\overline{X}, \overline{Y}, \overline{Y}, \ldots$,
- $\mathscr{F}$: unranked function symbols, denoted by $f, g, h, \ldots$.

Besides, $\mathscr{A}$ contains also auxiliary symbols such as parenthesis and comma, and a special constant $\circ$, called *hole*. A *variable* is an element of the set $\mathscr{V} = \mathscr{V}_\mathsf{T} \cup \mathscr{V}_\mathsf{S} \cup \mathscr{V}_\mathsf{F} \cup \mathscr{V}_\mathsf{C}$. A *functor*, denoted by F, is a common name for a function symbol or a function variable.

We define *terms, hedges, contexts*, and other syntactic categories over $\mathscr{A}$ as follows:

$t ::= x \mid f(\tilde{s}) \mid X(\tilde{s}) \mid \overline{X}(t)$	Term
$\tilde{t} ::= t_1, \ldots, t_n \quad (n \geq 0)$	Term sequence
$s ::= t \mid \overline{x}$	Hedge element
$\tilde{s} ::= s_1, \ldots, s_n \quad (n \geq 0)$	Hedge
$C ::= \circ \mid f(\tilde{s}_1, C, \tilde{s}_2) \mid X(\tilde{s}_1, C, \tilde{s}_2) \mid \overline{X}(C)$	Context

Hence, hedges are sequences of hedge elements, hedge variables are not terms, term sequences do not contain hedge variables, contexts (which are not terms either) contain a single occurrence of the hole. We do not distinguish between a singleton hedge and its sole element.

We denote the set of terms by $\mathscr{T}(\mathscr{F}, \mathscr{V})$, hedges by $\mathscr{H}(\mathscr{F}, \mathscr{V})$, and contexts by $\mathscr{C}(\mathscr{F}, \mathscr{V})$. Ground (i.e., variable-free) subsets of these sets are denoted by $\mathscr{T}(\mathscr{F})$, $\mathscr{H}(\mathscr{F})$, and $\mathscr{C}(\mathscr{F})$, respectively.

We make a couple of conventions to improve readability. We put parentheses around hedges, writing, e.g., $(f(a), \overline{x}, b)$ instead of $f(a), \overline{x}, b$. The empty hedge is written as $()$. The terms of the form $a()$ and $X()$ are abbreviated as a and X, respectively, when it is guaranteed that terms and symbols are not confused. For hedges $\tilde{s} = (s_1, \ldots, s_n)$ and $\tilde{s}' = (s'_1, \ldots, s'_m)$, the notation $(\tilde{s}, \tilde{s}')$ stands for the hedge $(s_1, \ldots, s_n, s'_1, \ldots, s'_m)$. We use $\tilde{s}$ and $\tilde{r}$ for arbitrary hedges, while $\tilde{t}$ is reserved for term sequences.

Below we will also need anonymous variables for each variable category. They are variables without name, well-known in declarative programming. We write the

single underscore _ for anonymous term and function variables, and the double underscore __ for anonymous hedge and context variables. The set of anonymous variables is denoted by $\mathscr{V}_{\mathsf{An}}$.

A syntactic expression (or, just an expression) is an element of the set $\mathscr{F} \cup \mathscr{V} \cup \mathscr{T}(\mathscr{F},\mathscr{V}) \cup \mathscr{H}(\mathscr{F},\mathscr{V}) \cup \mathscr{C}(\mathscr{F},\mathscr{V})$. We denote expressions by E.

We also introduce two notations: $\mathscr{V}(E)$ denotes the set of variables occurring in expression E, and $\mathscr{V}(E,\{p_1,...,p_n\})$, where p_i's are positions in E, is defined as $\mathscr{V}(E,\{p_1,...,p_n\}) = \cup_{i=1}^{n} \mathscr{V}(E|_{p_i})$, where $E|_{p_i}$ is the standard notation for a subexpression of E at position p_i.

Contexts can *apply* to contexts or terms. This meta-operation is denoted by $C_1[C_2]$ or $C_1[t]$ and is obtained from C_1 by replacing the hole in it by C_2 or t, respectively. Thus, $C_1[C_2]$ is a context and $C_1[t]$ is a term.

Substitution is a mapping σ from $\mathscr{V}$ to $\mathscr{T}(\mathscr{F},\mathscr{V}) \cup \mathscr{H}(\mathscr{F},\mathscr{V}) \cup \mathscr{C}(\mathscr{F},\mathscr{V}) \cup \mathscr{F} \cup \mathscr{V}_{\mathsf{F}}$, defined as

$$\begin{array}{ll} \sigma(x) \in \mathscr{T}(\mathscr{F},\mathscr{V}), & \sigma(\bar{x}) \in \mathscr{H}(\mathscr{F},\mathscr{V}), \\ \sigma(X) \in \mathscr{F} \cup \mathscr{V}_{\mathsf{F}}, & \sigma(\overline{X}) \in \mathscr{C}(\mathscr{F},\mathscr{V}), \end{array}$$

such that $\sigma(v) = v$ for all but finitely many term, hedge, and function variables v, and $\overline{X} = \overline{X}(\circ)$ for all but finitely many context variables $\overline{X}$.

Substitutions are denoted by Greek letters σ, ϑ, φ. The identity substitution is denoted by Id.

A substitution σ may *apply* to elements of the set $\mathscr{T}(\mathscr{F},\mathscr{V}) \cup \mathscr{H}(\mathscr{F},\mathscr{V}) \cup \mathscr{C}(\mathscr{F},\mathscr{V}) \cup \mathscr{F} \cup \mathscr{V}_{\mathsf{F}}$ in the following way:

$$\begin{array}{lll} x\sigma = \sigma(x), & F(\tilde{s})\sigma = (F\sigma)(\tilde{s}\sigma), & \overline{X}(t)\sigma = \sigma(\overline{X})[t\sigma], \\ \bar{x}\sigma = \sigma(\bar{x}), & (s_1,\ldots,s_n)\sigma = (s_1\sigma,\ldots,s_n\sigma), & X\sigma = \sigma(X), \quad f\sigma = f, \\ \circ\sigma = \circ, & F(\tilde{s}_1,C,\tilde{s}_2)\sigma = (F\sigma)(\tilde{s}_1\sigma,C\sigma,\tilde{s}_2\sigma), & \overline{X}(C)\sigma = \sigma(\overline{X})[C\sigma]. \end{array}$$

Proximity relations

Basic notions about proximity relations are defined following [11].

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

A fuzzy relation $\mathscr{R}$ on a set S is called a *proximity relation*, if it reflexive and symmetric:

Reflexivity: $\mathscr{R}(s,s) = 1$ for all $s \in S$;
Symmetry: $\mathscr{R}(s_1,s_2) = \mathscr{R}(s_2,s_1)$ for all $s_1,s_2 \in S$.

In this paper we consider only strict proximity relations:

Strictness: For all $s_1,s_2 \in S$, if $\mathscr{R}(s_1,s_2) = 1$ then $s_1 = s_2$.

A proximity relation is characterized by a set $\Lambda = \{\lambda_1, \ldots, \lambda_n \mid 0 < \lambda_i \leq 1\}$ of *approximation levels*. They express the degree of relationship of the related elements. We say that a value $\lambda \in \Lambda$ is a cut value. The λ-cut of $\mathscr{R}$, defined as $\mathscr{R}_\lambda = \{(s_1, s_2) \mid R(s_1, s_2) \geq \lambda\}$ is a usual two-valued tolerance (i.e., reflexive and symmetric) relation.

