JOHANNES KEPLER
UNIVERSITY LINZ

Submitted by
Dipl.-Ing.
Ioana-Cleopatra Pau

Submitted at
Research Institute for
Symbolic Computation

Supervisor and
First Evaluator
Priv.-Doz. Dr.
Temur Kutsia

Second Evaluator
Prof. Dr. Mircea Marin

May 2022

Symbolic Techniques for Approximate Reasoning

Doctoral Thesis
to obtain the academic degree of
Doktorin der technischen Wissenschaften
in the Doctoral Program
Technische Wissenschaften

Eidesstattliche Erklärung

Kurzfassung

Begründungen mit unvollständigen und unzulänglichen Informationen treten sehr häufig in der menschlichen Kommunikation auf. Ihre Modellierung ist eine hoch nichttriviale Aufgabe und ist ein Problem in vielen Anwendungen. Es gibt viele Umschreibungen für solche Informationen (z.B. Ungewissheit, Ungenauigkeit, Vagheit, Unschärfe) und Methoden wurden vorgeschlagen, um damit entsprechend umzugehen (z.B., Default-Logik, Wahrscheinlichkeiten, unscharfe Mengen, etc.)

Viele Probleme in diesem Gebiet werden in Theorien formuliert, in denen die exakte Gleichheit durch eine quantitative Annäherung ausgetauscht wird. Unscharfe Nachbarschaft und Ähnlichkeit sind bemerkenswerte Beispiele von solchen Erweiterungen, in denen die Gleichheit von Symbolen mit deren Nachbarschaft und Ähnlichkeit bis zu einem gewissen Grad ersetzt werden. Näherungsbeziehungen sind unscharfe Analoga von Toleranzrelationen (reflexiv, symmetrisch aber nicht notwendigerweise transitiv) während Ähnlichkeiten unscharfe Gegenstücke von Äquivalenzrelationen sind. Dabei erfordern die Beweismethoden und computerunterstützten Werkzeuge solcher Theorien grundlegende Techniken welche mit quantitativen Informationen arbeiten können.

In dieser Arbeit wenden wir uns dem Problem zu, solche fundamentalen Techniken für verschiedene Arten von unscharfen und annähernden Nebenbedingungen zu erarbeiten, welche mit zwei zentralen Ansätzen bezüglich ihrer Lösbarkeit charakterisiert werden können: eine basiert auf Nachbarschaftsblöcke die andere auf Nachbarschaftsklassen. Unsere Methoden sind meistens (aber nicht exklusiv) klassenbasiert. Wir führen den entsprechenden theoretischen Hintergrund ein und entwickeln und analysieren Algorithmen zum Lösen von Constraints. Zum einen decken diese das Lösen von näherungsweisen Gleichungssystemen (notwendig innerhalb von automatischer Deduktion, deklarativem Programmieren, oder Ersetzungs- und Transformationsprozessen) ab. Zum anderen erlauben diese Algorithmen näherungsweise Verallgemeinerungen von Constraints (nützlich in näherungsweiser induktiver Logik, analogbasiertem Schließen und Programmieren, der Entdeckung von Ähnlichkeiten von Befehlen in Programmiersprachen oder Texten der natürlichen Sprache). Eine Anwendung in regelbasierter Programmierung wird ebenfalls vorgestellt.

Im Zusammenhang von näherungsweisen Gleichungen stellen wir neue Algorithmen der Unifikation und Matching für Nachbarschaftsrelationen vor, wobei die Nichtübereinstimmung von Symbolnamen und auch ihrer Stelligkeit (völlig unscharfe Signaturen) erlaubt sind. Ebenso erarbeiten wir einen Algorithmus zum Lösen von mehrfachen Ähnlichkeitsrelationen. Für Generalisierungsconstraints untersuchen wir nachbarschaftsbasierte Anti-Unifikationsmethoden und stellen einen allgemeinen Rahmen für mehrere Algorithmen für Verallgemeinerungen in völlig unscharfen Signaturen vor. In dem block-basierten Ansatz für die näherungsweise Anti-Unifikation tritt ein interessantes Unterproblem auf, in dem alle maximalen Cliquenpartitionen in ungerichteten Graphen berechnet werden. Die Eigenschaften der Terminierung, Korrektheit und Vollständigkeit von allen entwickelten Algorithmen werden gezeigt und ihr Potential bezüglich neuer Anwendungen wird illustriert.

Abstract

Reasoning with incomplete, imperfect information is very common in human communication. Its modeling is a highly nontrivial task, and remains an important issue in many applications. There are various notions associated to such information (e.g., uncertainty, imprecision, vagueness, fuzziness) and different methodologies have been proposed to deal with them (e.g., approaches based on default logic, probability, fuzzy sets, etc.)

Many problems in this area are formulated in theories where the exact equality is replaced by its quantitative approximation. Fuzzy proximity and similarity relations are notable examples of such extensions, where instead of symbol equalities one talks about their proximity or similarity up to a certain degree. Proximity relations are fuzzy analogs of tolerance (reflexive, symmetric, but not necessarily transitive) relations, while similarities are fuzzy counterparts of equivalence relations. Reasoning methods and computational tools for such theories require fundamental techniques that deal with quantitative information.

In this thesis, we address the problem of developing such fundamental techniques for various kinds of fuzzy approximation constraints, characterizing two major approaches towards solving them: one based on proximity blocks and another one based on proximity classes. Our methods are mostly (but not exclusively) class-based ones. We introduce the corresponding theoretical background and design and analyze constraint solving algorithms, including those for solving systems of approximate equations (needed to perform a step in approximate automated deduction, declarative programming, or in rewriting and transformation process) and algorithms for solving approximate generalization constraints (useful for approximate inductive reasoning, reasoning and programming by analogy, similarity detection in programming language statements or in natural language texts). An application use case in rule-based programming is also discussed.

In the context of approximate equations, we designed unification and matching algorithms for proximity relations where mismatches are allowed in symbol names and also in their arities (fully fuzzy signatures), and a constraint solving algorithm for multiple similarity relations. For generalization constraints, we studied proximity-based anti-unification and developed a generic framework for multiple generalization algorithms in fully fuzzy signatures. Block-based approach to approximate anti-unification involved solving an interesting related problem of computing all maximal clique partitions in undirected graphs. Termination, soundness, and completeness properties of all the developed algorithms are proved, complexities are investigated, and their application potential is illustrated.

Aknowledgements

Thank you Mom! Thank you for showing me repeatedly that a strong woman can make the impossible possible. Thank you also for making mathematics such a happy place for me. And for steering me towards computer science. I wrote this thesis because of you and for you. After all, I had to make an honest woman out of you again. :)

Thank you Peter, for re-opening the science door for me.

Thank you Carsten for being such a kind friend and translating the abstract in German.

Thank you Puiu, thank you Temo. Without any of you this would not have been possible.

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

Parts of this work have been published in [23, 24, 45, 46, 47, 48, 49, 62].

Contents

Chapter 1

Introduction

In this work we present symbolic techniques (unification, matching, anti-unification) that are fundamental for automated approximate reasoning (proving, solving, and computing). Reasoning with incomplete, imperfect information is very common in human communication. Its modeling is a highly nontrivial task, and remains an important issue in applications of artificial intelligence. There are various notions associated to such information (e.g., uncertainty, imprecision, vagueness, fuzziness) and different methodologies have been proposed to deal with them (e.g., approaches based on default logic, probability, fuzzy sets, etc.)

For many problems in this area, exact equality is replaced by its approximation. Some approaches use proximity relations, others use similarity relations to express the approximation, modeling the corresponding imprecise information.

Proximity relations are reflexive and symmetric fuzzy binary relations, whose crisp (two-valued) counterpart are tolerance relations. The latter was deemed by Poincaré [64] as having a fundamental importance in distinguishing mathematics applied to the physical world from ideal mathematics. Introduced in [19], the proximity relations generalize similarity relations (a fuzzy version of equivalence), by dropping transitivity. Proximity relations help represent fuzzy information in situations where similarity is not adequate, providing more flexibility in expressing vague knowledge.

Unification, matching, and anti-unification are fundamental operations for many areas of symbolic computation. Unification and matching are central computational mechanisms in fields such as automated reasoning, rewriting, declarative programming. Anti-unification is a logic-based method for computing generalizations, with a wide range of applications, e.g. in inductive and analogical reasoning, program analysis, natural language processing.

Unification and matching are methods for solving equations between terms. Anti-unification aims at detecting similarities between different objects and at learning general structures from concrete instances. Unification and anti-unification can be seen as dual techniques, since unification computes a most specific common instance of given logical expressions, while anti-unification computes their least general generalization.

These techniques have been studied intensively for crisp equivalence relations. First-order syntactic and equational unification and matching have been considered, e.g., in [7, 8, 37, 61, 67, 70, 71], for first-order syntactic and equational anti-unification see, e.g., [63, 66, 4, 9, 43].

However, these techniques fail or overgeneralize when there is no match between two corresponding function symbols of the given terms. While in many situations this is the desired outcome, there are cases when some tolerance regarding the mismatches would offer a better result. The type of the accepted differences can vary, and some mismatches were already explored in the fuzzy context, concerning reasoning with imprecise, vague information, although not as extensively as in the crisp context.

Unification for similarity relations was studied in [28, 29, 27, 69] in the context of fuzzy logic programming. In [1], the authors extended the algorithm from [69] to fully fuzzy signature (permitting arity mismatches between function symbols) and studied also anti-unification.

Investigations of symbolic techniques for proximity relations have been started recently, and not many works have addressed them so far. Probably one of the earliest ones is [68], where the authors introduced a constraint logic programming schema with proximity relations.

Proximity relations (and their crisp counterpart, tolerance relations) can be represented by weighted or non-weighted undirected graphs, and the existing approaches to proximity-based constraint solving can be characterized by the way how proximal nodes are treated in such a graph. In the block-based approach, two symbols are considered proximal if they belong to the same maximal clique partition in this graph. In the class-based approach, a symbol is proximal to any of its neighbors in the graph. Both approaches can be used to extend the constraint solving methods from a single similarity relation to proximity or multiple similarity relations and have their advantages and disadvantages.

In the block-based approach to unification with proximity relations, the pioneering contribution is [33], which generalizes the similarity-based unification algorithm from [69] to proximity relations. This work was further extended in [35, 17], and used in the fuzzy logic programming system Bousi∼Prolog.

In this thesis we focus mostly on the class-bases approach. The only exception is our work from [45, 46], described in **Section 3.3**.

In this section, we study a block-based approach to anti-unification with proximity relations. To ensure that the algorithm computes a minimal complete set of generalizations, we need to consider disjoint blocks of symbols in the given proximity relation, which effectively refines it into a similarity relation. In order to compute all such refinements, we develop an algorithm that produces all maximal clique partitions of an undirected graph (that corresponds to the proximity relation). This algorithm is optimal in the sense that each maximal clique partition is computed only once, and generating and discarding false answers is avoided. It is incorporated into the anti-unification algorithm and generates partitions lazily, only on demand. We prove termination, soundness, and completeness of both algorithms.

Chapter 4 is devoted to the class-based techniques for proximity relations with mismatches permitted between symbol names with the same arities.

In **Section 4.2** we consider the class-based approach to unification for such relations. We develop an algorithm which computes a compact representation of the set of solutions. Considering neighborhoods of function symbols as finite sets, we work with term representation where in place of function symbols we permit neighborhoods or names. The latter are some kind of variables, which stand for unknown neighborhoods. The algorithm is split into two phases. In the first one, which is a generalization of syntactic unification for proximity relations, we produce a substitution together with two sets of constraints: over variables and over neighborhoods. A crucial step in the algorithm is variable elimination, which is done not with a term to which a variable should be unified, but with a copy of that term with fresh names and variables. This step also introduces new neighborhood constraints to ensure that the copy of the term remains close to its original, thus storing the proximity chains between terms.

In the second phase, the neighborhood constraint is solved by a dedicated algorithm. Combining each solution from the second phase with the substitution computed in the first phase and a solution of the variables constraint (which always exists), we obtain a compact representation of the minimal complete set of unifiers of the original problem. We prove termination, soundness and completeness of both algorithms. This is the first detailed study of class-based proximity-based unification. Its early version has been published in [48].

In **Section 4.3**, we develop a dedicated algorithm for class-based matching with proximity relations. 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. These results have been published in [47].

Section 4.4 is about class-based anti-unification for proximity 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. These results appeared in [47].

In **Chapter 5** we make a step further and consider a more general case of proximity relations in fully fuzzy signatures, where mismatches are allowed not only between proximal symbol names, but also between their arities. Arguments of such symbols are declared to be proximal via given argument relations.

Section 5.3 is dedicated to the development of class-based unification for such proximity relations, which generalizes, on the one hand, the proximity-based unification from Section 4.2 and, on the other hand, the similarity-based unification in fully fuzzy signatures from [2]. Argument relations are correspondence (i.e., left- and right-total) relations, which are not required to be functional. It is a very flexible approach, which opens a way to extending proximity-based unification towards special equational theories. We design a corresponding unification algorithm and prove its termination, soundness, and completeness. Its original version has been published in [62].

The matching algorithm for the same setting is studied in **Section 5.4**. Argument relations are not restricted. We prove that the algorithm is terminating, sound, and complete. It generalizes the matching algorithm from Section 4.3 and was published in [62].

In **Section 5.5** we study the anti-unification problem for the same class of proximity relations. For mismatching arguments, we consider four different variants of argument relations between different proximal symbols:

unrestricted relations / functions, and correspondence relations / functions. We design the corresponding algorithms, study their properties, and show how to obtain the existing fuzzy anti-unification problems as special cases of our problems. This gives a flexible generic framework for proximity-based anti-unification, which is published in [49].

Chapter 6 describes an extension of the class-based matching algorithm from Section 4.3 adapted to serve as a computation mechanism of a rule-based programming tool. The tool is based on ρLog [55], which is a calculus for conditional transformation of sequences of expressions, controlled by strategies. Its language is richer than what we considered in the previous chapters. 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. At the core of ρLog there is a powerful pattern matching algorithm [44].

ρLog-prox, described in this chapter, is an attempt to combine approximate reasoning and strategic rule-based programming. It extends ρLog with capabilities to process imprecise information represented by proximity relations. 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. We prove that it 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 results of this chapter have been published in [24].

In **Chapter 7**, we move the focus from proximity to similarity. The previous works on similarity-based unification usually assume a single similarity relation. However, in many practical situations, one needs to deal with several similarities between the objects from the same set. Multiple similarities pose challenges to constraint solving, since we can not rely on the transitivity property anymore. Note that proximity relations are not transitive either, but their unification methods have some limitations in dealing with multiple similarities simultaneously.

In this chapter, we address this problem. Since the existing similarity-

and proximity-based unification techniques cannot adequately solve it, we propose a special algorithm for constraints over multiple similarity relations.

Our algorithm does not depend on application or implementation preferences. It can be incorporated in a modular way in the constraint logic programming schema, can be used for constrained rewriting, querying, or similar purposes. It combines three parts: solving syntactic equations, solving similarity problems for one relation, and solving mixed problems. We permit not only variables for terms, but also variables for function symbols, since they are necessary in the process of finding an "intermediate object" between terms in different similarity relations. We prove termination, soundness and completeness theorems. This work has been published in [23].

Fig. 1.1 depicts our contribution (colored nodes) and the related work (white nodes) in the area of approximate unification, matching, and anti-unification, as well as the relationships between the problems solved.

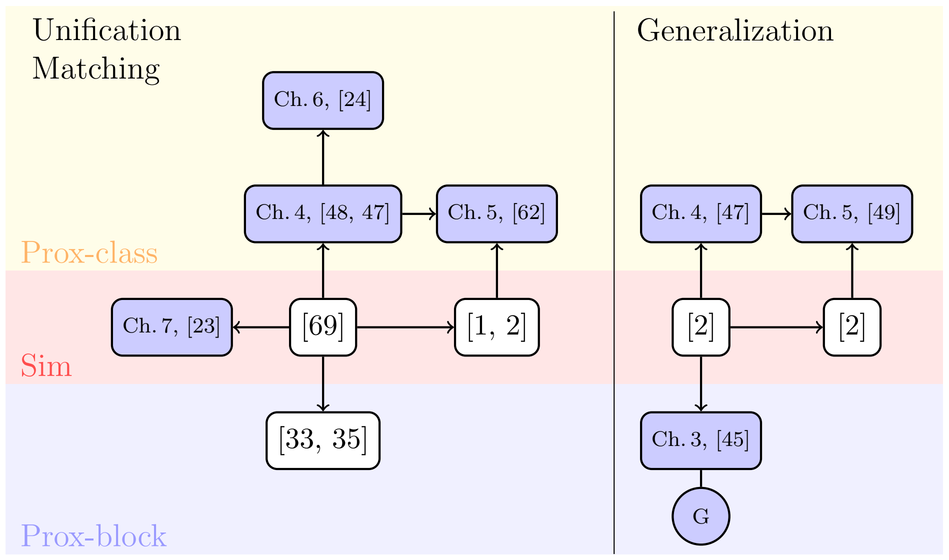


Figure 1.1: Problems, their relationships, and references.