A *T-norm* $\wedge$ is an associative, commutative, non-decreasing binary operation on $[0,1]$ with 1 as the unit element. In the rest of the paper, we take minimum in the role of T-norm.

The *proximity class of level* $\lambda > 0$ *of* $s \in S$ *in a relation* $\mathscr{R}$ (a λ*-class* of s in $\mathscr{R}$) is a set $\mathbf{pc}(s, \mathscr{R}, \lambda) = \{s' \mid \mathscr{R}(s, s') \geq \lambda\}$.

Our proximity relations are defined on the set of function symbols $\mathscr{F}$. We require them to be defined in such a way that the proximity class for each symbol is finite. Given a proximity relation $\mathscr{R}$ defined on $\mathscr{F}$, we extend it to $\mathscr{F} \cup \mathscr{V} \cup \mathscr{T}(\mathscr{F}, \mathscr{V}) \cup \mathscr{H}(\mathscr{F}, \mathscr{V}) \cup \mathscr{C}(\mathscr{F}, \mathscr{V})$:

- For variables, $V \in \mathscr{V}$:
  - $\mathscr{R}(V, V) = 1$.
- For terms, $t, t' \in \mathscr{T}(\mathscr{F}, \mathscr{V})$:
  - If t and t' have the same number of arguments, e.g., $t = F(s_1, \ldots, s_n)$ and $t' = F'(s'_1, \ldots, s'_n)$, then $\mathscr{R}(t, t') = \mathscr{R}(F, F') \wedge \mathscr{R}(s_1, s'_1) \wedge \cdots \wedge \mathscr{R}(s_n, s'_n)$.
- For hedges, $s, s' \in \mathscr{H}(\mathscr{F}, \mathscr{V})$:
  - If $\tilde{s}$ and $\tilde{s'}$ have the same number of elements, e.g., $\tilde{s} = (s_1, \ldots, s_n)$ and $\tilde{s'} = (s'_1, \ldots, s'_n)$, then $\mathscr{R}(\tilde{s}, \tilde{s'}) = \mathscr{R}(s_1, s'_1) \wedge \cdots \wedge \mathscr{R}(s_n, s'_n)$.
- For contexts, $C, C' \in \mathscr{C}(\mathscr{F}, \mathscr{V})$:
  - $\mathscr{R}(\circ, \circ) = 1$.
  - If C and C' have the same number of arguments and their context arguments appear in the same position, e.g., $C = F(s_1, \ldots, s_{i-1}, C_1, s_{i+1}, \ldots, s_n)$ and $C' = F'(s'_1, \ldots, s'_{i-1}, C'_1, s'_{i+1} \ldots, s'_n)$, then $\mathscr{R}(C, C') = \mathscr{R}(F, F') \wedge \mathscr{R}(s_1, s'_1) \wedge \cdots \mathscr{R}(s_{i-1}, s'_{i-1}) \wedge \mathscr{R}(C_1, C'_1) \wedge \mathscr{R}(s_{i+1}, s'_{i+1}) \wedge \mathscr{R}(s_n, s'_n)$.
- In all other cases, $\mathscr{R}(E, E') = 0$ for two syntactic expressions $E, E' \in \mathscr{V} \cup \mathscr{T}(\mathscr{F}, \mathscr{V}) \cup \mathscr{H}(\mathscr{F}, \mathscr{V}) \cup \mathscr{C}(\mathscr{F}, \mathscr{V})$.

When $\mathscr{R}$ is strict on $\mathscr{F}$, its extension to $\mathscr{F} \cup \mathscr{V} \cup \mathscr{T}(\mathscr{F}, \mathscr{V}) \cup \mathscr{H}(\mathscr{F}, \mathscr{V}) \cup \mathscr{C}(\mathscr{F}, \mathscr{V})$ is also strict.

The notion of proximity class extends to elements of $\mathscr{F} \cup \mathscr{V} \cup \mathscr{T}(\mathscr{F}, \mathscr{V}) \cup \mathscr{H}(\mathscr{F}, \mathscr{V}) \cup \mathscr{C}(\mathscr{F}, \mathscr{V})$. It is easy to see that each proximity class in this set is also finite.

3 ρLog-prox: ρLog with proximity relations

Syntactic matching and proximity matching problems

A *syntactic matching atom* is a formula of the form $E_1 \ll E_2$. It is solved if the expressions E_1 and E_2 are identical, i.e., if $E_1 = E_2$. A substitution σ is a *solution* (or a *matcher*) of a matching atom $E_1 \ll E_2$ iff $E_1\sigma = E_2$.

Example 1. The syntactic matching atom

$$(X(a),\bar{x},\overline{Y}(X(\bar{x},y)),\bar{z}) \ll (f(a),g(b,f(b),f(a,f(b))),b,c)$$

has two solutions:

$$\sigma_1 = \{X \mapsto f, \bar{x} \mapsto (), \overline{Y} \mapsto g(b,\circ,f(a,f(b))), y \mapsto b, \bar{z} \mapsto (b,c)\}$$
$$\sigma_2 = \{X \mapsto f, \bar{x} \mapsto (), \overline{Y} \mapsto g(b,f(b),f(a,\circ)), y \mapsto b, \bar{z} \mapsto (b,c)\}$$

A *syntactic matching problem* is a finite set of syntactic matching atoms. Its solution is a substitution which solves each of the atoms in the problem.

Given a proximity relation $\mathscr{R}$ and a cut value λ, an $(\mathscr{R},\lambda)$-*proximity atom* is a formula $E_1 \ll_{\mathscr{R},\lambda} E_2$ for the expressions E_1 and E_2. Its *solution* is a substitution σ such that $\mathscr{R}(E_1\sigma,E_2) \geq \lambda$. A *solution with the proximity degree* α is a substitution σ such that $\mathscr{R}(E_1\sigma,E_2) = \alpha \geq \lambda$.

Example 2. Let the proximity relation $\mathscr{R}$ be given by the following:

$$\mathscr{R}(g_1,h_1) = \mathscr{R}(g_2,h_1) = 0.4$$
$$\mathscr{R}(g_1,h_2) = \mathscr{R}(g_2,h_2) = 0.5$$
$$\mathscr{R}(g_2,h_3) = \mathscr{R}(g_3,h_3) = 0.6$$
$$\mathscr{R}(a,b) = 0.7$$

Let the proximity atom be

$$P = f(\bar{x},x,\overline{Y}(x),\bar{z}) \ll_{\mathscr{R},\lambda} f(g_1(a),g_2(b),f(g_3(a))).$$

Consider the approximation levels $\Lambda = \{0.4,0.5,0.6,0.7\}$. We get the following solutions to P: (In all cases, the proximity degrees of solutions coincide with λ.)

$\lambda = 0.4$:

$$\sigma_1 = \{\bar{x} \mapsto (), x \mapsto h_1(a), \overline{Y} \mapsto \circ, \bar{z} \mapsto f(g_3(a))\}$$
$$\sigma_2 = \{\bar{x} \mapsto (), x \mapsto h_2(a), \overline{Y} \mapsto \circ, \bar{z} \mapsto f(g_3(a))\}$$
$$\sigma_3 = \{\bar{x} \mapsto (), x \mapsto h_1(b), \overline{Y} \mapsto \circ, \bar{z} \mapsto f(g_3(a))\}$$
$$\sigma_4 = \{\bar{x} \mapsto (), x \mapsto h_2(b), \overline{Y} \mapsto \circ, \bar{z} \mapsto f(g_3(a))\}$$
$$\sigma_5 = \{\bar{x} \mapsto (), x \mapsto h_1(a), \overline{Y} \mapsto \circ, \bar{z} \mapsto f(g_3(b))\}$$

$$\sigma_6 = \{\overline{x} \mapsto (), x \mapsto h_2(a), \overline{Y} \mapsto \circ, \overline{z} \mapsto f(g_3(b))\}$$
$$\sigma_7 = \{\overline{x} \mapsto (), x \mapsto h_1(b), \overline{Y} \mapsto \circ, \overline{z} \mapsto f(g_3(b))\}$$
$$\sigma_8 = \{\overline{x} \mapsto (), x \mapsto h_2(b), \overline{Y} \mapsto \circ, \overline{z} \mapsto f(g_3(b))\}$$
$$\sigma_9 = \{\overline{x} \mapsto (), x \mapsto h_1(a), \overline{Y} \mapsto \circ, \overline{z} \mapsto f(h_3(a))\}$$
$$\sigma_{10} = \{\overline{x} \mapsto (), x \mapsto h_2(a), \overline{Y} \mapsto \circ, \overline{z} \mapsto f(h_3(a))\}$$
$$\sigma_{11} = \{\overline{x} \mapsto (), x \mapsto h_1(b), \overline{Y} \mapsto \circ, \overline{z} \mapsto f(h_3(a))\}$$
$$\sigma_{12} = \{\overline{x} \mapsto (), x \mapsto h_2(b), \overline{Y} \mapsto \circ, \overline{z} \mapsto f(h_3(a))\}$$
$$\sigma_{13} = \{\overline{x} \mapsto (), x \mapsto h_1(a), \overline{Y} \mapsto \circ, \overline{z} \mapsto f(h_3(b))\}$$
$$\sigma_{14} = \{\overline{x} \mapsto (), x \mapsto h_2(a), \overline{Y} \mapsto \circ, \overline{z} \mapsto f(h_3(b))\}$$
$$\sigma_{15} = \{\overline{x} \mapsto (), x \mapsto h_1(b), \overline{Y} \mapsto \circ, \overline{z} \mapsto f(h_3(b))\}$$
$$\sigma_{16} = \{\overline{x} \mapsto (), x \mapsto h_2(b), \overline{Y} \mapsto \circ, \overline{z} \mapsto f(h_3(b))\}$$
$$\sigma_{17} = \{\overline{x} \mapsto g_1(a), x \mapsto h_3(a), \overline{Y} \mapsto f(\circ), \overline{z} \mapsto ()\}$$
$$\sigma_{18} = \{\overline{x} \mapsto g_1(a), x \mapsto h_3(b), \overline{Y} \mapsto f(\circ), \overline{z} \mapsto ()\}$$
$$\sigma_{19} = \{\overline{x} \mapsto g_1(b), x \mapsto h_3(a), \overline{Y} \mapsto f(\circ), \overline{z} \mapsto ()\}$$
$$\sigma_{20} = \{\overline{x} \mapsto g_1(b), x \mapsto h_3(b), \overline{Y} \mapsto f(\circ), \overline{z} \mapsto ()\}$$
$$\sigma_{21} = \{\overline{x} \mapsto h_1(a), x \mapsto h_3(a), \overline{Y} \mapsto f(\circ), \overline{z} \mapsto ()\}$$
$$\sigma_{22} = \{\overline{x} \mapsto h_1(a), x \mapsto h_3(b), \overline{Y} \mapsto f(\circ), \overline{z} \mapsto ()\}$$
$$\sigma_{23} = \{\overline{x} \mapsto h_1(b), x \mapsto h_3(a), \overline{Y} \mapsto f(\circ), \overline{z} \mapsto ()\}$$
$$\sigma_{24} = \{\overline{x} \mapsto h_1(b), x \mapsto h_3(b), \overline{Y} \mapsto f(\circ), \overline{z} \mapsto ()\}$$
$$\sigma_{25} = \{\overline{x} \mapsto h_2(a), x \mapsto h_3(a), \overline{Y} \mapsto f(\circ), \overline{z} \mapsto ()\}$$
$$\sigma_{26} = \{\overline{x} \mapsto h_2(a), x \mapsto h_3(b), \overline{Y} \mapsto f(\circ), \overline{z} \mapsto ()\}$$
$$\sigma_{27} = \{\overline{x} \mapsto h_2(b), x \mapsto h_3(a), \overline{Y} \mapsto f(\circ), \overline{z} \mapsto ()\}$$
$$\sigma_{28} = \{\overline{x} \mapsto h_2(b), x \mapsto h_3(b), \overline{Y} \mapsto f(\circ), \overline{z} \mapsto ()\}$$

$\lambda = 0.5$:

$$\sigma_1 = \{\overline{x} \mapsto (), x \mapsto h_2(a), \overline{Y} \mapsto \circ, \overline{z} \mapsto f(g_3(a))\}$$
$$\sigma_2 = \{\overline{x} \mapsto (), x \mapsto h_2(b), \overline{Y} \mapsto \circ, \overline{z} \mapsto f(g_3(a))\}$$
$$\sigma_3 = \{\overline{x} \mapsto (), x \mapsto h_2(a), \overline{Y} \mapsto \circ, \overline{z} \mapsto f(g_3(b))\}$$
$$\sigma_4 = \{\overline{x} \mapsto (), x \mapsto h_2(b), \overline{Y} \mapsto \circ, \overline{z} \mapsto f(g_3(b))\}$$
$$\sigma_5 = \{\overline{x} \mapsto (), x \mapsto h_2(a), \overline{Y} \mapsto \circ, \overline{z} \mapsto f(h_3(a))\}$$
$$\sigma_6 = \{\overline{x} \mapsto (), x \mapsto h_2(b), \overline{Y} \mapsto \circ, \overline{z} \mapsto f(h_3(a))\}$$
$$\sigma_7 = \{\overline{x} \mapsto (), x \mapsto h_2(a), \overline{Y} \mapsto \circ, \overline{z} \mapsto f(h_3(b))\}$$
$$\sigma_8 = \{\overline{x} \mapsto (), x \mapsto h_2(b), \overline{Y} \mapsto \circ, \overline{z} \mapsto f(h_3(b))\}$$
$$\sigma_9 = \{\overline{x} \mapsto g_1(a), x \mapsto h_3(a), \overline{Y} \mapsto f(\circ), \overline{z} \mapsto ()\}$$

$$\sigma_{10} = \{\overline{x} \mapsto g_1(a), x \mapsto h_3(b), \overline{Y} \mapsto f(\circ), \overline{z} \mapsto ()\}$$
$$\sigma_{11} = \{\overline{x} \mapsto g_1(b), x \mapsto h_3(a), \overline{Y} \mapsto f(\circ), \overline{z} \mapsto ()\}$$
$$\sigma_{12} = \{\overline{x} \mapsto g_1(b), x \mapsto h_3(b), \overline{Y} \mapsto f(\circ), \overline{z} \mapsto ()\}$$
$$\sigma_{13} = \{\overline{x} \mapsto h_2(a), x \mapsto h_3(a), \overline{Y} \mapsto f(\circ), \overline{z} \mapsto ()\}$$
$$\sigma_{14} = \{\overline{x} \mapsto h_2(a), x \mapsto h_3(b), \overline{Y} \mapsto f(\circ), \overline{z} \mapsto ()\}$$
$$\sigma_{15} = \{\overline{x} \mapsto h_2(b), x \mapsto h_3(a), \overline{Y} \mapsto f(\circ), \overline{z} \mapsto ()\}$$
$$\sigma_{16} = \{\overline{x} \mapsto h_2(b), x \mapsto h_3(b), \overline{Y} \mapsto f(\circ), \overline{z} \mapsto ()\}$$