The problems can be categorized from two different points of view:

- the type of the considered fuzzy relation, illustrated in the horizontal layers corresponding to the similarity relations (Sim), and the two perspectives with respect to the proximity relations (Prox-block and Prox-class),

- the type of constraints solved by the problem, separated vertically, with equational constraints (addressed by unification and matching) on the left, and generalization constraints (addressed by anti-unification/generalization) on the right.

The figure is also a clear overview of all the ways in which problems generalize other problems, which were previously mentioned in the description of the whole content. The relationships are reproduced through arrows, in which the direction moves from a more specific to a more general problem. One can thus easily discern how the proximity-based techniques generalize and extend the similarity-based ones.

The vertical classification creates a symmetry that reflects the dual nature of unification and anti-unification problems in the same fuzzy setting.

Chapter 2

Preliminaries

We introduce here technical notions that will be used throughout the whole thesis. The notions specific only to a certain chapter will be defined inside each chapter in dedicated sections.

2.1 Fuzzy relations

We define the basic notions about proximity relations according to [33] and about similarity relations following [69].

A binary *fuzzy relation* on a set S is a mapping from $S \times S$ to the real interval $[0,1]$. A fuzzy relation is characterized by a set $\Lambda = \{\lambda_1, \ldots, \lambda_n \mid 0 < \lambda_i \leqslant 1\}$ 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 $\mathcal{R}$ on S, denoted $\mathcal{R}_\lambda$, is an ordinary (crisp) relation on S defined as $\mathcal{R}_\lambda := \{(s_1, s_2) \mid \mathcal{R}(s_1, s_2) \geqslant \lambda\}$.

A T-norm $\wedge$ in $[0,1]$ is a binary operation $\wedge : [0;1] \times [0,1] \to [0,1]$, which is associative, commutative, non-decreasing 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 [39]. In the thesis, we take the minimum in the role of the T-norm $\wedge$, which sometimes is even spelled out explicitly.

2.1.1 Proximity and similarity relations

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

Reflexivity: $\mathcal{R}(s,s) = 1$ for all $s \in S$;

Symmetry: $\mathcal{R}(s_1, s_2) = \mathcal{R}(s_2, s_1)$ for all $s_1, s_2 \in S$.

In this thesis we consider only strict proximity relations:

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

Tolerance relations are crisp reflexive and symmetric binary relations. A λ-cut of a proximity relation on S is a tolerance relation on S.

A proximity relation (on S) is called a *similarity relation* (on S) if it is transitive:

Transitivity: $\mathcal{R}(s_1, s_2) \geqslant \mathcal{R}(s_1, s) \wedge \mathcal{R}(s, s_2)$ for any $s_1, s_2, s \in S$.

A λ-cut of a similarity relation on S is an equivalence relation on S.

2.1.2 Blocks and classes

While similarity relations on sets define partitions of those sets, given by the similarity (fuzzy equivalence) classes, proximity relations are more complicated in this respect. For proximity we have two different, but related concepts: proximity blocks and proximity classes. They are defined below.

Definition 2.1.1. *(Proximity block of level λ). Given a proximity relation $\mathcal{R}$ on a set S and $\lambda \in (0, 1]$, a* proximity block of level λ *(or, shortly, a* λ-block*) is a subset B of S such that the restriction of $\mathcal{R}_\lambda$ to B is a total relation, and B is maximal with this property.*

Definition 2.1.2. *(Proximity class of level λ). Given a proximity relation $\mathcal{R}$ on a set S and $\lambda \in (0, 1]$, the* proximity class of level λ of $s \in S$ *(or, shortly,* λ-class *of s) is the set $\mathbf{pc}(s, \mathcal{R}, \lambda) = \{s' \mid \mathcal{R}(s, s') \geqslant \lambda\}$.*

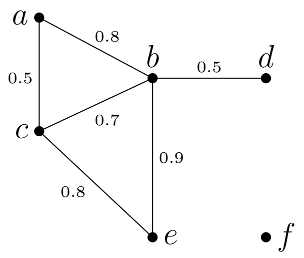
The blocks and the classes of a proximity relation cover completely the set S, but they are not necessarily partitions of S.

The following example illustrates proximity blocks:

Example 2.1.1. Let $S = \{a, b, c, d, e, f\}$ and $\mathcal{R}$ be a proximity relation on S defined by

$$\begin{aligned} &\mathcal{R}(a, b) = 0.8, \qquad &&\mathcal{R}(b, c) = 0.7, \qquad &&\mathcal{R}(c, e) = 0.8, \\ &\mathcal{R}(a, c) = 0.5, &&\mathcal{R}(b, e) = 0.9, &&\mathcal{R}(b, d) = 0.5. \end{aligned}$$

It can be visualized as shown, e.g., in the following diagram:



The set approximation levels of $\mathcal{R}$ is $\Lambda = \{1, 0.9, 0.8, 0.7, 0.5\}$.

The table below shows all λ-blocks for each approximation level λ:

λ	λ-blocks					
1	$\{a\}$	$\{b\}$	$\{c\}$	$\{d\}$	$\{e\}$	$\{f\}$
0.9	$\{a\}$	$\{b,e\}$	$\{c\}$	$\{d\}$	$\{f\}$	
0.8	$\{a,b\}$	$\{b,e\}$	$\{c,e\}$	$\{d\}$	$\{f\}$	
0.7	$\{a,b\}$	$\{b,c,e\}$	$\{d\}$	$\{f\}$		
0.5	$\{a,b,c\}$	$\{b,c,e\}$	$\{b,d\}$	$\{f\}$		

The λ-classes for each approximation level λ are shown in the following table:

λ	$\mathbf{pc}(a,\mathcal{R},\lambda)$	$\mathbf{pc}(b,\mathcal{R},\lambda)$	$\mathbf{pc}(c,\mathcal{R},\lambda)$	$\mathbf{pc}(d,\mathcal{R},\lambda)$	$\mathbf{pc}(e,\mathcal{R},\lambda)$	$\mathbf{pc}(f,\mathcal{R},\lambda)$
1	$\{a\}$	$\{b\}$	$\{c\}$	$\{d\}$	$\{e\}$	$\{f\}$
0.9	$\{a\}$	$\{b,e\}$	$\{c\}$	$\{d\}$	$\{b,e\}$	$\{f\}$
0.8	$\{a,b\}$	$\{a,b,e\}$	$\{c,e\}$	$\{d\}$	$\{b,c,e\}$	$\{f\}$
0.7	$\{a,b\}$	$\{a,b,c,e\}$	$\{b,c,e\}$	$\{d\}$	$\{b,c,e\}$	$\{f\}$
0.5	$\{a,b,c\}$	$\{a,b,c,d,e\}$	$\{a,b,c,e\}$	$\{b,d\}$	$\{b,c,e\}$	$\{f\}$

2.2 Terms and substitutions

2.2.1 Terms

We consider a first-order alphabet consisting of a set of fixed arity function symbols $\mathcal{F}$ and a set of variables $\mathcal{V}$, with $\mathcal{F}$ and $\mathcal{V}$ disjoint.

The set of terms $\mathcal{T}(\mathcal{F},\mathcal{V})$ over $\mathcal{F}$ and $\mathcal{V}$ is defined in the standard way: $t \in \mathcal{T}(\mathcal{F},\mathcal{V})$ iff t is defined by the grammar $t := x \mid f(t_1,\ldots,t_n)$, where $x \in \mathcal{V}$ and $f \in \mathcal{F}$ is an n-ary function symbol with $n \geqslant 0$.

We denote arbitrary function symbols by f, g, h, p, q, constants (0-ary function symbols) by a, b, c, d, e, variables by x, y, z, u, v, w, and terms by s, t, r. The *head* of a term is defined as $\mathit{head}(x) := x$ and $\mathit{head}(f(t_1, \ldots, t_n)) := f$. For a term t, we denote with $\mathcal{V}(t)$ the set of all variables appearing in t. If $\mathcal{V}(t) = \varnothing$, we call t a *ground* term. A term is called *linear* if no variable occurs in it more than once.

For a term t, its *set of positions* $\mathit{pos}(t)$ is a set of sequences of positive integers defined as follows: If t is a variable, then $\mathit{pos}(t) = \{\epsilon\}$, where ϵ is the empty sequence; If $t = f(s_1, \ldots, s_n)$, then $\mathit{pos}(t) = \{\epsilon\} \cup \bigcup_{i=1}^{n} \{i.p \mid p \in \mathit{pos}(s_i)\}$. By $t[p]$ we denote the symbol in t at position p. The notation $t|_p$ denotes the subterm of t at position p.

2.2.2 Substitutions

Substitutions over $\mathcal{T}(\mathcal{F}, \mathcal{V})$ are mappings from variables to terms, where all but finitely many variables are mapped to themselves. We use the Greek letters $\sigma, \vartheta, \varphi$ to denote substitutions, except for the identity substitution which is written as Id.

The *domain* of a substitution σ is defined as $\mathit{dom}(\sigma) := \{x \mid \sigma(x) \neq x\}$. We represent substitutions with the usual set notation: σ is represented as $\{x \mapsto \sigma(x) \mid x \neq \sigma(x)\}$. The *restriction* of a substitution σ on a set of variables V, denoted by $\sigma|_V$, is the substitution defined as $\sigma|_V(x) = \sigma(x)$ when $x \in V$ and $\sigma|_V(x) = x$ otherwise.

Application of a substitution σ to a term t, written in the postfix notation as $t\sigma$, is defined recursively as $x\sigma := \sigma(x)$ and $f(t_1, \ldots, t_n)\sigma := f(t_1\sigma, \ldots, t_n\sigma)$. Substitution *composition* is defined as a composition of mappings, and we write $\sigma\vartheta$ for the composition of σ with ϑ. The operation is associative but not commutative.

2.2.3 Extending proximity relations over terms

Let $\mathcal{R}$ be a proximity relation defined on $\mathcal{F}$, so that for all $f, g \in \mathcal{F}$, we have $\mathcal{R}(f, g) = 0$ if $\mathit{arity}(f) \neq \mathit{arity}(g)$. 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:

  $$\mathcal{R}(f(t_1,\dots,t_n),g(s_1,\dots,s_n)) = \mathcal{R}(f,g) \wedge \mathcal{R}(t_1,s_1) \wedge \cdots \wedge \mathcal{R}(t_n,s_n),$$

  when f and g are both n-ary.

- In all other cases, $\mathcal{R}(\mathtt{t}_1,\mathtt{t}_2) = 0$ for $\mathtt{t}_1,\mathtt{t}_2 \in \mathcal{F} \cup \mathcal{T}(\mathcal{F},\mathcal{V})$.

In the rest of the thesis, whenever the terms do not comply with the above restriction or are further extended (such as X-terms later), the extension of the proximity relation will be adapted in the corresponding chapter or section.

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

When $\mathcal{R}$ is strict on $\mathcal{F}$, its extension to $\mathcal{F} \cup \mathcal{T}(\mathcal{F},\mathcal{V})$ is also strict. When $\lambda = 1$, the relation $\simeq_{\mathcal{R},\lambda}$ does not depend on $\mathcal{R}$ due to strictness of the latter and is just the syntactic equality $=$.

2.2.4 Comparing terms and substitutions

Definition 2.2.1 (Relations $\lesssim_{\mathcal{R},\lambda}$ and $\preceq$). *The relations $\lesssim_{\mathcal{R},\lambda}$ and $\preceq$ and the corresponding notions are defined as follows:*

$\lesssim_{\mathcal{R},\lambda}$ **for terms:** *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 $t \lesssim_{\mathcal{R},\lambda} s$, 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.*

$\lesssim_{\mathcal{R},\lambda}$ **for substitutions:** *A substitution ϑ is* $(\mathcal{R},\lambda)$-more general than *φ (or φ is an* $(\mathcal{R},\lambda)$-instance *of ϑ), written $\vartheta \lesssim_{\mathcal{R},\lambda} \varphi$, if there exists a substitution σ such that $x\vartheta\sigma \simeq_{\mathcal{R},\lambda} x\varphi$ for all x.*

$\preceq$ **for terms:** *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 $t \preceq s$, if there exists a substitution σ such that $t\sigma = s$. We say that σ is a syntactic matcher of t to s.*

$\preceq$ **for substitutions:** *A substitution ϑ is* syntactically more general than *φ (or φ is a* syntactic instance *of ϑ), written $\vartheta \preceq \varphi$, if there exists a substitution σ such that $x\vartheta\sigma = x\varphi$ for all x.*

The strict part of $\lesssim_{\mathcal{R},\lambda}$ is denoted by $\prec_{\mathcal{R},\lambda}$. The strict part of $\preceq$ is denoted by $\prec$.

The relation $\lesssim_{\mathcal{R},\lambda}$ 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 $a \lesssim_{\mathcal{R},\lambda} b$, $b \lesssim_{\mathcal{R},\lambda} c$, and $a \not\lesssim_{\mathcal{R},\lambda} c$. Unlike $\lesssim_{\mathcal{R},\lambda}$, $\preceq$ is transitive. In fact, $\preceq$ is a quasi-ordering, called instantiation quasi-ordering. We also have $\preceq \; \subseteq \; \lesssim_{\mathcal{R},\lambda}$ for any $\mathcal{R}$ and λ.

Two substitutions σ and ϑ are called *equigeneral* iff $\sigma \preceq \vartheta$ and $\vartheta \preceq \sigma$. In this case we write $\sigma \simeq \vartheta$. It is an equivalence relation.

2.3 Unification, matching, anti-unification

2.3.1 Unification problems

Unification is a process of solving term equations. We write an $(\mathcal{R},\lambda)$-*equation* between t and s as $t \simeq^?_{\mathcal{R},\lambda} s$, with the question mark indicating that the equation is supposed to be solved (i.e., the terms t and s to be $(\mathcal{R},\lambda)$-*unified*). A *solution* (*unifier*) of such an equation is a substitution σ such that $t\sigma \simeq_{\mathcal{R},\lambda} s\sigma$. We say that $\mathcal{R}(t\sigma, s\sigma) \geqslant \lambda$ is the *approximation degree* of $(\mathcal{R},\lambda)$-unifying t and s by σ (or, equivalently, the *approximation degree* of solving $t \simeq^?_{\mathcal{R},\lambda} s$ by σ).

An $(\mathcal{R},\lambda)$-*unification problem* (or, briefly, a *unification problem*) is a finite set of $(\mathcal{R},\lambda)$-equations. A *solution* (*unifier*) of an $(\mathcal{R},\lambda)$-unification problem P is a substitution that solves all the equations in P. We say that σ is a *most general* $(\mathcal{R},\lambda)$-*unifier* ($(\mathcal{R},\lambda)$-*mgu*) of P if σ is a $(\mathcal{R},\lambda)$-unifier of P and no other $(\mathcal{R},\lambda)$-unifier of P is syntactically strictly more general than σ, i.e., for no other $(\mathcal{R},\lambda)$-unifier ϑ of t and s we have $\vartheta \prec \sigma$. The *approximation degree of the unification* of P by σ, denoted by $deg(P\sigma)$, is $\wedge_{eq\in P} deg(eq\sigma)$, where $deg(eq\sigma)$ is the approximation degree of solving $eq \in P$ by σ.

Instead of writing "a unifier of an $(\mathcal{R},\lambda)$-unification problem P", we often shortly say "an $(\mathcal{R},\lambda)$-unifier of P". The same applies to solutions. Sometimes we may completely skip $(\mathcal{R},\lambda)$ if it does not cause confusions.

Remark 2.3.1. Note that $\preceq$ preserves good properties of unifiers while $\lesssim_{\mathcal{R},\lambda}$ does not. Namely, if σ is a $(\mathcal{R},\lambda)$-unifier of P, then so is any ϑ for which $\sigma \preceq \vartheta$ holds. However, if $\sigma \lesssim_{\mathcal{R},\lambda} \vartheta$, then ϑ might not be an $(\mathcal{R},\lambda)$-unifier of P. A simple example of the latter is $P = \{x \simeq_{\mathcal{R},\lambda} a\}$ and $\mathcal{R}_\lambda = \{(a,b),(b,c)\}$. Then $\sigma = \{x \mapsto b\}$ is an $(\mathcal{R},\lambda)$-unifier of P, but $\vartheta = \{x \mapsto c\}$ is not. However, $\sigma \lesssim_{\mathcal{R},\lambda} \vartheta$.

Definition 2.3.1 (Complete set of unifiers). *Given a proximity relation $\mathcal{R}$, a cut value λ, and an $(\mathcal{R},\lambda)$-proximity unification problem P, the set of substitutions Σ is a* complete set of $(\mathcal{R},\lambda)$-unifiers *of P if the following conditions hold:*

Soundness: *Every substitution $\sigma \in \Sigma$ is an $(\mathcal{R},\lambda)$-unifier of P.*

Completeness: *For any $(\mathcal{R},\lambda)$-unifier ϑ of P, there exists $\sigma \in \Sigma$ such that $\sigma \preceq \vartheta$.*

Σ is *a minimal complete set of unifiers* of P if it is its complete set of unifiers and, in addition, the following condition holds:

Minimality: No two elements in Σ are comparable with respect to $\preceq$: For all $\sigma, \vartheta \in \Sigma$, if $\sigma \preceq \vartheta$, then $\sigma = \vartheta$.