$\lambda = 0.6$:

$$\sigma_1 = \{\overline{x} \mapsto g_1(a), x \mapsto h_3(a), \overline{Y} \mapsto f(\circ), \overline{z} \mapsto ()\}$$
$$\sigma_2 = \{\overline{x} \mapsto g_1(a), x \mapsto h_3(b), \overline{Y} \mapsto f(\circ), \overline{z} \mapsto ()\}$$
$$\sigma_3 = \{\overline{x} \mapsto g_1(b), x \mapsto h_3(a), \overline{Y} \mapsto f(\circ), \overline{z} \mapsto ()\}$$
$$\sigma_4 = \{\overline{x} \mapsto g_1(b), x \mapsto h_3(b), \overline{Y} \mapsto f(\circ), \overline{z} \mapsto ()\}$$

$\lambda = 0.7$: No solutions.

A *proximity matching problem* is a set of proximity atoms. A substitution σ is a *solution of a proximity matching problem* $\{A_1, \ldots, A_n\}$ (with proximity degree α), if σ is a solution of each atom A_i (with proximity degree α_i and $\alpha = \alpha_1 \wedge \cdots \wedge \alpha_n$).

Note that because of strictness, syntactic matching can be seen as special proximity matching for an arbitrary $\mathscr{R}$ with the lambda-cut equal to 1. Therefore, for simplicity, below we will talk only about proximity matching problems and refer to them briefly as proximity problems.

ρLog-prox programs and proximity relations

ρLog-prox programs consist of conditional rules for hedge transformations. A transformation is an atomic formula (an *atom*) of the form $\Longrightarrow(t, \langle \tilde{s}_1 \rangle, \langle \tilde{s}_2 \rangle)$, where $\Longrightarrow$ is a ternary predicate symbol and $\langle \cdot \rangle$ is a function symbol (which appears neither in t nor in $\tilde{s}_1$ and $\tilde{s}_2$). Such an atom is usually written as $t :: \tilde{s}_1 \Longrightarrow \tilde{s}_2$. Intuitively, it means that the hedge $\tilde{s}_1$ is transformed into the hedge $\tilde{s}_2$ by the *strategy* t. Atoms are denoted by A and B.

A ρLog-prox *query* is a conjunction of atoms, written as $B_1, \ldots, B_n$. A ρLog-prox *clause* has a form $A \leftarrow Q$, where $\leftarrow$ is the inverse implication sign, A is an atom, called the *head* of the clause, and Q is a query, called the *body* of the clause. ρLog-prox programs are finite sets of ρLog-prox clauses.

We assume that for each program there is an associated proximity relation defined on the set of function symbols. For such a relation $\mathscr{R}$, the set of (f,g) pairs with $\mathscr{R}(f,g) > 0$ is finite.

A special predefined strategy is **prox**, which takes a single argument, a number from the real interval $(0,1]$. The atom $\mathbf{prox}(\lambda) :: \tilde{s}_1 \Longrightarrow \tilde{s}_2$ is true iff the proximity problem $\tilde{s}_2 \ll_{\mathscr{R},\lambda} \tilde{s}_1$ is solvable for the given $\mathscr{R}$. When $\lambda = 1$, **prox** coincides

with the identity strategy **id** of the original ρLog [18] (the strictness assumption is important here).

For the original version of ρLog, semantics of programs can be defined in the same way as it is done for logic programming [1, 15]. Having defined proximity strategies as ρLog-prox atoms, we can do the same for our version of ρLog-prox programs.

Note that the same strategy can be defined by several clauses, which are treated as alternatives.

Now we introduce the inference system of ρLog-prox calculus with proximity relations. It has two rules: resolution and proximity factoring. A program and a proximity relation $\mathscr{R}$ are given.

Resolution takes a query with an atom selected in it and a renamed copy of a program clause and performs the inference step, producing a new query as follows:

$$\frac{str_{\mathrm{q}} :: lhs_{\mathrm{q}} \Longrightarrow rhs_{\mathrm{q}}, Q \qquad str_{\mathrm{p}} :: lhs_{\mathrm{p}} \Longrightarrow rhs_{\mathrm{p}} \leftarrow Body}{(Body,\ \mathbf{prox}(1) :: rhs_{\mathrm{p}} \Longrightarrow rhs_{\mathrm{q}},\ Q)\sigma},$$

where σ is a solution of the proximity problem $\{str_{\mathrm{p}} \ll_{\mathscr{R},1} str_{\mathrm{q}}, lhs_{\mathrm{p}} \ll_{\mathscr{R},1} lhs_{\mathrm{q}}\}$. The strategy str_{q} does not have the form $\mathbf{prox}(\lambda)$.

Proximity factoring takes a query, in which an atom with the proximity strategy is selected, and produces a new query:

$$\frac{\mathbf{prox}(\lambda) :: lhs_{\mathrm{q}} \Longrightarrow rhs_{\mathrm{q}},\ Q}{Q\sigma},$$

where σ is a solution of the proximity problem $\{rhs_{\mathrm{q}} \ll_{\mathscr{R},\lambda} lhs_{\mathrm{q}}\}$.

A *derivation* of a query Q from a program P (with respect to a proximity relation $\mathscr{R}$) is a sequence of queries $Q_0, Q_1, \ldots$, where $Q_0 = Q$ and Q_i is obtained from Q_{i-1} by resolution or proximity factoring. A derivation is *successful* if it ends with the empty query. In this case, the union of substitutions computed along the derivation, restricted to variables from Q, is called the *answer computed for Q via P*. A derivation is *failed*, if none of the inference rules can apply to the last query, which is nonempty. Like for the original ρLog, the inference system is sound: the computed answers are also correct with respect to the declarative semantics. It is not complete in general due to the leftmost query selection strategy. Completeness is ensured for queries with terminating derivations.

We can allow negations of atoms in queries and clause bodies, as in normal logic programs [1, 2, 15]. *Literal* is a common name for an atom and its negation. We use the letter L to denote them. To deal with negative literals, the inference system can be extended by the well-known negation-as-failure rule.

In order to guarantee that inference in ρLog-prox is performed by matching and not unification (because the latter problems may have infinitely many solutions [5, 12]), we work with well-moded programs and queries.

Definition 1 (Well-moded clauses, programs, queries). Let C be a (normal) clause

$$str_0 :: \tilde{r}_0 \Longrightarrow \tilde{s}_{n+1} \leftarrow L_1, \ldots, L_n,$$

where for each $1 \leq i \leq n$, the literal L_i is either an atom $str_i :: \tilde{s}_i \Longrightarrow \tilde{r}_i$ or a negation of an atom $str_i :: \tilde{s}_i \not\Longrightarrow \tilde{r}_i$. C is *well-moded* if for all $1 \leq i \leq n+1$, we have

- $\mathscr{V}(str_i) \cup \mathscr{V}(\tilde{s}_i) \subseteq \mathscr{V}(str_0) \cup \bigcup_{j=0}^{i-1} \mathscr{V}(\tilde{r}) \setminus \mathscr{V}_{\mathsf{An}}$, and
- if L_i is a negative literal, then $\mathscr{V}(\tilde{r}_i) \subseteq \mathscr{V}(str_0) \cup \bigcup_{j=0}^{i-1} \mathscr{V}(\tilde{r}) \cup \mathscr{V}_{\mathsf{An}}$.

A (normal) ρLog-prox program is *well-moded* if all clauses in it all well-moded.

A (normal) query $L_1, \ldots, L_n$ is *well-moded* if the clause $A \leftarrow L_1, \ldots, L_n$ is well-moded, where A is a dummy ground atom.

Example 3. In this rather extended example we illustrate ρLog-prox clauses, strategies, and evaluation mechanism. We borrow the material from [7] and adapt it to ρLog-prox.

An instance of a transformation is finding duplicated elements in a hedge and removing one of them. Let us call this process the merging of duplicates. The following strategy implements the idea:

$$\text{merge\_duplicates} :: (\bar{x}, x, \bar{y}, x, \bar{z}) \Longrightarrow (\bar{x}, x, \bar{y}, \bar{z}).$$

merge_duplicates is the strategy name. The clause is obviously well-moded. It says that if the hedge in *lhs* contains duplicates (expressed by two copies of the variable x) somewhere, then from these two copies only the first one should be kept in *rhs*. That "somewhere" is expressed by three hedge variables, where $\bar{x}$ stands for the subhedge before the first occurrence of x, $\bar{y}$ takes the subhedge between two occurrences of x, and $\bar{z}$ matches the remaining part. These subhedges remain unchanged in the *rhs*.

One does not need to code the actual search process of duplicates explicitly. The matching algorithm is supposed to do the job instead, looking for an appropriate instantiation of the variables. There can be several such instantiations.

Now one can ask, e.g., to merge duplicates in a hedge (a, b, c, b, a):

$$\text{merge\_duplicates} :: (a, b, c, b, a) \Longrightarrow \bar{x}.$$

To this query, ρLog-prox returns two answer substitutions: $\{\bar{x} \mapsto (a, b, c, b)\}$ and $\{\bar{x} \mapsto (a, b, c, a)\}$. Both are obtained from (a, b, c, b, a) by merging one pair of duplicates.

Now we generalize merge_duplicates allowing merging of approximate duplicates (we use l as a term variable):

$$\text{merge\_duplicates}(l) :: (\bar{x}, x, \bar{y}, y, \bar{z}) \Longrightarrow (\bar{x}, x, \bar{y}, \bar{z}) \leftarrow \mathbf{prox}(l) :: x \Longrightarrow y.$$

This clause (which is well-moded) removes y from the given hedge, if the hedge contains an x such that x and y are close to each other with respect to the given

proximity relation with the proximity degree l. The merge_duplicates strategy above is just a special case of merge_duplicates(l) with $l = 1$.

Assume now that in the proximity relation $\mathscr{R}$, we have $\mathscr{R}(a,e) = 0.6$ and $\mathscr{R}(b,d) = 0.7$. Then the query

$$\text{merge\_duplicates}(0.8) :: (a, b, c, d, e) \Longrightarrow \bar{x}$$

fails, because (a, b, c, d, e) does not contain elements which are close to each other with the proximity degree at least 0.8. If we take $l = 0.7$, i.e., the query

$$\text{merge\_duplicates}(0.7) :: (a, b, c, d, e) \Longrightarrow \bar{x},$$

we get a single answer: $\{\bar{x} \mapsto (a, b, c, e)\}$. Decreasing l further and taking the query

$$\text{merge\_duplicates}(0.6) :: (a, b, c, d, e) \Longrightarrow \bar{x},$$

we get two answers (via backtracking): $\{\bar{x} \mapsto (a, b, c, d)\}$ and $\{\bar{x} \mapsto (a, b, c, e)\}$.

A hedge without duplicates is a normal form with respect to this single-step merge_duplicates(l) transformation. ρLog-prox has a predefined strategy for computing normal forms, denoted by **nf**, and we can use it to define a new strategy merge_all_duplicates(l) in the following clause:

$$\text{merge\_all\_duplicates}(l) :: \bar{x} \Longrightarrow \bar{y} \;\leftarrow\; \mathbf{nf}(\text{merge\_duplicates}(l)) :: \bar{x} \Longrightarrow \bar{y}.$$

The effect of **nf** is that it applies merge_duplicates to $\bar{x}$, repeating this process iteratively as long as it is possible, i.e., as long as duplicates can be merged in the obtained hedges. When merge_duplicates is no more applicable, it means that the normal form of the transformation is reached. It is returned in $\bar{y}$.

Now, for the query

$$\text{merge\_all\_duplicates}(0.6) :: (a, b, c, d, e) \Longrightarrow \bar{x}.$$

we get a single answer $\bar{x} \mapsto (a, b, c)$. However, procedurally, this answer can be computed multiple times (via backtracking). To avoid such multiple computations, we can use another predefined strategy **first_one**:

$$\text{merge\_all\_duplicates}(l) :: \bar{x} \Longrightarrow \bar{y} \;\leftarrow$$
$$\quad \mathbf{first\_one}(\mathbf{nf}(\text{merge\_duplicates}(l))) :: \bar{x} \Longrightarrow \bar{y}.$$

first_one applies to a sequence of strategies, finds the first one among them, which successfully transforms the input hedge, and gives back just *one result* of the transformation. Here it has a single argument strategy **nf**(merge_duplicates(l)) and returns (by instantiating $\bar{y}$) only one result of its application to $\bar{x}$.

ρLog-prox is good not only in selecting arbitrarily many subexpressions in "horizontal direction" (by hedge variables), but also in working in "vertical direction", selecting subterms at arbitrary depth. Context variables provide this flexibility, by matching the context above the subterm to be selected. With the help of context and

function variables, from the merge_duplicates(l) strategy it is pretty easy to define a transformation that merges neighboring branches in a tree, which are approximately the same:

$$\text{merge\_duplicate\_branches}(l) :: \overline{X}(Y(\bar{x})) \Longrightarrow \overline{X}(Y(\bar{y})) \leftarrow$$
$$\text{merge\_duplicates}(l) :: \bar{x} \Longrightarrow \bar{y}.$$

Now, we can ask to merge neighboring branches in a given tree, which are 0.6-approximate of each other (for the same $\mathscr{R}$ as above):

$$\text{merge\_duplicate\_branches}(0.6) ::$$
$$f(g(a,b,e,h(c,c)),h(c),g(a,e,b,h(c))) \Longrightarrow x.$$

ρLog-prox computes three answers:

$$\{x \mapsto f(g(a,b,h(c,c)),h(c),g(a,e,b,h(c)))\},$$
$$\{x \mapsto f(g(a,b,e,h(c)),h(c),g(a,e,b,h(c)))\},$$
$$\{x \mapsto f(g(a,b,e,h(c,c)),h(c),g(a,b,h(c)))\}.$$

To obtain the first one, ρLog-prox matched the context variable $\overline{X}$ to the context $f(\circ,h(c),g(a,a,b,h(c)))$, the function variable Y to the function symbol g, and the hedge variable $\bar{x}$ to the hedge $(a,b,e,h(c,c))$. merge_duplicates(0.6) transformed $(a,b,e,h(c,c))$ to $(a,b,h(c,c))$. The other results have been obtained by taking different contexts and respective subbranches.