Taking into account the remark above, it should not be surprising that we used $\preceq$ in the completeness and minimality parts of this definition. However, one should be aware that elements of the minimal complete set of unifiers might be $\lesssim_{\mathcal{R},\lambda}$-comparable. For instance, under this definition, $\{x \simeq_{\mathcal{R},0.5} b\}$ for $\mathcal{R}(a,b) = 0.6$, $\mathcal{R}(b,c) = 0.5$ has a minimal complete set of unifiers $\{\{x \mapsto a\}, \{x \mapsto b\}, \{x \mapsto c\}\}$. The substitutions $\{x \mapsto a\}$ and $\{x \mapsto b\}$ are $\preceq$-incomparable, but $\lesssim_{\mathcal{R},\lambda}$-comparable. The same is true for $\{x \mapsto a\}$ and $\{x \mapsto c\}$.

2.3.2 Matching problems

An equation with a ground side is called a *matching equation*. We write $t \lesssim^{?}_{\mathcal{R},\lambda} s$ for an $(\mathcal{R},\lambda)$-matching equation between t and s, where s is the ground side. A *solution* (*matcher*) of such an equation is an $(\mathcal{R},\lambda)$-matcher of t to s, i.e., a substitution σ that matches t to s with respect to $\mathcal{R}$ and λ, i.e., $t\sigma \simeq_{\mathcal{R},\lambda} s$. The number $\mathcal{R}(t\sigma, s) \geqslant \lambda$ is the corresponding *approximation degree*.

Analogously to unification, we will use the notion of *matching problems* and their *most general solution*, and write $(\mathcal{R},\lambda)$-*mgm* for most general $(\mathcal{R},\lambda)$-matchers.

2.3.3 Generalization problems

Definition 2.3.2 ($(\mathcal{R},\lambda)$-lgg). *A term r is called an $(\mathcal{R},\lambda)$*-least general generalization *(an $(\mathcal{R},\lambda)$*-lgg*) of terms t and s iff*

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

Theorem 2.3.1. *If r is an $(\mathcal{R}, \lambda)$-generalization of t, then any syntactic generalization r' of r is also an $(\mathcal{R}, \lambda)$-generalization of t, i.e., $r \precsim_{\mathcal{R},\lambda} t$ and $r' \preceq r$ imply $r' \precsim_{\mathcal{R},\lambda} t$.*

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

Corollary 2.3.1.1. *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 $\preceq$. The syntactic lgg of two terms is unique modulo variable renaming, see, e.g., [63, 66]. 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 $r \preceq r'$.

Example 2.3.1. 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 the $(\mathcal{R}, \lambda)$-lgg of a and c is b, while their syntactic lgg is x.

Definition 2.3.3 (Minimal complete set of $(\mathcal{R}, \lambda)$-generalizations)**.** *Given $\mathcal{R}$, λ, t_1, and t_2, a set of terms T is a* complete set of $(\mathcal{R}, \lambda)$-generalizations *of t_1 and t_2 if*

(a) every $r \in T$ is an $(\mathcal{R}, \lambda)$-generalization of t_1 and t_2,

(b) if r' is an $(\mathcal{R}, \lambda)$-generalization of t_1 and t_2, then there exists $r \in T$ such that $r' \preceq r$ (note that we use syntactic generalization here).

In addition, T is minimal, if it satisfies the following property:

(c) if $r, r' \in T$, $r \neq r'$, then neither $r \prec_{\mathcal{R},\lambda} r'$ nor $r' \prec_{\mathcal{R},\lambda} r$.

A minimal complete set of $(\mathcal{R}, \lambda)$-generalizations *($(\mathcal{R}, \lambda)$-mcsg) of two terms is unique modulo variable renaming. The elements of the $(\mathcal{R}, \lambda)$-mcsg of t_1 and t_2 are $(\mathcal{R}, \lambda)$-lggs of t_1 and t_2.*

This definition directly extends to generalizations of finitely many terms.

Chapter 3

Block-Based Symbolic Techniques for Proximity Relations

In this chapter we present unification and study anti-unification algorithms in the block-based proximity setting. In this setting, mismatches between function symbol names (but not in the arities) are tolerated, but once a symbol a is considered 'close' to another one, say, b, then a cannot be 'close' anymore to the symbols that are *not proximal* to b, even if a was originally 'close' to them. The presented algorithms follow thus a so-called 'choosing sides' strategy. This type of strategy was first applied by Julián-Iranzo et al [33] in their unification algorithm, presented below. We followed the same path with an anti-unification algorithm, to which the major part of this chapter is dedicated.

3.1 Notions and terminology

We adjust the notion of proximity for terms to fit the block-based approach. The intuition behind it, according to [33], is based on the following idea: two terms t_1 and t_2 are λ-approximate when they have the same set of positions; their symbols, in their corresponding positions, belong to the same λ-block; and a certain symbol is always assigned to the same λ-block (throughout a computation). The following definition formalizes this intuition:

Definition 3.1.1. *Given a proximity relation $\mathcal{R}$ on $\mathcal{F}$ and $\lambda \in (0,1]$, two terms t and s are* λ-approximate *(or* λ-close*) with respect to $\mathcal{R}$, written $t \simeq_{\mathcal{R},\lambda} s$, if the following conditions hold:*

1. *$pos(t) = pos(s)$, i.e, the terms have exactly the same positions, hence, the same structure.*
2. *For all $p \in pos(t)$, $t[p]$ and $s[p]$ belong to the same λ-block of $\mathcal{R}$.*

3. For all positions $p, p' \in pos(t)$ with $p \neq p'$,

(a) If $t[p] = t[p']$, then $s[p]$ and $s[p']$ belong to the same λ-block of $\mathcal{R}$.

(b) If $s[p] = s[p']$, then $t[p]$ and $t[p']$ belong to the same λ-block of $\mathcal{R}$.

We consider undirected graphs $G = (V, E)$, where V is the set of vertices and E is the set of edges. A *clique* in a graph is a set of vertices $W \subseteq V$ such that for each pair of vertices in W there is an edge in E. A clique is *maximal* if it is not a proper subset of another clique. These notions are relevant here, because when λ is fixed, a proximity relation can be represented as an (unweighted) undirected graph and its maximal cliques (maximal complete subgraphs) are counterparts to blocks. In general, maximal cliques play an important role in the block-based approach and we need to define some notions related to them.

A *clique partition* of a graph G is a set of its cliques $\{C_1, \ldots, C_n\}$ such that $\bigcup_{i=1}^{n} C_i = V$ and $C_i \cap C_j = \emptyset$ for all $1 \leqslant i, j \leqslant n$, $i \neq j$.

Let $S_1 = \{C_1, \ldots, C_n\}$ and $S_2 = \{D_1, \ldots, D_m\}$ be two sets of cliques of the same graph. We say that S_1 is *subsumed* by S_2, written $S_1 \sqsubseteq S_2$, iff for all $1 \leqslant i \leqslant n$ there exists $1 \leqslant j \leqslant m$ such that $C_i \subseteq D_j$. If S_1 and S_2 are, in particular, partitions, then we also say that S_1 is a *subpartition* of S_2 if S_1 is subsumed by S_2.

A clique partition of a graph is *maximal* if it is not (properly) subsumed by another partition of the graph. A graph may have several maximal clique partitions.

In what follows, whenever appropriate we do not distinguish between $\mathcal{R}$ and the graph that represents it.

It is important to mention that the term and substitution comparison relation used in this chapter is $\lesssim_{\mathcal{R},\lambda}$. As we have seen, it is not a quasi-ordering since it is not transitive in general, but as noted in [33], under certain restrictions some kind of transitivity is preserved, which is exactly what we need in the block-based approach, where proximity between terms is defined as in Definition 3.1.1.

To make it more precise, note that in itself, defining $\lesssim_{\mathcal{R},\lambda}$ as in Definition 3.1.1 does not guarantee that it is transitive. For instance, if $\mathcal{R}_\lambda = \{(a, b), (b, c)\}$, then $a \lesssim_{\mathcal{R},\lambda} b$ (when the relation is partitioned as $\{\{a, b\}, \{c\}\}$), $b \lesssim_{\mathcal{R},\lambda} c$ (when the relation is partitioned as $\{\{a\}, \{b, c\}\}$), but not $a \lesssim_{\mathcal{R},\lambda} c$. However, if we have $t \lesssim_{\mathcal{R},\lambda} s$ and $s \lesssim_{\mathcal{R},\lambda} r$, both with respect to the same maximal clique partition of $\mathcal{R}$, then $t \lesssim_{\mathcal{R},\lambda} r$ in the same partition of $\mathcal{R}$.

3.2 Related work: block-based unification for proximity relations

We will dwell neither in the history of the fuzzy logic programming, nor in the full coverage of Bousi∼Prolog. It suffices to say that while similarity relations have been incorporated in the FLP for a while, Julián-Iranzo and his collaborators stumbled during their work upon problems in which the similarity-based algorithms did not offer a solution, even though intuitively the answer seemed straightforward.

For example, they considered a deductive database that stores information about people and their preferences on teaching, containing the following fragment:

```
%% PROXIMITY EQUATIONS
physics ~ math = 0.8.
physics ~ chemistry = 0.8.
chemistry ~ math = 0.6.

%% FACTS
likes_teaching(john, physics).
likes_teaching(mary, chemistry).
has_degree(john, physics).
has_degree(mary, chemistry).

%% RULES
can_teach(X, M) :-
has_degree(X, M),
likes_teaching(X, M).
```

Systems that disregard proximity relations would provide no answer for the query `"?-can_teach(X, math)"`. However, the intuition as well as the Bousi∼Prolog system answer `"X=john with 0.8"` and `"X=mary with 0.6"`.

Unlike Bousi∼Prolog and its proximity-based unification algorithm, the earlier fuzzy unification algorithms did not offer a solution to problems like this. These algorithms were all dealing with similarity exclusively, and their application to problems based on proximity relations would lead to incompleteness.

The block-based unification algorithm for proximity relations from [33] involves two procedures: the unification itself and the procedure for checking satisfiability of proximity constraints.[1] The latter is used as a condition to perform a step in the unification algorithm.

The unification rules work on configurations (in the paper they are called weak unification states), which are tuples of the form $P; \sigma; C; \alpha$ (we modify the original ones a bit to fit to the notation of this thesis), where P is the unification problems to be solved, σ is the substitution computed so far, C is the set of proximity constraints, and α is the unification degree computed so far. Proximity constraints are unordered pairs of function symbols of the form f—g, indicating that f and g are proximal and belong to the same proximity block.

The rules are as follows (the name abbreviations are combined with **U** for unification and with **JIRM** for the last names of the authors of [33], although we do not exactly follow their notation):

Tri-U-JIRM: Trivial

$\{x \simeq^?_{\mathcal{R},\lambda} x\} \uplus P; \sigma; C; \alpha \Longrightarrow P; \sigma; C; \alpha.$

Dec1-U-JIRM: Decomposition 1

$$\{f(t_1,\ldots,t_n) \simeq^?_{\mathcal{R},\lambda} f(s_1,\ldots,s_n)\} \uplus P; \sigma; C; \alpha \Longrightarrow$$
$$\{t_i \simeq^?_{\mathcal{R},\lambda} s_i \mid 1 \leqslant i \leqslant n\} \cup P; \sigma; C; \alpha.$$

Dec2-U-JIRM: Decomposition 2

$$\{f(t_1,\ldots,t_n) \simeq^?_{\mathcal{R},\lambda} g(s_1,\ldots,s_n)\} \uplus P; \sigma; C; \alpha \Longrightarrow$$
$$\{t_i \simeq^?_{\mathcal{R},\lambda} s_i \mid 1 \leqslant i \leqslant n\} \cup P; \sigma; \{f\text{—}g\} \cup C; \alpha \wedge \beta,$$

where $\mathcal{R}(f,g) = \beta \geqslant \lambda$ and $\{f\text{—}g\} \cup C$ is satisfiable.

Ori-U-JIRM: Orient

$\{t \simeq^?_{\mathcal{R},\lambda} x\} \uplus P; \sigma; C; \alpha \Longrightarrow \{x \simeq^?_{\mathcal{R},\lambda} t\} \cup P; \sigma; C; \alpha,$

where t is not a variable.

Elim-U-JIRM: Variable elimination

$\{x \simeq^?_{\mathcal{R},\lambda} t\} \uplus P; \sigma; C; \alpha \Longrightarrow P\{x \mapsto t\}; \sigma\{x \mapsto t\}; C; \alpha, \qquad \text{if } x \notin \mathcal{V}(t).$

[1]A more efficient block-based unification algorithm for proximity relations was described in [36].

Cla-U-JIRM: **Clash**
$\{f(t_1,\ldots,t_n) \simeq^?_{\mathcal{R},\lambda} g(s_1,\ldots,s_m)\} \uplus P; \sigma; C; \alpha \Longrightarrow \bot$,
if $\mathcal{R}(f,g) < \lambda$ or $\{f$—$g\} \cup C$ is unsatisfiable.

Occ-U-JIRM: **Occurrence check**
$\{x \simeq^?_{\mathcal{R},\lambda} t\} \uplus P; \sigma; C; \alpha \Longrightarrow \bot, \qquad$ if $x \in \mathcal{V}(t)$.

The block-based strategy is enforced by satisfiability of the proximity constraint C. Intuitively, C is satisfiable iff no symbol belongs to two different $(\mathcal{R},\lambda)$-blocks with respect to C. The corresponding algorithm, taken from [33], consists of three steps:

Input: a proximity constraint C, a proximity relation $\mathcal{R}$ and a cut value λ.

Output: sat (if C is satisfiable) or unsat (if C is unsatisfiable).

1. Compute the transitive closure C^* of C.
2. If there exists f—$g \in C^*$ such that $\mathcal{R}(f,g) < \lambda$, then **return** unsat.
3. **Return** sat.

To unify terms t and s with respect to a proximity relation $\mathcal{R}$ and a cut value λ in this approach, one starts with $\{t \simeq^?_{\mathcal{R},\lambda} s\}; Id; \emptyset; 1$ and applies the unification rules exhaustively. If the successful configuration $\emptyset; \sigma; C; \alpha$ is reached, σ is a most general $(\mathcal{R},\lambda)$-unifier of t and s with the approximation degree α. Otherwise, $\bot$ is returned indicating that t and s are not $(\mathcal{R},\lambda)$-unifiable in the block-based approach.

Example 3.2.1 (From [33]). Given a proximity relation defined as $\mathcal{R}(p,q) = 0.6$, $\mathcal{R}(b,d) = 0.3$, $\mathcal{R}(e,c) = 0.4$, $\mathcal{R}(r,s) = 0.5$, a cut value $\lambda = 0.2$, consider the set of unification problems $P = \{p(x,y,b) \simeq^?_{\mathcal{R},\lambda} q(a,b,d),\ r(z,e) \simeq^?_{\mathcal{R},\lambda} s(w,c)\}$, where the variables are x,y,z,w. In order to determine whether P is unifiable by proximity, the algorithm proceeds as follows:

$\{p(x,y,b) \simeq^?_{\mathcal{R},\lambda} q(a,b,d),\ r(z,e) \simeq^?_{\mathcal{R},\lambda} s(w,c)\}; Id; \emptyset; 1 \Longrightarrow_{\text{Dec2-U-JIRM}}$
$\{x \simeq^?_{\mathcal{R},\lambda} a,\ y \simeq^?_{\mathcal{R},\lambda} b,\ b \simeq^?_{\mathcal{R},\lambda} d,\ r(z,e) \simeq^?_{\mathcal{R},\lambda} s(w,c)\};$
$\quad Id; \{p$—$q\}; 0.6 \Longrightarrow_{\text{Elim-U-JIRM}}$
$\{y \simeq^?_{\mathcal{R},\lambda} b,\ b \simeq^?_{\mathcal{R},\lambda} d,\ r(z,e) \simeq^?_{\mathcal{R},\lambda} s(w,c)\};$
$\quad \{x \mapsto a\}; \{p$—$q\}; 0.6 \Longrightarrow_{\text{Elim-U-JIRM}}$

$$
\begin{aligned}
&\{b \simeq^?_{\mathcal{R},\lambda} d,\ r(z,e) \simeq^?_{\mathcal{R},\lambda} s(w,c)\}; \{x \mapsto a, y \mapsto b\}; \{p—q\}; 0.6 \Longrightarrow_{\text{Dec2-U-JIRM}}\\
&\{r(z,e) \simeq^?_{\mathcal{R},\lambda} s(w,c)\}; \{x \mapsto a, y \mapsto b\}; \{p—q, b—d\}; 0.3 \Longrightarrow_{\text{Dec2-U-JIRM}}\\
&\{z \simeq^?_{\mathcal{R},\lambda} w,\ e \simeq^?_{\mathcal{R},\lambda} c\}; \{x \mapsto a, y \mapsto b\}; \{p—q, b—d, r—s\}; 0.3 \Longrightarrow_{\text{Elim-U-JIRM}}\\
&\{e \simeq^?_{\mathcal{R},\lambda} c\}; \{x \mapsto a, y \mapsto b, z \mapsto w\}; \{p—q, b—d, r—s\}; 0.3 \Longrightarrow_{\text{Dec2-U-JIRM}}\\
&\emptyset; \{x \mapsto a, y \mapsto b, z \mapsto w\}; \{p—q, b—d, r—s, e—c\}; 0.3.
\end{aligned}
$$

It is easy to see that the constrains satisfiability check at each corresponding step in this derivation succeeds. The obtained substitution $\{x \mapsto a, y \mapsto b, z \mapsto w\}$ solves P with the approximation degree 0.3, because

$$
\begin{aligned}
&p(x,y,b)\{x \mapsto a, y \mapsto b, z \mapsto w\} = p(a,b,b).\\
&q(a,b,d)\{x \mapsto a, y \mapsto b, z \mapsto w\} = q(a,b,d).\\
&p(a,b,b) \simeq_{\mathcal{R},0.3} q(a,b,d) \text{ because } \mathcal{R}(p(a,b,b), q(a,b,d)) = 0.3.\\
&r(z,e)\{x \mapsto a, y \mapsto b, z \mapsto w\} = r(w,e).\\
&s(w,c)\{x \mapsto a, y \mapsto b, z \mapsto w\} = s(w,c).\\
&r(w,e) \simeq_{\mathcal{R},0.3} s(w,c) \text{ because } \mathcal{R}(r(w,e), s(w,c)) = 0.4 \geqslant 0.3.
\end{aligned}
$$

Moreover, the proximity constraint $\{p—q, b—d, r—s, e—c\}$ is satisfiable.