The right hand side of transformations in the queries need not be variables. One can have an arbitrary hedge there. For instance, we may be interested in trees that contain $h(c,c)$:

$$\text{merge\_duplicate\_branches}(0.6) ::$$
$$f(g(a,b,e,h(c,c)),h(c),g(a,e,b,h(c))) \Longrightarrow \overline{X}(h(c,c)).$$

We get here two answers, which show instantiations of $\overline{X}$ by the relevant contexts:

$$\{\overline{X} \mapsto f(g(a,b,\circ),h(c),g(a,e,b,h(c)))\},$$
$$\{\overline{X} \mapsto f(g(a,b,e,\circ),h(c),g(a,b,h(c)))\}.$$

Similar to merging all duplicates in a hedge above, we can also define a strategy that merges all approximately duplicate branches in a tree repeatedly. Naturally, the built-in strategy for normal forms plays a role also here:

$$\text{merge\_all\_duplicate\_branches}(l) :: x \Longrightarrow y \leftarrow$$
$$\textbf{first\_one}(\textbf{nf}(\text{merge\_duplicate\_branches}(l))) :: x \Longrightarrow y.$$

For the query

merge_all_duplicate_branches(0.6) ::
$f(g(a,b,e,h(c,c)),h(c),g(a,e,b,h(c))) \Longrightarrow \bar{x}.$

we get a single answer $\{\bar{x} \mapsto f(g(a,b,h(c)),h(c))\}$.

4 Solving proximity problems

As one could see in the previous section, the inference rules of ρLog-prox heavily rely on solving proximity problems. Well-modedness guarantees that only proximity problems with ground right hand side arise during derivations of queries from ρLog-prox programs. Resolving negative literals reduces to the problem of testing whether two ground expressions are in the given proximity relation with respect to the given cut value.

Here we describe an algorithm, which computes not only solutions to proximity problems, but also the degree of proximity for the solutions. They can be used to report the proximity degree of a query instance that is proved from the program.

We say that a set of equations $\{V_1 \approx E_1, \ldots, V_n \approx E_n\}$ is in

- *matching pre-solved form*, if the E's are ground,
- *matching solved form*, if it is in matching pre-solved form and each variable V_i appears in the set only once.

If S is a solved form, we define an associated substitution $\sigma_S := \{V_i \mapsto E_i \mid V_i \approx E_i \in S\}$.

The proximity matching algorithm $\mathfrak{P}$ is formulated in a rule-based way. Rules work on configurations, which are either a special symbol $\bot$ or triples of the form $M;S;\alpha$, where M is the proximity matching problem to be solved, S is a set of equations in matching pre-solved form (the candidate set for a solution computed so far), and α is the proximity degree of a solution computed so far. A rule that produces $\bot$ is called a failure rule. We have six success and four failure rules:

RFS: **Removing function symbols**
$\{f(\tilde{s}) \ll_{\mathscr{R},\lambda} g(\tilde{t})\} \uplus M;\ S;\ \alpha \rightsquigarrow M \cup \{\tilde{s} \ll_{\mathscr{R},\lambda} \tilde{t}\};\ S;\ \alpha \wedge \beta,$
where $\mathscr{R}(f,g) = \beta \geq \lambda$.

Dec: **Decomposition**
$\{(t,\tilde{s}) \ll_{\mathscr{R},\lambda} (t',\tilde{t})\} \uplus M;\ S;\ \alpha \rightsquigarrow M \cup \{t \ll_{\mathscr{R},\lambda} t', \tilde{s} \ll_{\mathscr{R},\lambda} \tilde{t}\};\ S;\ \alpha,$
where $\tilde{s} \neq ()$ and $\tilde{t} \neq ()$.

FVE: **Function variable elimination**
$\{X(\tilde{s}) \ll_{\mathscr{R},\lambda} g(\tilde{t})\} \uplus M;\ S;\ \alpha \rightsquigarrow M \cup \{\tilde{s} \ll_{\mathscr{R},\lambda} \tilde{t}\};\ S \cup \{X \approx g'\};\ \alpha \wedge \beta,$
where $\mathscr{R}(g',g) = \beta \geq \lambda$.

CVE: **Context variable elimination**
$\{\overline{X}(t_1) \ll_{\mathscr{R},\lambda} C(t_2)\} \uplus M;\ S;\ \alpha \rightsquigarrow M \cup \{t_1 \ll_{\mathscr{R},\lambda} t_2\};\ S \cup \{\overline{X} \approx C'\};\ \alpha \wedge \beta,$
where $\mathscr{R}(C', C) = \beta \geq \lambda$.

TVE: **Term variable elimination**
$\{x \ll_{\mathscr{R},\lambda} t\} \uplus M;\ S;\ \alpha \rightsquigarrow M;\ S \cup \{x \approx t'\};\ \alpha \wedge \beta, \qquad$ where $\mathscr{R}(t', t) = \beta \geq \lambda$.

HVE: **Hedge variable elimination**
$\{(\overline{x}, \tilde{s}) \ll_{\mathscr{R},\lambda} (\tilde{t}_1, \tilde{t}_2)\} \uplus M;\ S;\ \alpha \rightsquigarrow M \cup \{\tilde{s} \ll_{\mathscr{R},\lambda} \tilde{t}_2\};\ S \cup \{\overline{x} \approx \tilde{t}'_1\};\ \alpha \wedge \beta,$
where $\mathscr{R}(\tilde{t}'_1, \tilde{t}_1) = \beta \geq \lambda$.

Cla1: **Clash 1**
$\{f(\tilde{s}) \ll_{\mathscr{R},\lambda} g(\tilde{t})\} \uplus M;\ S;\ \alpha \rightsquigarrow \bot, \qquad$ if $\mathscr{R}(f, g) < \lambda$.

Cla2: **Clash 2**
$\{(t, \tilde{s}) \ll_{\mathscr{R},\lambda} ()\} \uplus M;\ S;\ \alpha \rightsquigarrow \bot$.

Cla3: **Clash 3**
$\{() \ll_{\mathscr{R},\lambda} (t, \tilde{t})\} \uplus M;\ S;\ \alpha \rightsquigarrow \bot$.

Inc: **Inconsistency**
$M;\ S;\ \alpha \rightsquigarrow \bot,$
if S contains two equations with the same variable in the left hand side.

To solve a proximity matching problem M, we create the initial configuration $M; \emptyset; 1$ and start applying the rules exhaustively. If the same configuration can be transformed by multiple rules, they are applied concurrently except one of the rules is Inc: in this case only Inc applies. Each elimination rule instantiates a variable not exactly with the corresponding expression in the right hand side, but with its approximate expression. Since proximity classes of objects are finite, these choices cause only finite branching. The other source of branching is the choice of a hedge and a context from the right hand side in CVE and HVE rules. Also here, there are finitely many ways to branch. The described process defines the algorithm $\mathfrak{P}$.