Example 3.2.2. In this example we show an unsuccessful unification attempt. Assume $\mathcal{R}(a,b) = 0.4$, $\mathcal{R}(b,c) = 0.5$, $\lambda = 0.3$ and consider the unification problem $P = \{f(x,x,x) \simeq^?_{\mathcal{R},\lambda} f(b,a,c)\}$. Then the unification algorithm proceeds as

$$
\begin{aligned}
&\{f(x,x,x) \simeq^?_{\mathcal{R},\lambda} f(b,a,c)\}; Id; \emptyset; 1 \Longrightarrow_{\text{Dec1-U-JIRM}}\\
&\{x \simeq^?_{\mathcal{R},\lambda} b,\ x \simeq^?_{\mathcal{R},\lambda} a,\ x \simeq^?_{\mathcal{R},\lambda} c\}; Id; \emptyset; 1 \Longrightarrow_{\text{Elim-U-JIRM}}\\
&\{b \simeq^?_{\mathcal{R},\lambda} a, b \simeq^?_{\mathcal{R},\lambda} c\}; \{x \mapsto b\}; \emptyset; 1 \Longrightarrow_{\text{Dec2-U-JIRM}}\\
&\{b \simeq^?_{\mathcal{R},\lambda} c\}; \{x \mapsto b\}; \{b—a\}; 0.4 \Longrightarrow_{\text{Cla-U-JIRM}}\\
&\perp.
\end{aligned}
$$

The rule **Cla-U-JIRM** applies, because the constraint $\{b—a, b—c\}$ is unsatisfiable: its transitive closure $\{b—a, b—c, a—c\}$ contains $a—c$, but $\mathcal{R}(a,c) = 0 < \lambda$.

3.3 Block-based anti-unification for proximity relations

In this section we present the development of an algorithm for anti-unification with the same restriction as that considered by [33] for the proximity relation, which is allowed to be partitioned block-wise.

Block-based anti-unification turns out to be more involved that its unification counterpart. Unlike unification, we do not have a single result here, as this example shows:

Example 3.3.1. If (a, b) and (a, c) both belong to $\mathcal{R}_\lambda$ but (b, c) does not, then $f(a, a)$ and $f(b, b)$ are $(\mathcal{R}, \lambda)$-close to each other, but $f(a, a)$ and $f(b, c)$ are not. The latter pair has two $(\mathcal{R}, \lambda)$-lggs: $f(a, x)$ and $f(x, a)$. They originate from two maximal clique partitions of the graph corresponding to $\mathcal{R}_\lambda$: the first partition being $\{\{a, b\},\ \{c\}\}$, and the second one $\{\{a, c\},\ \{b\}\}$.

The example shows that to compute the minimal complete set of $(\mathcal{R}, \lambda)$-generalizations of two terms, we need to be able to compute all maximal clique partitions of $\mathcal{R}$. The next two sections are dedicated to these problems: block-based approach to anti-unification with proximity relations and all maximal clique partitions of a graph, where we present our algorithms for solving them.

In the remaining of this chapter, we assume that the cut value λ is fixed and instead of $\mathcal{R}$, we work with the λ-cut $\mathcal{R}_\lambda$. That means that, essentially, we work with crisp tolerance relations. It simplifies the exposition, since the algorithms become less verbose. Adding the proximity degrees to them is not difficult and at the end we indicate how it can be done.

3.3.1 The anti-unification algorithm

Our anti-unification algorithm works on tuples $A; C; S; \mathcal{R}; G$, called configurations. Here A, C, and S are sets of anti-unification triples (AUTs), $\mathcal{R}$ is a crisp version of a proximity relation, and G is a term. The rules transform configurations into configurations. Intuitively, the problem set A contains AUTs that have not been solved yet, the set C contains AUTs of the form $x : a \triangleq b$, where a and b are constants such that $(a, b) \in \mathcal{R}$ and the AUTs are not solved yet. The store S contains the already solved AUTs, $\mathcal{R}$ is the proximity relation which gets more and more refined during computation by identifying symbols that belong to the same clique in some partition of $\mathcal{R}$, and G is the generalization which becomes more and more specific as the algorithm progresses by applying the following rules:

Dec: **Decomposition**

$$\begin{aligned}&\{x_1 : f_1(s^1_1,\dots,s^1_{k_1}) \triangleq g_1(t^1_1,\dots,t^1_{k_1}),\dots,\\ &\quad x_n : f_n(s^n_1,\dots,s^n_{k_n}) \triangleq g_n(t^n_1,\dots,t^n_{k_n})\};\ C;\ S;\ \mathcal{R};\ G \Longrightarrow\\ &\quad \{y^j_1 : s^j_1 \triangleq t^j_1,\dots,y^j_{k_j} : s^j_{k_j} \triangleq t^j_{k_j} \mid 1 \leqslant j \leqslant m\};\ C;\\ &\quad \{x_j : f_j(s^j_1,\dots,s^j_{k_j}) \triangleq g_j(t^j_1,\dots,t^j_{k_j}) \mid m+1 \leqslant j \leqslant n\} \cup S;\ \mathcal{R}';\ G\vartheta,\end{aligned}$$

where

(a) $k_i > 0$ and $(f_i, g_i) \in \mathcal{R}$ for all $1 \leqslant i \leqslant n$;

(b) there exist a maximal vertex-clique partition P of the subrelation $\mathcal{Q} = \{(f_1, g_1),\dots,(f_n, g_n)\} \subseteq \mathcal{R}$ and the index $1 \leqslant m \leqslant n$ such that for each (f_j, g_j), $1 \leqslant j \leqslant m$, there is a clique $Cl \in P$ with $f_j, g_j \in Cl$, and for no (f_j, g_j), $m+1 \leqslant j \leqslant n$ there is such a clique;

(c) $\mathcal{R}'$ is obtained from $\mathcal{R}$ by replacing the subrelation $\mathcal{Q}$ by its partition P;

(d) $\vartheta = \{x_j \mapsto f_j(y^j_1,\dots,y^j_{k_j}) \mid 1 \leqslant j \leqslant m\}$.

Sol: **Solve**

$$\begin{aligned}&\{x : f(s_1,\dots,s_k) \triangleq g(t_1,\dots,t_k)\} \uplus A;\ C;\ S;\ \mathcal{R};\ G \Longrightarrow\\ &\quad A;\ C; \{x : f(s_1,\dots,s_k) \triangleq g(t_1,\dots,t_k)\} \cup S;\ \mathcal{R};\ G,\end{aligned}$$

if $(f, g) \notin \mathcal{R}$.

Post: **Postpone**

$$\{x : a \triangleq b\} \uplus A;\ C;\ S;\ \mathcal{R};\ G \Longrightarrow A;\ \{x : a \triangleq b\} \cup C; S;\ \mathcal{R};\ G,$$

if $(a, b) \in \mathcal{R}$.

Gen-Con: **Generalize Constants**

$$\begin{aligned}&\varnothing; \{x_1 : a_1 \triangleq b_1,\dots,x_n : a_n \triangleq b_n\};\ S;\ \mathcal{R};\ G \Longrightarrow\\ &\quad \varnothing;\ \varnothing;\ \{x_j : a_j \triangleq b_j \mid m+1 \leqslant j \leqslant n\} \cup S;\ \mathcal{R}';\ G\vartheta,\end{aligned}$$

where

(a) $(a_i, b_i) \in \mathcal{R}$ for all $1 \leqslant i \leqslant n$;

(b) there exist a maximal vertex-clique partition P of the subrelation $\mathcal{Q} = \{(a_1, b_1), \dots, (a_n, b_n)\} \subseteq \mathcal{R}$ and the index $1 \leqslant m \leqslant n$ such that for each (a_j, b_j), $1 \leqslant j \leqslant m$ there is a clique $Cl \in P$ with $a_j, b_j \in Cl$, and for no (a_j, b_j), $m + 1 \leqslant j \leqslant n$ there is such a clique;

(c) $\mathcal{R}'$ is obtained from $\mathcal{R}$ by replacing $\mathcal{Q}$ by its partition P;

(d) $\vartheta = \{x_i \mapsto a_i \mid 1 \leqslant i \leqslant m\}$.

To anti-unify two terms s and t with respect to the proximity relation $\mathcal{R}$, we create the initial tuple $\{x : s \triangleq t\}; \emptyset; \emptyset; \mathcal{R}; x$ and apply the rules in all ways as long as possible. In the search space, branching is caused by all possible maximal clique partitions in Dec and Gen-Con. Generalizations in successful branches form the computed result. We call this algorithm $\mathfrak{A}_{\textit{block-lin}}$. The subscript lin indicates that it computes linear generalizations (i.e., those in which each generalization variable appears at most once).

Theorem 3.3.1. $\mathfrak{A}_{\text{block-lin}}$ *terminates and computes a minimal complete set of linear generalizations.*

Proof. We prove separately termination, soundness, completeness and minimality.

Termination. By analyzing the rules we notice that Sol and Post each eliminate an AUT from A, thus decreasing its size. Dec also strictly reduces the number of symbols, and consequently the size of A. Although it branches, the process is finite since the maximum number of possible branches equals the number of maximal vertex-clique partitions of a subgraph of $\mathcal{R}$, which is finite. The finitely many applications of these rules lead to A becoming empty.

When the problem set A is empty, Gen-Con is the only rule that can be applied. It strictly decreases the size of C. Its application will cause a branching, which similarly to what happens in the Dec rule, is finite. Eventually C becomes empty and afterwards no rule application is possible anymore.

Soundness. The algorithm starts with x as G, which is the most general generalization. Sol and Post do not change G. Dec and Gen-Con refine G, by replacing generalization variables with representatives of the cliques from a clique partition for the heads of the AUTs from A and C, respectively. The generalization variables are replaced only when the heads of the corresponding AUTs belong to the same clique, thus being proximal, and therefore, the obtained G is again a common generalization of the given terms. Simultaneously, $\mathcal{R}$ is adjusted by removing the connections of the involved symbols to all symbols that do not belong to the same clique. In this way the block restriction is kept for future application of the two rules.

Completeness. In computed answers, no generalization variable appears more than once, because there is no rule that would merge them. Hence, computed generalizations are linear. They are also lggs among linear generalizations, because (a) the algorithm decomposes the terms as much as possible, and (b) it maximizes the number of nonvariable subterms appearing in generalizations, which is done with the help of clique partitions of subrelations (not of the entire relation!) at each decomposition and constant generalization steps. All linear lggs are computed, because branching at **Dec** and **Gen-Con** rules explores all maximal clique partitions.

Minimality. **Dec** and **Gen-Con** generate branches, leading thus to alternative generalizations. If they are applied, generalizations computed on different branches belong to different cliques partitions, and therefore none of them can be more general than the other. □

Example 3.3.2. For terms $f(g_1(g_2(a)), g_2(a), a)$ and $f(g_2(g_3(b)), g_3(c), b)$ and the relation $\mathcal{R}$ given in the form of a maximal clique set (not a partition) $\{\{f\}, \{g_1, g_2\}, \{g_2, g_3\}, \{a, b\}, \{b, c\}\}$, the algorithm $\mathfrak{A}_{\textit{block-lin}}$ returns two $\mathcal{R}$-lggs: $f(g_1(z_1), y_2, a)$ and $f(y_1, g(y_2), a)$ and misses the nonlinear one $f(g_1(y_2), y_2, y_3)$. We illustrate now how the algorithm works:

$$\begin{aligned}
&\{x : f(g_1(g_2(a)), g_2(a), a) \triangleq f(g_2(g_3(b)), g_3(c), b)\};\ \emptyset;\ \emptyset;\ \mathcal{R};\ x \Longrightarrow_{\mathsf{Dec}}\\
&\{y_1 : g_1(g_2(a)) \triangleq g_2(g_3(b)),\ y_2 : g_2(a) \triangleq g_3(c),\ y_3 : a \triangleq b\};\\
&\qquad \emptyset;\ \emptyset;\ \mathcal{R};\ f(y_1, y_2, y_3) \Longrightarrow_{\mathsf{Post}}\\
&\{y_1 : g_1(g_2(a)) \triangleq g_2(g_3(b)),\ y_2 : g_2(a) \triangleq g_3(c)\};\ \{y_3 : a \triangleq b\};\\
&\qquad \emptyset;\ \mathcal{R};\ f(y_1, y_2, y_3).
\end{aligned}$$

At this stage, the subrelation $\{(g_1, g_2), (g_2, g_3)\}$ of $\mathcal{R}$ can be partitioned in two ways, which gives two new relations $\mathcal{R}_1 = \{\{f\}, \{g_1, g_2\}, \{g_3\}, \{a, b\}, \{b, c\}\}$ and $\mathcal{R}_2 = \{\{f\}, \{g_1\}, \{g_2, g_3\}, \{a, b\}, \{b, c\}\}$. Therefore, we can use the **Dec** rule and proceed in two different ways:

Alternative 1. Proceeding by $\mathcal{R}_1$.

$$\{y_1 : g_1(g_2(a)) \triangleq g_2(g_3(b)),\ y_2 : g_2(a) \triangleq g_3(c)\};\ \{y_3 : a \triangleq b\};$$

$$
\begin{aligned}
&\qquad \emptyset;\ \mathcal{R};\ f(y_1, y_2, y_3) \Longrightarrow_{\mathsf{Dec}}\\
&\{z_1 : g_2(a) \triangleq g_3(b)\};\ \{y_3 : a \triangleq b\};\ \{y_2 : g_2(a) \triangleq g_3(c)\};\\
&\qquad \mathcal{R}_1;\ f(g_1(z_1), y_2, y_3) \Longrightarrow_{\mathsf{Sol}}\\
&\emptyset;\ \{y_3 : a \triangleq b\};\ \{y_2 : g_2(a) \triangleq g_3(c), z_1 : g_2(a) \triangleq g_3(b)\};\\
&\qquad \mathcal{R}_1;\ f(g_1(z_1), y_2, y_3) \Longrightarrow_{\mathsf{Gen\text{-}Con}}\\
&\emptyset;\ \emptyset;\ \{y_2 : g_2(a) \triangleq g_3(c), z_1 : g_2(a) \triangleq g_3(b)\};\ \mathcal{R}_{11};\ f(g_1(z_1), y_2, a).
\end{aligned}
$$

where $\mathcal{R}_{11} = \{\{f\}, \{g_1, g_2\}, \{g_3\}, \{a, b\}, \{c\}\}$. Note that if we required in the condition of the **Gen-Con** rule to partition the relation itself (instead of its subrelation), we would get also $\mathcal{R}_{12} = \{\{f\}, \{g_1, g_2\}, \{g_3\}, \{a\}, \{b, c\}\}$, which would lead to another successful branch

$$
\begin{aligned}
&\emptyset;\ \{y_3 : a \triangleq b\};\ \{y_2 : g_2(a) \triangleq g_3(c), z_1 : g_2(a) \triangleq g_3(b)\};\\
&\qquad \mathcal{R}_1;\ f(g_1(z_1), y_2, y_3) \Longrightarrow_{\mathsf{Gen\text{-}Con}}\\
&\emptyset;\ \emptyset;\ \{y_2 : g_2(a) \triangleq g_3(c), z_1 : g_2(a) \triangleq g_3(b), y_3 : a \triangleq b\};\\
&\qquad \mathcal{R}_{12};\ f(g_1(z_1), y_2, y_3).
\end{aligned}
$$

However, the computed generalization is not an lgg, since it is more general than the previous one.

Alternative 2. Proceeding by $\mathcal{R}_2$.

$$
\begin{aligned}
&\{y_1 : g_1(g_2(a)) \triangleq g_2(g_3(b)),\ y_2 : g_2(a) \triangleq g_3(c)\};\ \{y_3 : a \triangleq b\};\\
&\qquad \emptyset;\ \mathcal{R};\ f(y_1, y_2, y_3) \Longrightarrow_{\mathsf{Dec}}\\
&\{z_2 : a \triangleq c\};\ \{y_3 : a \triangleq b\};\ \{y_1 : g_1(g_2(a)) \triangleq g_2(g_3(b))\};\\
&\qquad \mathcal{R}_2;\ f(y_1, g_2(z_2), y_3) \Longrightarrow_{\mathsf{Sol}}\\
&\emptyset;\ \{y_3 : a \triangleq b\};\ \{y_1 : g_1(g_2(a)) \triangleq g_2(g_3(b)), z_2 : a \triangleq c\};\\
&\qquad \mathcal{R}_2;\ f(y_1, g_2(z_2), y_3) \Longrightarrow_{\mathsf{Gen\text{-}Con}}\\
&\emptyset;\ \emptyset;\ \{y_1 : g_1(g_2(a)) \triangleq g_2(g_3(b)), z_2 : a \triangleq c\};\ \mathcal{R}_{21};\ f(y_1, g_2(z_2), a),
\end{aligned}
$$

where $\mathcal{R}_{21} = \{\{f\}, \{g_1\}, \{g_2, g_3\}, \{a, b\}, \{c\}\}$. Again, if we were allowed to partition the whole $\mathcal{R}_2$ in **Gen-Con**, we would get another partition $\mathcal{R}_{22} = \{\{f\}, \{g_1\}, \{g_2, g_3\}, \{a\}, \{b, c\}\}$, which would give the following successful branch:

$$
\begin{aligned}
&\emptyset;\ \{y_3 : a \triangleq b\};\ \{y_1 : g_1(g_2(a)) \triangleq g_2(g_3(b)), z_2 : a \triangleq c\};\\
&\qquad \mathcal{R}_2;\ f(y_1, g(y_z), y_3) \Longrightarrow_{\mathsf{Gen\text{-}Con}}
\end{aligned}
$$

$$\begin{aligned}&\emptyset;\ \emptyset;\ \{y_1 : g_1(g_2(a)) \triangleq g_2(g_3(b)), z_2 : a \triangleq c, y_3 : a \triangleq b\};\\ &\qquad \mathcal{R}_{22};\ f(y_1, g(z_2), y_3).\end{aligned}$$

However, the solution obtained in this branch is more general than the previous one.

As the next step, we extend the algorithm to add a rule for merging variables. It uses a partial function $\mathit{refine}(\{s_1 \approx t_1, \dots, s_n \approx t_n\}, \mathcal{R})$, which is supposed to refine the given relation $\mathcal{R}$ into a new relation $\mathcal{R}'$ so that $s_i \approx_{\mathcal{R}'} t_i$, $1 \leqslant i \leqslant n$, if such a refinement of $\mathcal{R}$ exists. The function *refine* is defined as follows:

$$\begin{aligned}&\mathit{refine}(\emptyset, \mathcal{R}) = \mathcal{R}.\\ &\mathit{refine}(\{t \approx t\} \uplus E, \mathcal{R}) = \mathit{refine}(E, \mathcal{R}).\\ &\mathit{refine}(\{f(s_1, \dots, s_n) \approx g(t_1, \dots, t_n)\} \uplus E, \mathcal{R}) =\\ &\qquad \mathit{refine}(\{s_1 \approx t_1, \dots, s_n \approx t_n\} \cup E, \mathcal{R}'),\end{aligned}$$

if $(f, g) \in \mathcal{R}$ and $\mathcal{R}' = \mathcal{R} \backslash (S_1 \cup S_2)$, where $S_1 = \{(f, h) \mid (g, h) \notin \mathcal{R}\} \cup \{(h, f) \mid (g, h) \notin \mathcal{R}\}$ and $S_2 = \{(g, h) \mid (f, h) \notin \mathcal{R}\} \cup \{(h, g) \mid (f, h) \notin \mathcal{R}\}$. Otherwise, *refine* is not defined.

Mer: Merge

$$\begin{aligned}&A;\ C;\ \{x : s_1 \triangleq t_1,\ y : s_2 \triangleq t_2\} \uplus S;\ \mathcal{R};\ G \Longrightarrow\\ &\qquad A;\ C;\ \{x : s_1 \triangleq t_1\} \cup S;\ \mathcal{R}';\ G\{y \mapsto x\},\end{aligned}$$

where $\mathcal{R}' = \mathit{refine}(\{s_1 \approx s_2, t_1 \approx t_2\}, \mathcal{R})$.

The obtained algorithm is denoted by $\mathfrak{A}_{\mathit{block}}$.

Mer is an alternative to the rules from $\mathfrak{A}_{\mathit{block\text{-}lin}}$, meaning that it would introduce additional branching, and $\mathfrak{A}_{\mathit{block}}$ might also recompute the same solution on different branches.

Example 3.3.3. Let $\mathcal{R} = \{\{f\}, \{g_1, g_2\}, \{h\}, \{a, b\}\}$, $s = f(a, g_1(a), g_2(a))$, and $t = f(b, h(a), h(a))$. Then we have two branches that compute the same result. The initial part of them can look, e.g., like this:

$$\begin{aligned}&\{x : f(a, g_1(a), g_2(a)) \triangleq f(b, h(a), h(a))\};\ \emptyset;\ \emptyset;\ \mathcal{R};\ x \Longrightarrow_{\mathsf{Dec}}\\ &\{y_1 : a \triangleq b, y_2 : g_1(a) \triangleq h(a), y_3 : g_2(a) \triangleq h(a)\}; \emptyset;\ \ \emptyset;\\ &\qquad \mathcal{R};\ f(y_1, y_2, y_3) \Longrightarrow^2_{\mathsf{Sol}}\end{aligned}$$

$$\begin{aligned}&\{y_1 : a \triangleq b\};\ \emptyset;\ \ \{y_2 : g_1(a) \triangleq h(a), y_3 : g_2(a) \triangleq h(a)\};\\ &\qquad \mathcal{R};\ f(y_1, y_2, y_3) \Longrightarrow_{\mathsf{Post}}\\ &\emptyset;\ \{y_1 : a \triangleq b\};\ \ \{y_2 : g_1(a) \triangleq h(a), y_3 : g_2(a) \triangleq h(a)\};\ \mathcal{R};\ f(y_1, y_2, y_3).\end{aligned}$$

From this moment on, either we first do **Gen-Con** and then **Mer**, or first **Mer** and then **Gen-Con**. In both cases we compute the same generalization $f(a, y_2, y_2)$.

However, we cannot postpone **Mer** till the end, after A and C get empty (as it is usually done in anti-unification algorithms), because in this case we will miss solutions, as the example below shows.

Example 3.3.4. Let $\mathcal{R} = \{\{f\}, \{h_1\}, \{h_2\}, \{a\}, \{g_0, g_1\}, \{g_0, g_2\}\}$, $s = f(g_0(a), h_1(g_0(a)), h_1(g_0(a)))$, and $t = f(g_1(a), h_2(g_2(a)), h_2(g_0(a)))$. The first steps of $\mathfrak{A}_{block}$ could be as it follows:

$$\begin{aligned}&\{x : f(g_0(a), h_1(g_0(a)), h_1(g_0(a))) \triangleq f(g_1(a), h_2(g_2(a)), h_2(g_0(a)))\};\\ &\qquad \emptyset;\ \emptyset;\ \mathcal{R};\ x \Longrightarrow_{\mathsf{Dec}}\\ &\{y_1 : g_0(a) \triangleq g_1(a), y_2 : h_1(g_0(a)) \triangleq h_2(g_2(a)), y_3 : h_1(g_0(a)) \triangleq h_2(g_0(a))\};\\ &\qquad \emptyset;\ \emptyset;\ \mathcal{R};\ f(y_1, y_2, y_3) \Longrightarrow^2_{\mathsf{Sol}}\\ &\{y_1 : g_0(a) \triangleq g_1(a)\};\ \emptyset;\\ &\qquad \{y_2 : h_1(g_0(a)) \triangleq h_2(g_2(a)), y_3 : h_1(g_0(a)) \triangleq h_2(g_0(a))\};\ \mathcal{R};\ f(y_1, y_2, y_3)\end{aligned}$$

From here $\mathfrak{A}_{block}$ branches and computes two solutions: $f(g_0(a), y_2, y_3)$ and $f(y_1, y_2, y_2)$. The first one is obtained by applying **Dec** before **Mer**, and the second one in the other way around. However, if **Mer** is applied only at the very end, then the second solution is not computed.

Merging variables can significantly increase the size of the computed set of generalizations:

Example 3.3.5. Let the arity of f be $n+1$, $\mathcal{R} = \{\{f\}, \{h_1\}, \{h_2\}, \{a\}, \{g_0, g_1\}, \dots, \{g_0, g_n\}\}$ and

$$\begin{aligned}s &= f(g_0(a),\ h_1(g_0(a)), \dots,\ h_1(g_0(a)),\ h_1(g_0(a))),\\ t &= f(g_1(a),\ h_2(g_2(a)), \dots,\ h_2(g_n(a)),\ h_2(g_0(a))).\end{aligned}$$

$\mathfrak{A}_{block\text{-}lin}$ computes only one generalization: $f(g_0(a), y_2, \dots, y_n, y_{n+1})$. With $\mathfrak{A}_{block}$, we have, in addition, $n-1$ other generalizations: $f(y_1, y_2, \dots, y_n, y_2)$, $\dots, f(y_1, y_2, \dots, y_n, y_n)$.

Theorem 3.3.2. $\mathfrak{A}_{\text{block}}$ *computes a minimal complete set of generalizations.*

Proof. The properties of the $\mathfrak{A}_{\textit{block-lin}}$ have already been proven in Theorem 3.3.1. Here we focus on the **Mer** rule.

Termination. Since the store S is finite, and **Mer** reduces its size, it is obvious that $\mathfrak{A}_{block}$ terminates.

Soundness. The function *refine* is defined only for pairs of terms that are proximal to each other. When applied in **Mer**, the symbols at the same positions of the proximal terms are choosing the same clique, and all their connections to symbols that would break the clique are removed. Consequently, the refined relation considered in the next **Mer** applications will correctly stay inside the block restriction. From the proximity of the considered candidates and the compliance to the above restriction it follows that **Mer** is sound, and thus, also $\mathfrak{A}_{block}$ is sound.

Completeness. **Mer** considers all possible combinations between AUTs from the store, and therefore it cannot lose non-linear solutions.

Minimality. **Mer** generates branches, leading thus to alternative generalizations. Even though **Mer** does not explicitly compute a maximal cliques partition, the *refine* function, through the removal of some edges has the same effect. Therefore, generalizations computed on different branches belong to different partitions, and one cannot be more general than the other. □

When dealing with proximity relations, the computation of the approximation degrees is an important matter. Even though the above algorithm does not consider them, once a generalization is computed, it is a straightforward task to compute the approximation degrees of this generalization with respect to the original terms. One starts by substituting the variables in the computed generalization with the corresponding side of the variables AUTs from the store S. Note that all the symbols appearing in the generalization term on which substitution was applied are cliques representatives and belong to the same cliques as the symbols at the same positions in the original terms. One needs to take each symbol from the original term and choose the closest symbol (with its proximity degree) from the corresponding clique. The minimum of all the proximity degrees will be the computed approximation degree for the original term.

3.3.2 Computing all maximal clique partitions

The anti-unification algorithm in the previous section relies on the computation of all maximal clique partitions in an undirected graph. A graph may have several maximal clique partitions. In the literature, a problem that was studied intensively is to compute a maximal clique partition with the smallest number of cliques. Tseng's algorithm [75], introduced to solve this problem, was motivated by its application in the design of processors. Later, Bhasker and Samad [12] proposed two other algorithms. They also derived the upper bound on the number of cliques in a partition and showed that there exists a partition containing a maximal clique of the graph.

A problem closely related to clique partition is the vertex coloring problem [31], which requires to color the vertices of a graph in such a way that two adjacent vertices have different colors. In fact, a clique-partitioning problem of a graph is equivalent to the coloring problem of its complement graph. Both problems are NP-complete [38].

The problem of computing all maximal cliques is also well-studied, see, e.g. [13, 74, 73, 14]. However, the problem we encountered in the previous section, computing all maximal clique partitions, was not studied before, to the best of our knowledge. This is what we address in this section.

For a graph G, we denote by *all-max-cliques*(G) the set of all its maximal cliques. We start with computing this set (e.g., by Bron-Kerbosch algorithm [13]) and give each of its elements a name. For the graph in Fig. 3.1, there are four of them: $C_1 = \{1,2,3\}, C_2 = \{2,3,4\}, C_3 = \{4,5,6\}, C_4 = \{5,6,7\}$. These cliques will get revised during computation by removing elements from them. At the end, we report those which are not empty.

After computing the initial cliques, we collect all shared vertices and indicate among which cliques they are shared. In the graph in Fig. 3.1, the shared vertices are 2, 3, 4, 5, and 6. We have $2 \in C_1 \cap C_2$, $3 \in C_1 \cap C_2$, $4 \in C_2 \cap C_3$, $5 \in C_3 \cap C_4$, and $6 \in C_3 \cap C_4$.

Our goal is to compute each solution exactly once. At the end, it can happen that some cliques consist of shared vertices only. Such cliques can have any of the names of the original cliques they stem from. For instance, the node 4 alone can form a clique either as C_2 or C_3, giving two identical partitions which differ only by the clique names:

$$C_1 = \{1,2,3\}, \quad C_2 = \{4\}, \quad C_3 = \varnothing, \quad C_4 = \{5,6,7\},$$
$$C_1 = \{1,2,3\}, \quad C_2 = \varnothing, \quad C_3 = \{4\}, \quad C_4 = \{5,6,7\}.$$

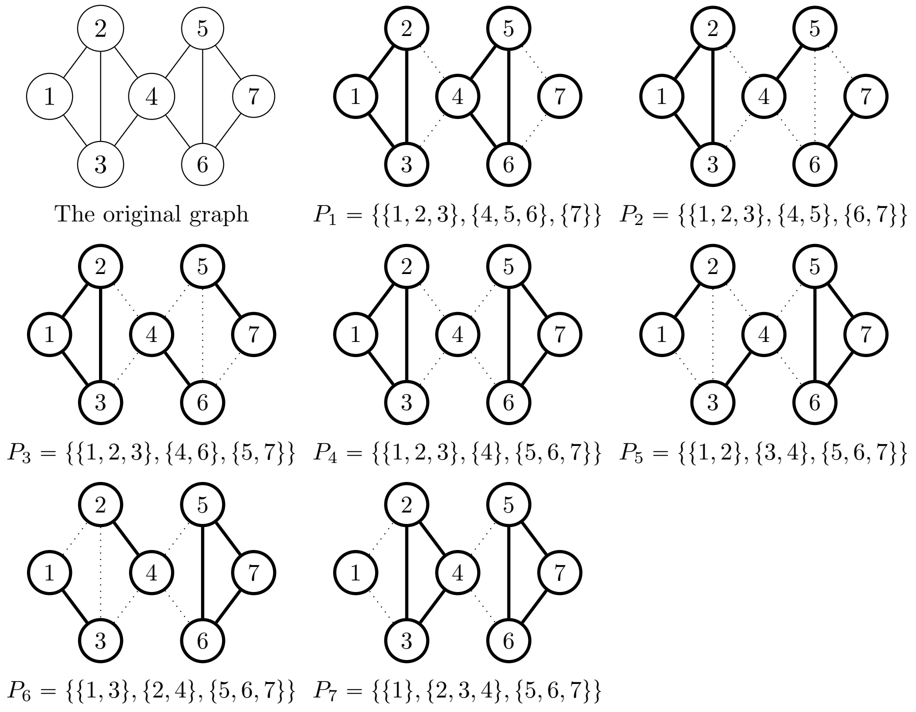

The original graph
$P_1 = \{\{1,2,3\},\{4,5,6\},\{7\}\}$
$P_2 = \{\{1,2,3\},\{4,5\},\{6,7\}\}$
$P_3 = \{\{1,2,3\},\{4,6\},\{5,7\}\}$
$P_4 = \{\{1,2,3\},\{4\},\{5,6,7\}\}$
$P_5 = \{\{1,2\},\{3,4\},\{5,6,7\}\}$
$P_6 = \{\{1,3\},\{2,4\},\{5,6,7\}\}$
$P_7 = \{\{1\},\{2,3,4\},\{5,6,7\}\}$

Figure 3.1: All maximal clique partitions of a graph.

We want to avoid such duplicates. Therefore, for such alternatives we choose one single clique to which a shared vertex can belong in this configuration, and forbid the others. For the example graph in Fig. 3.1, we can allow the vertices 2 and/or 3 to form a clique as C_2, the vertex 4 to form a clique as C_3, and the vertices 5 and/or 6 to form a clique as C_4. (Note that allowing does not necessarily mean that we will get result cliques of that form. For instance, in C_4 we will have 7 as well.) Thus, the candidates for forbidden configurations are $C_1 \neq \{2\}, C_1 \neq \{3\}, C_1 \neq \{2,3\}, C_2 \neq \{4\}, C_3 \neq \{5\}, C_3 \neq \{6\}, C_3 \neq \{5,6\}$. This can be further simplified by observing that C_1 contains a non-shared vertex 1 and, hence, cannot consist of shared vertices only. Therefore, we can omit the first three candidate disequations and obtain the forbidden configuration $C_2 \neq \{4\}, C_3 \neq \{5\}, C_3 \neq \{6\}, C_3 \neq \{5,6\}$.