Theorem 1 (Termination). *The proximity matching algorithm $\mathfrak{P}$ terminates. Each final configuration has the form either $\bot$ or $\emptyset; S; \alpha$, where S is in matching solved form.*

Proof. Let $size(E)$ be the number of symbols in E. By $Msize(M)$ we denote the multiset $\{size(E_2) \mid E_1 \ll_{\mathscr{R},\lambda} E_2 \in M\}$. To each configuration $M; S; \alpha$ we associate the complexity measure, the pair $\langle Msize(M), varocc(M)\rangle$, where $varocc(M)$ is the number of variable occurrences in M. The measures are compared lexicographically, where the used orderings for the components are multiset ordering [4] and the standard ordering on natural numbers. The RFC and Dec rules decrease the first

component of the measure. (Note that for Dec it is ensured by the requirement that $\tilde{s}$ and $\tilde{t}$ are not empty hedges.) The elimination rules do not increase the first component and decrease the second one. The failure rules stop immediately, since $\bot$ is not transformed further. Hence, the algorithm terminates.

Since for each possible shape of a proximity problem there is the corresponding rule, the process stops either with $\bot$ or with a configuration of the form $\emptyset;S;\alpha$. In the latter case, S should be in solved form, otherwise Inc would transform it into $\bot$. □

From each final configuration $\emptyset;S;\alpha$, we can extract the corresponding substitution σ_S. These substitutions are called *computed answers*.

We say that σ is a solution of a (pre-solved) set of equations $\{V_1 \approx E_1, \ldots, V_n \approx E_n\}$ iff $V_i\sigma = E_i$ for each $1 \le i \le n$. A solution of a pair $M;S$ of a proximity matching problem M and a set of equations in pre-solved form S is a substitution σ that solves both M and S. The configuration $\bot$ has not solutions.

Theorem 2 (Soundness). *Let M be a proximity problem and σ be its computed answer with the proximity degree α. Then σ is a solution of M with the proximity degree α.*

Proof. Let $M_1;S_1;\alpha_1 \rightsquigarrow_{\mathsf{R}} M_2;S_2;\alpha_2$ be the step made by R, where R is one of the rules above. We show that if σ is a solution of M_2 (with the degree α_2) and S_2, then σ is a solution of M_1 (with the same degree α_2) and S_1.

R is RFS. Then $\alpha_2 = \alpha_1 \wedge \beta$, where $\mathscr{R}(f,g) = \beta \ge \lambda$. Obviously, if $\mathscr{R}(\tilde{s}\sigma,\tilde{t}) \ge \alpha_1 \wedge \beta$, then $\mathscr{R}(f(\tilde{s})\sigma, g(\tilde{t})) \ge \alpha_1 \wedge \beta$. Hence, in this case σ is a solution of M_1 with the degree α_2 and S_1 (which is the same as S_2).

R is FVE. Then $\alpha_2 = \alpha_1 \wedge \beta$ where $\mathscr{R}(g',g) = \beta$. Besides, $g' = X\sigma$. Therefore, if $\mathscr{R}(\tilde{s}\sigma,\tilde{t}) \ge \alpha_1 \wedge \beta$, then $\mathscr{R}(X(\tilde{s})\sigma, g(\tilde{t})) \ge \alpha_1 \wedge \beta$, and if σ solves S_2, then it solves also S_1. Hence, also in this case σ is a solution of M_1 with the degree α_2 and S_1.

For the other success rules the proof is similar or easier.

To prove the soundness theorem, we just need to proceed by induction on the length of a successful derivation, using the single-step soundness result we just established. □

Lemma 1. *If $M;S;\alpha \rightsquigarrow \bot$, then $M;S$ has no solution.*

Proof. Assume M is a $(\mathscr{R},\lambda)$-matching problem and analyze the rules that lead to $\bot$. For the Cla1 rule, M is unsolvable, because $\mathscr{R}((f(\tilde{s}))\sigma, g(\tilde{t})) = \mathscr{R}(f(\tilde{s}\sigma), g(\tilde{t})) = \mathscr{R}(f,g) \wedge \mathscr{R}(\tilde{s}\sigma,\tilde{t}) \le \mathscr{R}(f,g) < \lambda$. In Cla2 and Cla3 rules, unsolvability of M follows from the fact that a nonempty hedge can not be approximated by the empty hedge. In the Inc rule, if we have two equations with the same variable in the left hand side, it means that their right hand sides are different. Since equations in S are solved syntactically, it implies that S has no solution. □

Theorem 3 (Completeness). *Let M be a proximity problem and σ be its solution with the proximity degree α. Then there exists a derivation in $\mathfrak{P}$ ending with a configuration $M;\emptyset;1 \rightsquigarrow^* \emptyset;S;\alpha$, such that $\sigma = \sigma_S$.*

Proof. We construct the desired derivation under the guidance of σ. At each variable elimination step, we choose the proximal object of the variable exactly as σ does. This will guarantee that proximity degrees at each such step will be also in accordance to σ. Making RFS and Dec steps will not make the proximity degree differ from α, because σ is a solution. No clashing and inconsistency step will be performed, because by Lemma 1 it would contradict the solvability of M. Hence, if $\beta_1,\dots,\beta_n$ are all β's in the derivation, then $\beta_1 \wedge \cdots \wedge \beta_n = \alpha$. Since we start from the proximity degree 1, the computed proximity degree will be $1 \wedge \beta_1 \wedge \cdots \wedge \beta_n = \alpha$. By construction, $\sigma_S = \sigma$. □

Example 4. We use the proximity relation and problem from Example 2. The relation $\mathscr{R}$ is

$$\mathscr{R}(g_1,h_1) = \mathscr{R}(g_2,h_1) = 0.4, \qquad \mathscr{R}(g_1,h_2) = \mathscr{R}(g_2,h_2) = 0.5,$$
$$\mathscr{R}(g_2,h_3) = \mathscr{R}(g_3,h_3) = 0.6, \qquad \mathscr{R}(a,b) = 0.7.$$

The proximity problem is

$$f(\bar{x},x,\overline{Y}(x),\bar{z}) \ll_{\mathscr{R},\lambda} f(g_1(a),g_2(b),f(g_3(a))).$$

We take the cut $\lambda = 0.6$ and show how $\mathfrak{P}$ computes one of the solutions of this problem, the substitution $\sigma_3 = \{\bar{x} \mapsto g_1(b), x \mapsto h_3(b), \overline{Y} \mapsto f(\circ), \bar{z} \mapsto ()\}$:

$$\{f(\bar{x},x,\overline{Y}(x),\bar{z}) \ll_{\mathscr{R},0.6} f(g_1(a),g_2(b),f(g_3(a)))\};\emptyset;1 \rightsquigarrow_{\mathsf{RFS}}$$
$$\{(\bar{x},x,\overline{Y}(x),\bar{z}) \ll_{\mathscr{R},0.6} (g_1(a),g_2(b),f(g_3(a)))\};\emptyset;1 \rightsquigarrow_{\mathsf{HVE}}$$
$$\{(x,\overline{Y}(x),\bar{z}) \ll_{\mathscr{R},0.6} (g_2(b),f(g_3(a)))\};\{\bar{x} \approx g_1(b)\};0.7 \rightsquigarrow_{\mathsf{TVE}}$$
$$\{(\overline{Y}(x),\bar{z}) \ll_{\mathscr{R},0.6} (f(g_3(a)))\};\{\bar{x} \approx g_1(b), x \approx h_3(b)\};0.6 \rightsquigarrow_{\mathsf{CVE}}$$
$$\{(x,\bar{z}) \ll_{\mathscr{R},0.6} (g_3(a))\};\{\bar{x} \approx g_1(b), x \approx h_3(b), \overline{Y} \approx f(\circ)\};0.6 \rightsquigarrow_{\mathsf{TVE}}$$
$$\{\bar{z} \ll_{\mathscr{R},0.6} ()\};\{\bar{x} \approx g_1(b), x \approx h_3(b), \overline{Y} \approx f(\circ)\};0.6 \rightsquigarrow_{\mathsf{HVE}}$$
$$\emptyset;\{\bar{x} \approx g_1(b), x \approx h_3(b), \overline{Y} \approx f(\circ), \bar{z} \approx ()\};0.6.$$