Starting from the initial set of cliques, our algorithm **All-Maximal-Clique-Partitions** performs the following steps:

1. Compute the set of shared vertices and the forbidden configurations.

2. If the set of shared vertices is empty, return the current set of cliques and stop.

3. Select a shared vertex and nondeterministically assign it to one of the cliques it belongs to. Remove the vertex from the other cliques and from the set of shared vertices.

4. For each pair of cliques C_i, C_j, where $C_i \subseteq C_j$, make C_i empty and adjust the set of shared elements. In addition, if C_i was the chosen clique for the shared elements, remove those elements from the forbidden list of C_j.

5. If the union of two nonempty cliques is a subset of an original clique, or if a forbidden configuration arises, stop the development of this branch with failure. Otherwise go to step 2.

Checking for the subset relations is needed to avoid computing cliques which are not maximal. For instance, the partition $C_1 = \{1, 2\}$, $C_2 = \{3\}$, $C_3 = \{4\}$, $C_4 = \{5, 6, 7\}$ should be rejected because $\{1, 2\} \cup \{3\}$ is a subset of the original C_1 clique. Step 5 helps to detect such situations early.

The partitions shown in Fig. 3.1 correspond to the following final values of cliques, computed by the **All-Maximal-Clique-Partitions** algorithm:

$$\begin{array}{lllll}
P_1: & C_1 = \{1,2,3\}, & C_2 = \varnothing, & C_3 = \{4,5,6\}, & C_4 = \{7\} \\
P_2: & C_1 = \{1,2,3\}, & C_2 = \varnothing, & C_3 = \{4,5\}, & C_4 = \{6,7\} \\
P_3: & C_1 = \{1,2,3\}, & C_2 = \varnothing, & C_3 = \{4,6\}, & C_4 = \{5,7\} \\
P_4: & C_1 = \{1,2,3\}, & C_2 = \varnothing, & C_3 = \{4\}, & C_4 = \{5,6,7\} \\
P_5: & C_1 = \{1,2\}, & C_2 = \{3,4\}, & C_3 = \varnothing, & C_4 = \{5,6,7\} \\
P_6: & C_1 = \{1,3\}, & C_2 = \{2,4\}, & C_3 = \varnothing, & C_4 = \{5,6,7\} \\
P_7: & C_1 = \{1\}, & C_2 = \{2,3,4\}, & C_3 = \varnothing, & C_4 = \{5,6,7\}.
\end{array}$$

Before proving the properties of the algorithm, we illustrate it with some examples.

Example 3.3.6. Let G be the graph shown in Fig. 3.1. We start with the set of all maximal cliques in it: $C_1 := \{1, 2, 3\}$, $C_2 := \{2, 3, 4\}$, $C_3 := \{4, 5, 6\}$, $C_4 := \{5, 6, 7\}$. The set of all shared vertices, represented as elements of intersections, is *shared* $:= \{2 \in C_1 \cap C_2,\ 3 \in C_1 \cap C_2,\ 4 \in C_2 \cap C_3,\ 5 \in C_3 \cap C_4,\ 6 \in C_3 \cap C_4\}$, and the set of all forbidden configurations is *fc* $:= \{C_2 \neq$

$\{4\}, C_3 \neq \{5\}, C_3 \neq \{6\}, C_3 \neq \{5,6\}\}$. The steps of the algorithm below are shown as the nodes in the complete search tree, and they are enumerated as the positions of those nodes in the search tree.

1. Step 3: $2 \in C_1$, $C_2 := \{3,4\}$, $\mathit{shared} := \mathit{shared} \backslash \{2 \in C_1 \cap C_2\}$.
Step 4 and Step 5 do not apply. Go to Step 2. It does not apply.

1.1 Step 3: $3 \in C_1$, $C_2 := \{4\}$, $\mathit{shared} := \mathit{shared} \backslash \{3 \in C_1 \cap C_2\}$.
Step 4: $C_2 \subseteq C_3$, $C_2 := \varnothing$, $\mathit{shared} := \mathit{shared} \backslash \{4 \in C_2 \cap C_3\}$.
Step 5 does not apply. Go to Step 2. It does not apply.

1.1.1 Step 3: $5 \in C_3$, $C_4 := \{6,7\}$, $\mathit{shared} := \mathit{shared} \backslash \{5 \in C_3 \cap C_4\}$.
Step 4 and Step 5 do not apply. Go to Step 2. It does not apply.

1.1.1.1 Step 3: $6 \in C_3$, $C_4 := \{7\}$, $\mathit{shared} := \mathit{shared} \backslash \{6 \in C_3 \cap C_4\}$.
Step 4 and Step 5 do not apply. Go to Step 2.
Step 2: $\mathit{shared} = \varnothing$,
Return $C_1 = \{1,2,3\}$, $C_2 = \varnothing$, $C_3 = \{4,5,6\}$, $C_4 = \{7\}$.

1.1.1.2 Step 3: $6 \in C_4$, $C_3 := \{4,5\}$, $\mathit{shared} := \mathit{shared} \backslash \{6 \in C_3 \cap C_4\}$.
Step 4 and Step 5 do not apply. Go to Step 2.
Step 2: $\mathit{shared} = \varnothing$,
Return $C_1 = \{1,2,3\}$, $C_2 = \varnothing$, $C_3 = \{4,5\}$, $C_4 = \{6,7\}$.

1.1.2 Step 3: $5 \in C_4$, $C_3 := \{4,6\}$, $\mathit{shared} := \mathit{shared} \backslash \{5 \in C_3 \cap C_4\}$.
Step 4 and Step 5 do not apply. Go to Step 2. It does not apply.

1.1.2.1 Step 3: $6 \in C_3$, $C_4 := \{5,7\}$, $\mathit{shared} := \mathit{shared} \backslash \{6 \in C_3 \cap C_4\}$.
Step 4 and Step 5 do not apply. Go to Step 2.
Step 2: $\mathit{shared} = \varnothing$,
Return $C_1 = \{1,2,3\}$, $C_2 = \varnothing$, $C_3 = \{4,6\}$, $C_4 = \{5,7\}$.

1.1.2.2 Step 3: $6 \in C_4$, $C_3 := \{4\}$, $\mathit{shared} := \mathit{shared} \backslash \{6 \in C_3 \cap C_4\}$.
Step 4 and Step 5 do not apply. Go to Step 2.
Step 2: $\mathit{shared} = \varnothing$,
Return $C_1 = \{1,2,3\}$, $C_2 = \varnothing$, $C_3 = \{4\}$, $C_4 = \{5,6,7\}$.

1.2 Step 3: $3 \in C_2$, $C_1 := \{1,2\}$, $\mathit{shared} := \mathit{shared} \backslash \{3 \in C_1 \cap C_2\}$.
Step 4 and Step 5 do not apply. Go to Step 2. It does not apply.

1.2.1 Step 3: $4 \in C_2$, $C_3 := \{5,6\}$, $\mathit{shared} := \mathit{shared} \backslash \{4 \in C_2 \cap C_3\}$.

Step 4: $C_3 \subseteq C_4,\ C_3 := \emptyset$,

$shared := shared \backslash \{5 \in C_3 \cup C_4,\ 6 \in C_3 \cup C_4\}$

Step 5 does not apply. Go to Step 2.

Step 2: $shared = \emptyset$,

Return $C_1 = \{1, 2\}$, $C_2 = \{3, 4\}$, $C_3 = \emptyset$, $C_4 = \{5, 6, 7\}$.

1.2.2 Step 3: $4 \in C_3,\ C_2 := \{3\},\ shared := shared \backslash \{4 \in C_2 \cap C_3\}$.

Step 4 does not apply.

Step 5: $C_1 \cup C_2 = \{1, 2, 3\}$ which is one of the original cliques.

Fail.

2. Step 3: $2 \in C_2,\ C_1 := \{1, 3\},\ shared := shared \backslash \{2 \in C_1 \cap C_2\}$.

Step 4 and Step 5 do not apply. Go to Step 2. It does not apply.

2.1 Step 3: $3 \in C_1,\ C_2 := \{2, 4\},\ shared := shared \backslash \{3 \in C_1 \cap C_2\}$.

Step 4 and Step 5 do not apply. Go to Step 2. It does not apply.

2.1.1 Step 3: $4 \in C_2,\ C_3 := \{5, 6\},\ shared := shared \backslash \{4 \in C_2 \cap C_3\}$.

Step 4: $C_3 \subseteq C_4,\ C_3 := \emptyset$,

$shared := shared \backslash \{5 \in C_3 \cup C_4,\ 6 \in C_3 \cup C_4\}$

Step 5 does not apply. Go to Step 2.

Step 2: $shared = \emptyset$,

Return $C_1 = \{1, 3\}$, $C_2 = \{2, 4\}$, $C_3 = \emptyset$, $C_4 = \{5, 6, 7\}$.

2.1.2 Step 3: $4 \in C_3,\ C_2 := \{2\},\ shared := shared \backslash \{4 \in C_2 \cap C_3\}$.

Step 4 does not apply.

Step 5: $C_1 \cup C_2 = \{1, 2, 3\}$ which is one of the original cliques.

Fail.

2.2 Step 3: $3 \in C_2,\ C_1 := \{1\},\ shared := shared \backslash \{3 \in C_1 \cap C_2\}$.

Step 4 and Step 5 do not apply. Go to Step 2. It does not apply.

2.2.1 Step 3: $4 \in C_2,\ C_3 := \{5, 6\},\ shared := shared \backslash \{4 \in C_2 \cap C_3\}$.

Step 4: $C_3 \subseteq C_4,\ C_3 := \emptyset$,

$shared := shared \backslash \{5 \in C_3 \cup C_4, 6 \in C_3 \cup C_4\}$

Step 5 does not apply. Go to Step 2.

Step 2: $shared = \emptyset$,

Return $C_1 = \{1\}$, $C_2 = \{2, 3, 4\}$, $C_3 = \emptyset$, $C_4 = \{5, 6, 7\}$.

2.2.2 Step 3: $4 \in C_3$, $C_2 := \{2,3\}$, $\mathit{shared} := \mathit{shared} \backslash \{4 \in C_2 \cap C_3\}$.
Step 4 does not apply.
Step 5: $C_1 \cup C_2 = \{1,2,3\}$ which is one of the original cliques.
Fail.

The returned partitions are exactly those shown in Fig. 3.1.

Example 3.3.7. Let G be the graph given by its all maximal cliques: $C_1 := \{1,2\}$, $C_2 := \{1,3\}$, $C_3 := \{3,4\}$, $C_4 := \{2,5\}$. The set of all shared vertices, represented as elements of intersections, is *shared* $:= \{1 \in C_1 \cap C_2$, $2 \in C_1 \cap C_4$, $3 \in C_2 \cap C_3\}$, and the set of all forbidden configurations is $\mathit{fc} := \{C_1 \neq \{1\}, C_1 \neq \{2\}, C_2 \neq \{3\}\}$. The steps of the algorithm below are shown as the nodes in the complete search tree, and they are enumerated as the positions of those nodes in the search tree.

1. Step 3: $1 \in C_1$, $C_2 := \{3\}$, $\mathit{shared} := \mathit{shared} \backslash \{1 \in C_1 \cap C_2\}$.
Step 4: $C_2 \subseteq C_3$, $C_2 := \varnothing$, $\mathit{shared} := \mathit{shared} \backslash \{3 \in C_2 \cap C_3\}$.
Step 5 does not apply. Go to Step 2. It does not apply.

1.1 Step 3: $2 \in C_1$, $C_4 := \{5\}$, $\mathit{shared} := \mathit{shared} \backslash \{2 \in C_1 \cap C_4\}$.
Step 4 and Step 5 do not apply. Go to Step 2.
Step 2: $\mathit{shared} = \varnothing$,
Return $C_1 = \{1,2\}$, $C_2 = \varnothing$, $C_3 = \{3,4\}$, $C_4 = \{5\}$.

1.2 Step 3: $2 \in C_4$, $C_1 := \{1\}$, $\mathit{shared} := \mathit{shared} \backslash \{2 \in C_1 \cap C_4\}$.
Step 4 do not apply.
Step 5: C_1 is forbidden by *fc*.
Fail.

2. Step 3: $1 \in C_2$, $C_1 := \{2\}$, $\mathit{shared} := \mathit{shared} \backslash \{1 \in C_1 \cap C_2\}$.
Step 4: $C_1 \subseteq C_4$, $C_1 := \varnothing$, $\mathit{shared} = \mathit{shared} \backslash \{2 \in C_1 \cap C_4\}$.
Step 5 does not apply. Go to Step 2. It does not apply.

2.1 Step 3: $3 \in C_2$, $C_3 := \{4\}$, $\mathit{shared} := \mathit{shared} \backslash \{3 \in C_2 \cap C_3\}$.
Step 4 and Step 5 do not apply. Go to Step 2.
Step 2: $\mathit{shared} = \varnothing$,
Return $C_1 = \varnothing$, $C_2 = \{1,3\}$, $C_3 = \{4\}$, $C_4 = \{2,5\}$.

2.2 Step 3: $3 \in C_3$, $C_2 := \{1\}$, $\mathit{shared} := \mathit{shared} \backslash \{3 \in C_2 \cap C_3\}$.

Step 4 and Step 5 do not apply. Go to Step 2.

Step 2: *shared* $= \varnothing$,

Return $C_1 = \varnothing$, $C_2 = \{1\}$, $C_3 = \{3, 4\}$, $C_4 = \{2, 5\}$.

Hence, the algorithm computes three maximal clique partitions. One can see how the forbidden configuration prevented to compute the partition $C_1 = \{1\}$, $C_2 = \varnothing$, $C_3 = \{3, 4\}$, $C_4 = \{2, 5\}$ in the node **1.2**, which would be a duplicate of the partition computed in the node **2.2**.

Looking at the algorithm, one can easily notice that if we did not have the steps 4 and 5, we would still compute all maximal clique partitions (since at Step 3 we assign shared vertices to one of the cliques they belong to). However, in addition, subpartitions of these partitions might also be generated. It might also happen that the same maximal partition is computed more than once. Therefore, we need to show that in steps 4 and 5 we eliminate exactly those subpartitions and duplicates.

Strictly speaking, one can omit Step 4 completely. In the subsequent splitting of shared vertices between C_i and C_j, if a shared vertex goes from C_j to C_i (or if forbidden configuration involving C_i arises), Step 5 will block the development of the branch, effectively imitating the behavior of Step 4, but doing it in several steps. Step 4 is there to reduce this extra work.

Theorem 3.3.3. *Each set of cliques computed by the* All-Maximal-Clique-Partitions *for the given graph is a maximal clique partition for the graph.*

Proof. Let $S = \{C_1, \dots, C_n\}$ be a set of cliques of the given graph G the algorithm returns.

First, show that S is a clique partition of G. The algorithm works by removing vertices from a precomputed set of all maximal cliques of G. Hence, every element of each C_i is a vertex of G. During computation, no vertex gets lost: For Step 4 it is obvious, and for Step 5 it follows from the fact that if two cliques are included in a bigger original clique, then that bigger clique will appear on a neighboring branch and its elements will not get lost. For forbidden configurations it is also easy to see, because those vertices that are forbidden in one clique, are allowed in another. At the end of the computation, they either remain where they are allowed for a clique, or appear in the forbidden ones together with some other vertices. In any case, they are not lost.

Hence, no vertex of G is missing from S. Since the algorithm stops when there are no shared vertices, we get that $C_i \cap C_j = \varnothing$ for all $1 \leqslant i, j \leqslant n$, $i \neq j$ and, hence S is a clique partition of G.

Now we need to show that the partition is maximal. Assume by contradiction that it is not. Then there are $C_i, C_j \in S$ such that $C_i \cup C_j$ would be also a clique in G. But then $C_i \cup C_j$ is a subset of an maximal clique in the original clique set where the algorithm starts from. Therefore, Step 5 would block the development of the branch of the algorithm and S would not be computed. A contradiction. □

Theorem 3.3.4. All-Maximal-Clique-Partitions *computes each maximal clique partition exactly once.*

Proof. Since shared vertices are distributed to the cliques they belong (Step 3), the duplication may arise when the same set of shared vertices is collected in a clique with one name in one branch, and in a clique with a different name in another branch. However, the forbidden configurations prevent such cases. They declare which clique (identified by its name) may consist of which shared vertices only. Such clique names are uniquely determined. The same set of shared vertices cannot form a clique with different names in different partitions. Therefore, the second part of Step 5 prevents to compute the same partition more than once. □

Theorem 3.3.5. All-Maximal-Clique-Partitions *computes all (and only) maximal clique partitions.*

Proof. We need to prove that the steps 4 and 5 do not eliminate any maximal clique partitions.

Step 4 prevents to generate clique sets containing cliques of the form $C_i \backslash V$ and $C_j \backslash (C_i \backslash V)$ for $V \subset C_i$, where $C_i \sqsubseteq C_j$. Such a clique set S_0 will be subsumed by a clique set S_1 obtained by taking $V = C_i$. Therefore, for each partition P_0 originating from S_0 there will be a partition P_1 originating from S_1 such that $P_0 \sqsubseteq P_1$. Hence, removing the execution branch which cuts S_0 will not eliminate any maximal clique partition of the given graph.

The first part of Step 5 prevents to generate clique sets containing nonempty cliques C_i and C_j such that $C_i \cup C_j \sqsubseteq C$, where C is one of the original maximal cliques. Such a clique set S_0 will be subsumed by a clique set S_1 which retains C. Hence, no maximal clique partition is lost by eliminating S_0 and proceeding to compute partitions from S_1.

Forbidden configurations simply prevent the same partitions from reappearing under different names. Also here, there is no danger of losing a maximal partition. □

Given a graph G, the clique-multiplicity number of a vertex v of G is the number of cliques in the set of all maximal cliques of G to which v belongs: $\mathit{clique\text{-}mult}(v, G) = |\{C \mid C \in \mathit{all\text{-}max\text{-}cliques}(G) \text{ and } v \in C\}|$.

Theorem 3.3.6. *Let G be a graph with the set of vertices $\{v_1, \dots, v_n\}$. Then the cardinality of the set of all maximal clique partitions of G is at most $\prod_{i=1}^{n} \mathit{clique\text{-}mult}(v_i, G)$.*

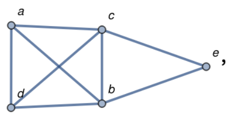
Proof. The result directly follows from the fact that shared vertices are distributed in their containing cliques in all possible ways. □

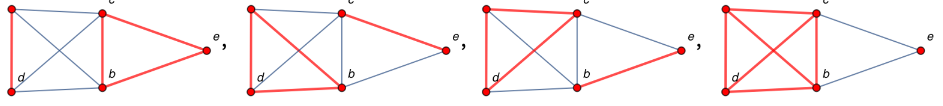
It is easy to see that this upper bound can be reached. Just consider the graph with two maximal cliques: $C_1 = \{p_1, \dots, p_n, \mathit{true}\}$ and $C_2 = \{p_1, \dots, p_n, \mathit{false}\}$. The set of all maximal clique partitions imitates the truth assignment in propositional logic, containing 2^n maximal clique partitions.

This theorem implies that the algorithm is exponential in the number of vertices shared among multiple cliques. On the other hand, the length of each branch of the algorithm is polynomially bounded, since it requires at most as many steps as there are vertices shared among multiple cliques. Besides, the branches can be executed independently, in parallel of each other.

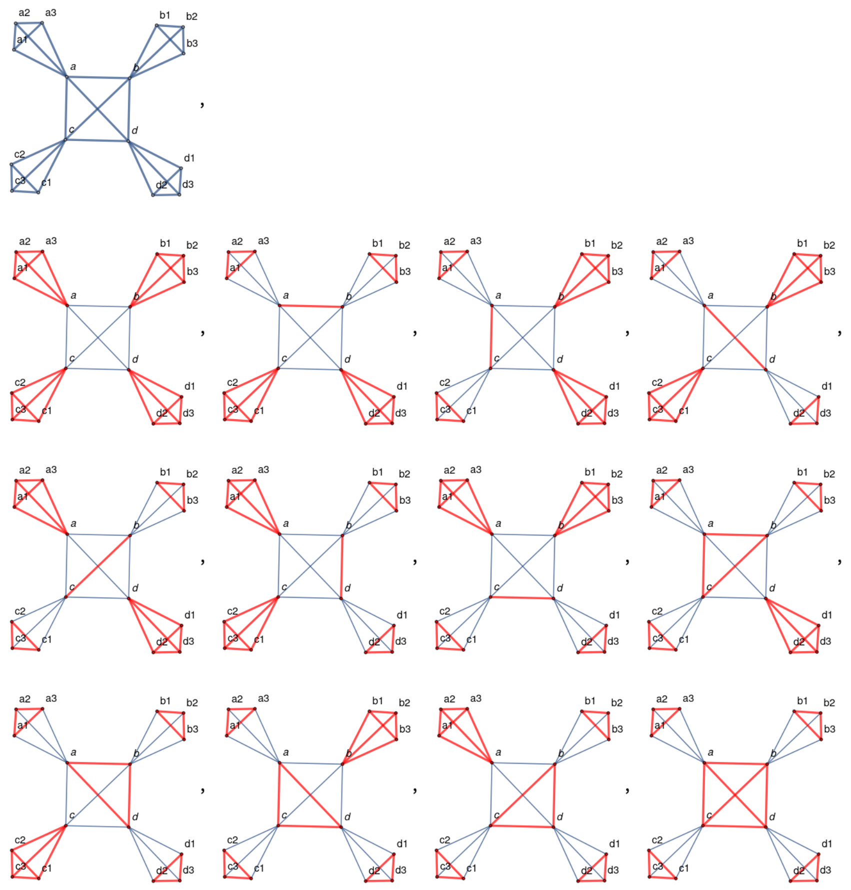
Example 3.3.8. Here we show some results an experimental Mathematica implementation of our algorithm produces. The given graph is in gray, while each maximal clique partition is shown in a separate graph in red:

1. Graph with 5 vertices, 8 edges, and 4 maximal clique partitions:





2. Graph with 16 vertices, 30 edges, and 12 maximal clique partitions:



3.4 Conclusion

Block-based approach to proximity relations has some interesting applications, where the proximal elements need to ‘choose side’. This corresponds to maximal clique partition in the graph representation of the proximity relation. We presented in this chapter an algorithm for anti-unification in such a setting. Each solution computed by our algorithm corresponds to a different maximal clique partitions. It required an algorithm to compute all maximal vertex-clique partitions in an undirected graph. We designed such an algorithm and used it in the anti-unification algorithm. It starts from a set of all maximal cliques and refines it, reducing the number of shared vertices by assigning them to one of the cliques they belong to. In this process, we avoid computing and discarding false answers by detecting the failing branches early, and compute each partition only once. The set of computed partitions can be exponentially large with respect to the number of vertices shared among multiple cliques. Each partition can be computed in polynomial time (starting from all maximal cliques). The algorithm can be also used to compute a limited number of maximal partitions. Guiding by heuristics for choosing shared nodes, one can give the priority to one kind of partitions over the others, computing the preferred ones earlier. It plays an important role in anti-unification, guaranteeing to generate different incomparable generalizations. It may have other interesting applications as well, for instance, in the resource allocation problem, when one looks for alternative ways to allocate resources.

We proved termination, soundness and completeness of both algorithms.

Chapter 4

Class-Based Symbolic Techniques for Proximity Relations

The previous chapter described algorithms that consider proximity relations from the block-based perspective. In that setting, a symbol cannot be close to two symbols at the same time, when those symbols are not close to each other. One of the two symbols should be chosen as the proximal candidate to the initial one. For the unification and matching algorithms this means that $p(x, x)$ cannot be unified, respectively matched with $p(a, c)$ when a and c are not close to each other, even if there exists a b which is close both to a and c. In this chapter we relax this constraint, which leads to the so-called class-based approach to unification, matching and anti-unification for proximity relations.

4.1 Notions and terminology

Extended terms and substitutions. In this chapter we need to extend the notion of *term* to include, beside variables and function symbols, finite sets of function symbols, whose elements have the same arity. They will be denoted by lower case bold face letters: $\mathbf{f}, \mathbf{g}, \mathbf{h}$. If we want to talk about finite sets of constants, we use the letters $\mathbf{a}, \mathbf{b}$, and $\mathbf{c}$.

Extended terms or, shortly, *X-terms* over $\mathcal{F}$ and $\mathcal{V}$ are defined by the grammar

$$\mathbf{t} := x \mid \mathbf{f}(\mathbf{t}_1, \ldots, \mathbf{t}_n), \quad n \geqslant 0,$$

where $\mathbf{f} \neq \varnothing$ 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 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. Analogously to denoting by $\mathcal{V}(t)$ the set of variables for a term t, we will denote by $\mathcal{V}(\mathbf{t})$ the set of variables for an X-term $\mathbf{t}$.

The standard notions related to terms defined in Chapter 2, extend to X-terms. For completeness, we list them here.

An X-term is called *linear* if every variable occurs in it at most once. The *head* of an X-term is defined as

$$\begin{aligned} &\mathit{head}(x) := x, \\ &\mathit{head}(\mathbf{f}(\mathbf{t}_1, \dots, \mathbf{t}_n)) := \mathbf{f}. \end{aligned}$$

The notions of *positions* and *subterm* extend straightforwardly 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 positions $\mathbf{t}|_\epsilon = \mathbf{t}$, $\mathbf{t}|_1 = \{g, h\}(x, \{a, b, c\})$, $\mathbf{t}|_{1.1} = x$, $\mathbf{t}|_{1.2} = \{a, b, c\}$, and $\mathbf{t}|_2 = \{b, c, d\}$.

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

$$\begin{aligned} &\mathsf{terms}(x) := \{x\}, \\ &\mathsf{terms}(\mathbf{f}(\mathbf{t}_1, \dots, \mathbf{t}_n)) := \{f(t_1, \dots, t_n) \mid f \in \mathbf{f},\ t_i \in \mathsf{terms}(\mathbf{t}_i),\ 1 \leqslant i \leqslant n\}. \end{aligned}$$

For a term t, a proximity relation $\mathcal{R}$, and a cut value λ, we can represent compactly the $(\mathcal{R}, \lambda)$-proximity class of t as an X-term $\mathbf{xpc}(t, \mathcal{R}, \lambda)$, defined as follows:

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

It is easy to see that $\mathbf{xpc}(t, \mathcal{R}, \lambda)$ is indeed a representation of the $(\mathcal{R}, \lambda)$-proximity class of t, since $\mathbf{pc}(t, \mathcal{R}, \lambda) = \mathsf{terms}(\mathbf{xpc}(t, \mathcal{R}, \lambda))$.

The example below illustrates these notions.

Example 4.1.1. Let the proximity relation $\mathcal{R}$ be defined as

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

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

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

We illustrate the relation to proximity classes of $f(g_1(a_1), g_2(a_2))$ for the case $0.7 < \lambda \leqslant 0.8$:

$$\begin{aligned}
&\mathsf{terms}(\mathbf{xpc}(f(g_1(a_1), g_2(a_2)), \mathcal{R}, \lambda)) =\\
&\quad \mathsf{terms}(\{f\}(\{g_1\}(\{a_1\}), \{g_2, h_2\}(\{a_2, b\}))) =\\
&\quad \{f(g_1(a_1), g_2(a_2)),\ f(g_1(a_1), g_2(b)),\ f(g_1(a_1), h_2(a_2)),\ f(g_1(a_1), h_2(b))\} =\\
&\quad \mathbf{pc}(f(g_1(a_1), g_2(a_2)), \mathcal{R}, \lambda).
\end{aligned}$$

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

- $x \sqcap x = x$ for all $x \in \mathcal{V}$.
- $\mathbf{t} \sqcap \mathbf{s} = (\mathbf{f} \cap \mathbf{g})(\mathbf{t}_1 \sqcap \mathbf{s}_1, \dots, \mathbf{t}_n \sqcap \mathbf{s}_n)$, $n \geqslant 0$, if $\mathbf{f} \cap \mathbf{g} \neq \varnothing$ and $\mathbf{t}_i \sqcap \mathbf{s}_i \neq \varnothing$ for all $1 \leqslant i \leqslant n$, where $\mathbf{t} = \mathbf{f}(\mathbf{t}_1, \dots, \mathbf{t}_n)$ and $\mathbf{t} = \mathbf{g}(\mathbf{s}_1, \dots, \mathbf{s}_n)$.
- $\mathbf{t} \sqcap \mathbf{s} = \varnothing$ in all other cases.

Theorem 4.1.1. *Given a proximity relation $\mathcal{R}$, a cut value λ, and two terms t and s, each $r \in \mathsf{terms}(\mathbf{xpc}(t, \mathcal{R}, \lambda) \sqcap \mathbf{xpc}(s, \mathcal{R}, \lambda))$ is $(\mathcal{R}, \lambda)$-close both to t and to s.*

Proof. Follows directly from the definition of **xpc**, $\sqcap$ and terms. $\square$

Substitutions over $\mathcal{T}_{\mathsf{ext}}(\mathcal{F}, \mathcal{V})$ are mappings from variables to X-terms, where all but finitely many variables are mapped to themselves. We use the term "*X-substitution*", and denote them by bold upright Greek letters $\boldsymbol{\sigma}$, $\boldsymbol{\vartheta}$, $\boldsymbol{\varphi}$, $\boldsymbol{\mu}$, $\boldsymbol{\nu}$, and $\boldsymbol{\xi}$.

The *domain* of an X-substitution $\boldsymbol{\sigma}$ is defined as $dom(\boldsymbol{\sigma}) = \{x \mid \boldsymbol{\sigma}(x) \neq x\}$. We use the usual set notation for substitutions, writing, e.g., $\boldsymbol{\sigma}$ as $\boldsymbol{\sigma} = \{x \mapsto \boldsymbol{\sigma}(x) \mid x \in dom(\boldsymbol{\sigma})\}$. *X-substitution application* to X-terms is defined recursively as $x\boldsymbol{\sigma} = \boldsymbol{\sigma}(x)$ and $\mathbf{f}(\mathbf{t}_1, \ldots, \mathbf{t}_n)\boldsymbol{\sigma} = \mathbf{f}(\mathbf{t}_1\boldsymbol{\sigma}, \ldots, \mathbf{t}_n\boldsymbol{\sigma})$. We can treat terms as a special case of X-terms, e.g., $f(a, g(b, x))$ can be treated as $\{f\}(\{a\}, \{g\}(\{b\}, x))$ (and similarly for X-substitutions). Taking this convention into account, we can have applications such as $\mathbf{t}\sigma$ (substitution to an X-term) and $t\boldsymbol{\sigma}$ (X-substitution to a term) defined.

The *set of substitutions represented by an X-substitution* $\boldsymbol{\sigma}$ is the set

$$\mathsf{substs}(\boldsymbol{\sigma}) := \{\sigma \mid \sigma(x) \in \mathsf{terms}(\boldsymbol{\sigma}(x)) \text{ for all } x \in \mathcal{V}\}.$$

The restriction of an X-substitution $\boldsymbol{\sigma}$ to a set of variables Var is denoted by $\boldsymbol{\sigma}|_{Var} := \{x \mapsto x\boldsymbol{\sigma} \mid x \in dom(\boldsymbol{\sigma}) \cap Var\}$.

4.2 Unification

In order to address the specificities of our class-based unification algorithm, some notions that are relevant only to this section need to be defined.

4.2.1 Additional notions

Name-neighborhood mappings. Consider $\mathcal{N}$ a countable *set of names*, which are symbols with associated arity (like function symbols). We use the letters $\mathrm{N}, \mathrm{M}, \mathrm{K}$ for them. It is assumed that $\mathcal{N} \cap \mathcal{F} = \varnothing$ and $\mathcal{N} \cap \mathcal{V} = \varnothing$.

Neighborhood is either a name, or a finite subset of $\mathcal{F}$, where all elements have the same arity. In this section we extend the notion of *extended term* to also include names appearing at any functional position, thus being defined over $\mathcal{F}$, $\mathcal{N}$, and $\mathcal{V}$. The set of X-terms is denoted by $\mathcal{T}_{\mathsf{ext}}(\mathcal{F}, \mathcal{N}, \mathcal{V})$. All other definitions related to the X-term remain the same. Instead of the finite sets of symbols, the new *extended term* will contain neighborhoods. We will use upper case bold face letters $\mathbf{F}$ and $\mathbf{G}$ to denote the neighborhoods. $arity(\mathbf{F})$ is defined as the arity of elements of $\mathbf{F}$. The set of all neighborhoods is denoted by Nb.

In this section, when we speak about X-terms, by default we mean X-terms from $\mathcal{T}_{\mathsf{ext}}(\mathcal{F}, \mathcal{N}, \mathcal{V})$. When we refer to an X-term from $\mathcal{T}_{\mathsf{ext}}(\mathcal{F}, \mathcal{V})$, we either mention the set explicitly or say *name-free X-term*.

X-terms, in which every neighborhood set is a singleton, are called *singleton X-terms* or, shortly, *SX-terms*. Slightly abusing the notation, we assume that a term (i.e., an element of $\mathcal{T}(\mathcal{F}, \mathcal{V})$) is a special case of an SX-term (as an SX-term without names), identifying a function symbol f with the singleton neighborhood $\{f\}$. We will use this assumption in the rest of the section.

The set of names occurring in an X-term $\mathbf{t}$ is denoted by $\mathcal{N}(\mathbf{t})$. Approximate extended equations (X-equations) are pairs of X-terms.

A *name-neighborhood mapping* $\Phi : \mathcal{N} \longrightarrow \mathsf{Nb}\backslash\mathcal{N}$ is a finite mapping from names to non-name neighborhoods (i.e., finite sets of function symbols of the same arity) such that if $\mathrm{N} \in dom(\Phi)$ (where *dom* is the domain of mapping), then $arity(\mathrm{N}) = arity(\Phi(\mathrm{N}))$. They are also represented as finite sets, writing Φ as $\{\mathrm{N} \mapsto \Phi(\mathrm{N}) \mid \mathrm{N} \in dom(\Phi)\}$.