5 Conclusion

We extended the ρLog calculus with the capabilities to work with strict proximity relations. This extension, called ρLog-prox, can process both crisp and fuzzy data. With the help of the corresponding strategies, the user has full control on how fuzzy (proximity) relations are used. There are no hidden assumptions about fuzziness.

We showed that matching modulo proximity can be naturally embedded in the strategy-based transformation rule framework of ρLog-prox. We developed a proximity matching algorithm for expressions involving four different kinds of variables (for terms, for hedges, for function symbols, and for contexts), and proved its termination, soundness, and completeness.

Acknowledgements This research has been partially supported by the Austrian Science Fund (FWF) under the project 28789-N32 and by the Shota Rustaveli National Science Foundation of Georgia (SRNSFG) under the grant YS-18-1480.

References

1. K. R. Apt. Logic programming. In van Leeuwen [25], pages 493–574.
2. K. R. Apt and R. N. Bol. Logic programming and negation: A survey. *J. Log. Program.*, 19/20:9–71, 1994.
3. W. Belkhir, A. Giorgetti, and M. Lenczner. A symbolic transformation language and its application to a multiscale method. *J. Symb. Comput.*, 65:49–78, 2014.
4. N. Dershowitz and Z. Manna. Proving termination with multiset orderings. *Commun. ACM*, 22(8):465–476, 1979.
5. B. Dundua, M. Florido, T. Kutsia, and M. Marin. *CLP(H):* constraint logic programming for hedges. *TPLP*, 16(2):141–162, 2016.
6. B. Dundua, T. Kutsia, and M. Marin. Strategies in Pρ Log. In M. Fernández, editor, *Proceedings Ninth International Workshop on Reduction Strategies in Rewriting and Programming, WRS 2009, Brasilia, Brazil, 28th June 2009*, volume 15 of *EPTCS*, pages 32–43, 2009.
7. B. Dundua, T. Kutsia, and K. Reisenberger-Hagmayer. An overview of Pρ Log. In Y. Lierler and W. Taha, editors, *Practical Aspects of Declarative Languages - 19th International Symposium, PADL 2017, Paris, France, January 16-17, 2017, Proceedings*, volume 10137 of *Lecture Notes in Computer Science*, pages 34–49. Springer, 2017.
8. F. A. Fontana and F. Formato. A similarity-based resolution rule. *Int. J. Intell. Syst.*, 17(9):853–872, 2002.
9. S. Guadarrama, S. Muñoz-Hernández, and C. Vaucheret. Fuzzy Prolog: a new approach using soft constraints propagation. *Fuzzy Sets and Systems*, 144(1):127–150, 2004.
10. P. Julián-Iranzo and C. Rubio-Manzano. An efficient fuzzy unification method and its implementation into the Bousi∼Prolog system. In *FUZZ-IEEE 2010, IEEE International Conference on Fuzzy Systems, Barcelona, Spain, 18-23 July, 2010, Proceedings*, pages 1–8. IEEE, 2010.
11. P. Julián-Iranzo and C. Rubio-Manzano. Proximity-based unification theory. *Fuzzy Sets and Systems*, 262:21–43, 2015.
12. T. Kutsia. Solving equations with sequence variables and sequence functions. *J. Symb. Comput.*, 42(3):352–388, 2007.
13. T. Kutsia and M. Marin. Matching with regular constraints. In G. Sutcliffe and A. Voronkov, editors, *Logic for Programming, Artificial Intelligence, and Reasoning, 12th International Conference, LPAR 2005, Montego Bay, Jamaica, December 2-6, 2005, Proceedings*, volume 3835 of *Lecture Notes in Computer Science*, pages 215–229. Springer, 2005.
14. R. C. T. Lee. Fuzzy logic and the resolution principle. *J. ACM*, 19(1):109–119, 1972.
15. J. W. Lloyd. *Foundations of Logic Programming, 2nd Edition*. Springer, 1987.
16. M. Marin. A system for rule-based programming in Mathematica. Available from http://staff.fmi.uvt.ro/˜mircea.marin/rholog/, 2019.
17. M. Marin and T. Kutsia. On the implementation of a rule-based programming system and some of its applications. In B. Konev and R. Schmidt, editors, *Proceedings of the 4th International Workshop on the Implementation of Logics (WIL'03)*, pages 55–68, Almaty, Kazakhstan, 2003.
18. M. Marin and T. Kutsia. Foundations of the rule-based system ρLog. *Journal of Applied Non-Classical Logics*, 16(1-2):151–168, 2006.
19. M. Marin and F. Piroi. Rule-based programming with Mathematica. In *Proceedings of the 6th International Mathematica Symposium, Alberta, Canada*, 2004.
20. J. Medina, M. Ojeda-Aciego, and P. Vojtás. Similarity-based unification: a multi-adjoint approach. In J. M. Garibaldi and R. I. John, editors, *Proceedings of the 2nd International Conference in Fuzzy Logic and Technology, Leicester, United Kingdom, September 5-7, 2001*, pages 273–276. De Montfort University, Leicester, UK, 2001.

21. J. Medina, M. Ojeda-Aciego, and P. Vojtás. Similarity-based unification: a multi-adjoint approach. *Fuzzy Sets and Systems*, 146(1):43–62, 2004.
22. P. Nguyen. *Meta-mining: a meta-learning framework to support the recommendation, planning and optimization of data mining workflows*. PhD thesis, Department of Computer Science, University of Geneva, 2015.
23. L. D. Raedt and A. Kimmig. Probabilistic (logic) programming concepts. *Machine Learning*, 100(1):5–47, 2015.
24. M. I. Sessa. Approximate reasoning by similarity-based SLD resolution. *Theor. Comput. Sci.*, 275(1-2):389–426, 2002.
25. J. van Leeuwen, editor. *Handbook of Theoretical Computer Science, Volume B: Formal Models and Semantics*. Elsevier and MIT Press, 1990.
26. S. Wolfram. *The Mathematica book, 5th Edition*. Wolfram-Media, 2003.
27. B. Yang, W. Belkhir, R. N. Dhara, M. Lenczner, and A. Giorgetti. Computer-aided multiscale model derivation for MEMS arrays. In *Proc. 12th Int. Conf. Thermal, Mechanical and Multi-Physics Simulation and Experiments in Microelectronics and Microsystems*. IEEE Computer Society, 2011.