A name-neighborhood mapping Φ can *apply* to an X-term $\mathbf{t}$, resulting in an X-term $\Phi(\mathbf{t})$, which is obtained by replacing each name N in $\mathbf{t}$ by the neighborhood $\Phi(\mathrm{N})$. The *application of* Φ *to a set of X-equations* P*,* denoted by $\Phi(P)$, is a set of equations obtained from P by applying Φ to both sides of each equation in P.

Proximity relations over X-terms. We twist a bit the proximity relation $\mathcal{R}$ to be defined on the set $\mathsf{Nb} \cup \mathcal{V}$ (where neighborhoods are assumed to be nonempty) in such a way that it satisfies the following conditions (in addition to reflexivity and symmetry):

(a) $\mathcal{R}(\mathbf{F}, \mathbf{G}) = 0$ if $arity(\mathbf{F}) \neq arity(\mathbf{G})$;

(b) $\mathcal{R}(\mathbf{F}, \mathbf{G}) = \min\{\mathcal{R}(f, g) \mid f \in \mathbf{F}, g \in \mathbf{G}\}$, if $\mathbf{F} = \{f_1, \ldots, f_n\}$, $\mathbf{G} = \{g_1, \ldots, g_m\}$, $n, m > 0$, and $arity(\mathbf{F}) = arity(\mathbf{G})$;

(c) $\mathcal{R}(\mathrm{N}, \mathbf{F}) = 0$, if $\mathbf{F} \notin \mathcal{N}$.

(d) $\mathcal{R}(\mathrm{N}, \mathrm{M}) = 0$, if $\mathrm{N} \neq \mathrm{M}$;

(e) $\mathcal{R}(x, y) = 0$, if $x \neq y$ for all $x, y \in \mathcal{V}$.

$\mathcal{R}(\mathbf{F}, \mathbf{G})$ is undefined, if $\mathbf{F} = \varnothing$ or $\mathbf{G} = \varnothing$.

We write $\mathbf{F} \approx_{\mathcal{R},\lambda} \mathbf{G}$ if $\mathcal{R}(\mathbf{F}, \mathbf{G}) \geqslant \lambda$. Note that for $\mathbf{F} = \{f_1, \ldots, f_n\}$ and $\mathbf{G} = \{g_1, \ldots, g_m\}$, $\mathbf{F} \approx_{\mathcal{R},\lambda} \mathbf{G}$ is equivalent to $\mathcal{R}(f, g) \geqslant \lambda$ for all $f \in \mathbf{F}$ and $g \in \mathbf{G}$. It is easy to see that the obtained relation is again a proximity relation. Furthermore, it can be extended to X-terms (which do not contain the empty neighborhood):

1. $\mathcal{R}(\mathbf{s}, \mathbf{t}) := 0$ if $\mathcal{R}(head(\mathbf{s}), head(\mathbf{t})) = 0$.

2. $\mathcal{R}(\mathbf{s}, \mathbf{t}) := 1$ if $\mathbf{s} = \mathbf{t}$ and $\mathbf{s}, \mathbf{t} \in \mathcal{V}$.

3. $\mathcal{R}(\mathbf{s}, \mathbf{t}) := \mathcal{R}(\mathbf{F}, \mathbf{G}) \wedge \mathcal{R}(\mathbf{s}_1, \mathbf{t}_1) \wedge \cdots \wedge \mathcal{R}(\mathbf{s}_n, \mathbf{t}_n)$, if $\mathbf{s} = \mathbf{F}(\mathbf{s}_1, \ldots, \mathbf{s}_n)$, $\mathbf{t} = \mathbf{G}(\mathbf{t}_1, \ldots, \mathbf{t}_n)$.

$\mathcal{R}(\mathbf{s}, \mathbf{t})$ is not defined, if $\mathbf{s}$ or $\mathbf{t}$ contains the empty neighborhood $\varnothing$. The obtained relation is a proximity relation for X-terms.

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

Neighborhood equations, unification problems. We introduce the notions of problems we would like to solve.

Definition 4.2.1 (Neighborhood equations). *Given $\mathcal{R}$ and λ, an $(\mathcal{R}, \lambda)$-*neighborhood equation *is a pair of neighborhoods, written as $\mathbf{F} \approx^?_{\mathcal{R},\lambda} \mathbf{G}$. The question mark indicates that it has to be solved.*

A name-neighborhood mapping Φ is a solution *of an $(\mathcal{R}, \lambda)$-neighborhood equation $\mathbf{F} \approx^?_{\mathcal{R},\lambda} \mathbf{G}$ if $\Phi(\mathbf{F}) \approx_{\mathcal{R},\lambda} \Phi(\mathbf{G})$. The notation implies that $\mathcal{R}(\Phi(\mathbf{F}), \Phi(\mathbf{G}))$ is defined, i.e., neither $\Phi(\mathbf{F})$ nor $\Phi(\mathbf{G})$ contains the empty neighborhood.*

An $(\mathcal{R}, \lambda)$-neighborhood constraint is a finite set of $(\mathcal{R}, \lambda)$-neighborhood equations. A name-neighborhood mapping Φ is a solution of an $(\mathcal{R}, \lambda)$-neighborhood constraint C if it is a solution of every $(\mathcal{R}, \lambda)$-neighborhood equation in C.

We shortly write "an $(\mathcal{R}, \lambda)$-solution to C" instead of "a solution to an $(\mathcal{R}, \lambda)$-neighborhood constraint C".

To each X-term $\mathbf{t}$ we associate a set of SX-terms $\mathsf{SX\text{-}terms}(\mathbf{t})$ defined as follows:

$$\begin{aligned}
&\mathsf{SX\text{-}terms}(x) := \{x\},\\
&\mathsf{SX\text{-}terms}(\mathrm{N}(\mathbf{t}_1, \ldots, \mathbf{t}_n)) :=\\
&\qquad \{\mathrm{N}(\mathbf{s}_1, \ldots, \mathbf{s}_n) \mid \mathbf{s}_i \in \mathsf{SX\text{-}terms}(\mathbf{t}_i),\ 1 \leqslant i \leqslant n\}.\\
&\mathsf{SX\text{-}terms}(\mathbf{F}(\mathbf{t}_1, \ldots, \mathbf{t}_n)) :=\\
&\qquad \{f(\mathbf{s}_1, \ldots, \mathbf{s}_n) \mid f \in \mathbf{F},\ \mathbf{s}_i \in \mathsf{SX\text{-}terms}(\mathbf{t}_i),\ 1 \leqslant i \leqslant n\},\ \text{where } \mathbf{F} \notin \mathcal{N}.
\end{aligned}$$

The notation extends to substitutions as well:

$$\mathsf{SX\text{-}substs}(\boldsymbol{\mu}) := \{\vartheta \mid x\vartheta \in \mathsf{SX\text{-}terms}(x\boldsymbol{\mu}) \text{ for all } x \in \mathcal{V}\}.$$

There is a natural correspondence between the notions of $\mathsf{SX\text{-}terms}$ and terms, and between $\mathsf{SX\text{-}substs}$ and substs. The figure below summarizes it:

X-terms and X-substitutions over $\mathcal{T}_{\mathsf{ext}}(\mathcal{F}, \mathcal{V})$	X-terms and X-substitutions over $\mathcal{T}_{\mathsf{ext}}(\mathcal{F}, \mathcal{N}, \mathcal{V})$
$\mathsf{terms}(\mathbf{t})$	$\mathsf{SX\text{-}terms}(\mathbf{t})$
$\mathsf{substs}(\boldsymbol{\mu})$	$\mathsf{SX\text{-}substs}(\boldsymbol{\mu})$

Definition 4.2.2 (Approximate X-unification). *Given $\mathcal{R}$ and λ, a finite set P of $(\mathcal{R}, \lambda)$-equations between X-terms is called an $(\mathcal{R}, \lambda)$-X-unification problem. A mapping-substitution pair $(\Phi, \boldsymbol{\mu})$ is called an $(\mathcal{R}, \lambda)$-*solution *of an $(\mathcal{R}, \lambda)$-X-equation $\mathbf{t} \simeq^?_{\mathcal{R},\lambda} \mathbf{s}$, if $\Phi(\mathbf{t}\boldsymbol{\mu}) \simeq_{\mathcal{R},\lambda} \Phi(\mathbf{s}\boldsymbol{\mu})$. An $(\mathcal{R}, \lambda)$-solution of P is a pair $(\Phi, \boldsymbol{\mu})$ which solves each equation in P.*

*If $(\Phi, \boldsymbol{\mu})$ is an $(\mathcal{R}, \lambda)$-solution of P, then the X-substitution $\Phi(\boldsymbol{\mu})$ is called an $(\mathcal{R}, \lambda)$-*X-unifier *of P.*

SX-unification problems, SX-solutions and SX-unifiers are defined analogously. For unification between terms, we do not use any prefix, talking about unification problems, solutions, and unifiers.

Unification between terms. Our approximate unification problems will be formulated between terms. The notion of *set of unifiers* for such problems has been already introduced in Definition 2.3.1. Extended terms will not be a part of the problem (but the input terms will be treated as a special case of SX-terms). We will need X-terms and X-substitutions in the formulation of the algorithms, in proving their properties, and in representing the solutions of term unification problems in a compact form.

4.2.2 The algorithm

We start with a high-level view of the process of solving an approximate unification problem $s \simeq^?_{\mathcal{R},\lambda} t$ between terms s and t (we omit $\mathcal{R}$ and λ below):

- First, we treat the input equation as an SX-equation and apply rules of the pre-unification algorithm. Pre-unification works on SX-equations. It either fails (in this case the input terms are not unifiable) or results in a set of equations V between variables (variables-only constraint V), a neighborhood constraint C and a substitution $\boldsymbol{\mu}$ over $\mathcal{T}_{\mathsf{ext}}(\varnothing, \mathcal{N}, \mathcal{V})$ such that $dom(\boldsymbol{\mu}) \cap \mathcal{V}(V) = \varnothing$.

- Next, we solve C by the neighborhood constraint solving algorithm. If the process fails, then the input terms are not unifiable. Otherwise, we get a finite set of name-neighborhood mappings $\mathcal{M} = \{\Phi_1, \dots, \Phi_n\}$. Note that Φ's do not necessarily map names to singleton sets here. Note also that Φ_i's map all names occurring in $\boldsymbol{\mu}$ to sets of function symbols, i.e. $\Phi_i(\boldsymbol{\mu})$ is name-free for all $1 \leqslant i \leqslant n$.

- For each $\Phi_i \in \mathcal{M}$ and each X-unifier $\boldsymbol{\nu}$ of V, the pair $(\Phi_i, \boldsymbol{\mu\nu})$ solves the original unification problem, i.e., the X-substitution $\Phi_i(\boldsymbol{\mu\nu})$ is an X-unifier of it. Moreover, if $\boldsymbol{\nu}$ contains no names except those occurring in $\boldsymbol{\mu}$, then $\Phi_i(\boldsymbol{\mu\nu})$ is name-free.

- The obtained set $\{\Phi_1(\boldsymbol{\mu}), \dots, \Phi_n(\boldsymbol{\mu})\}$ and the variables-only constraint V are related to the minimal complete set $mcsu_{\mathcal{R},\lambda}(s,t)$ of $(\mathcal{R},\lambda)$-unifiers of s and t in the following way: For each $\sigma \in mcsu_{\mathcal{R},\lambda}(s,t)$ there exist $\mu \in \mathsf{substs}(\Phi_i(\boldsymbol{\mu}))$ for some $1 \leqslant i \leqslant n$ and an $(\mathcal{R},\lambda)$-unifier ν of V such that $\sigma = \mu\nu$. Note that $\mathsf{substs}(\Phi_i(\boldsymbol{\mu}))$ is defined since $\Phi_i(\boldsymbol{\mu})$ is name-free.

Hence, the algorithm consists of two phases: pre-unification and constraint solving. They are described in separate subsections below.

4.2.3 Pre-unification rules

We start with the definition of a technical notion needed later:

Definition 4.2.3. *We say that a set of SX-equations* $\{x \simeq^?_{\mathcal{R},\lambda} \mathbf{t}\} \uplus P$ *contains an* occurrence cycle *for the variable* x *if* $\mathbf{t} \notin \mathcal{V}$ *and there exist SX-term-pairs* $(x_0, \mathbf{t}_0), (x_1, \mathbf{t}_1), \dots, (x_n, \mathbf{t}_n)$ *such that* $x_0 = x$, $\mathbf{t}_0 = \mathbf{t}$, *for each* $0 \leqslant i \leqslant n$ P *contains an equation* $x_i \simeq^?_{\mathcal{R},\lambda} \mathbf{t}_i$ *or* $\mathbf{t}_i \simeq^?_{\mathcal{R},\lambda} x_i$, *and* $x_{i+1} \in \mathcal{V}(\mathbf{t}_i)$ *where* $x_{n+1} = x_0$.

Lemma 4.2.1. *If a set of SX-equations* P *contains an occurrence cycle for some variable, then* P *has no* $(\mathcal{R},\lambda)$*-solution for any* $\mathcal{R}$ *and* λ.

Proof. The requirement that neighborhoods of different arity are not $(\mathcal{R},\lambda)$-close to each other guarantees that an SX-term cannot be $(\mathcal{R},\lambda)$-close to its proper subterm. Therefore, equations containing an occurrence cycle cannot have an $(\mathcal{R},\lambda)$-solution. □

In the rules below we will use the *renaming function* ρ that works on (SX-)terms from $\mathcal{T}_{\mathsf{ext}}(\mathcal{F}, \mathcal{N}, \mathcal{V})$ and gives a term from $\mathcal{T}(\mathcal{N}, \mathcal{V})$. Applied to a term, ρ gives its fresh copy, obtained by replacing each occurrence of a symbol from $\mathcal{F} \cup \mathcal{N}$ by a new name and each variable occurrence by a fresh variable. For instance, if the term is $f(\mathrm{N}(a, x, x, f(a)))$, where $f, a \in \mathcal{F}$ and $\mathrm{N} \in \mathcal{N}$, then $\rho(f(\mathrm{N}(a, x, x, f(a))) = \mathrm{N}_1(\mathrm{N}_2(\mathrm{N}_3, x_1, x_2, \mathrm{N}_4(\mathrm{N}_5)))$, where $\mathrm{N}_1, \mathrm{N}_2, \mathrm{N}_3, \mathrm{N}_4, \mathrm{N}_5 \in \mathcal{N}$ are new names and x_1, x_2 are new variables.

Given $\mathcal{R}$ and λ, an *equational* $(\mathcal{R}, \lambda)$*-configuration* is a triple $P; C; \boldsymbol{\mu}$, where

- P is a finite set of $(\mathcal{R}, \lambda)$-SX-equations. It is initialized with the unification equation between the original terms;
- C is an $(\mathcal{R}, \lambda)$-neighborhood constraint;
- $\boldsymbol{\mu}$ is an X-substitution over $\mathcal{T}_{\mathsf{ext}}(\varnothing, \mathcal{N}, \mathcal{V})$, initialized by Id. It serves as an accumulator, keeping the pre-unifier computed so far.

The pre-unification algorithm takes the given terms s and t, creates the initial configuration $\{s \simeq^?_{\mathcal{R},\lambda} t\}; \varnothing; Id$ and applies the rules given below exhaustively.

The rules are very similar to the syntactic unification algorithm with the difference that here the function symbol clash does not happen unless their arities differ, and variables are not replaced by other variables. (The notation $\overline{exp_n}$ in the rules below abbreviates the sequence $exp_1, \ldots, exp_n$.)

(Tri) Trivial: $\{x \simeq^?_{\mathcal{R},\lambda} x\} \uplus P;\ C;\ \boldsymbol{\mu} \Longrightarrow P;\ C;\ \boldsymbol{\mu}$.

(Dec) Decomposition:

$\{\mathbf{F}(\overline{\mathbf{s}_n}) \simeq^?_{\mathcal{R},\lambda} \mathbf{G}(\overline{\mathbf{t}_n})\} \uplus P; C; \boldsymbol{\mu} \Longrightarrow \overline{\{\mathbf{s}_n \simeq^?_{\mathcal{R},\lambda} \mathbf{t}_n\}} \cup P; \{\mathbf{F} \approx^?_{\mathcal{R},\lambda} \mathbf{G}\} \cup C; \boldsymbol{\mu}$, where each of $\mathbf{F}$ and $\mathbf{G}$ is a name or a function symbol treated as a singleton neighborhood.

(VE) Variable Elimination:

$\{x \simeq^?_{\mathcal{R},\lambda} \mathbf{t}\} \uplus P;\ C;\ \boldsymbol{\mu} \Longrightarrow \{\mathbf{t}' \simeq^?_{\mathcal{R},\lambda} \mathbf{t}\} \cup P\{x \mapsto \mathbf{t}'\};\ C;\ \boldsymbol{\mu}\{x \mapsto \mathbf{t}'\}$, where $\mathbf{t} \notin \mathcal{V}$, there is no occurrence cycle for x in $\{x \simeq^?_{\mathcal{R},\lambda} \mathbf{t}\} \uplus P$, and $\mathbf{t}' = \rho(\mathbf{t})$.

(Ori) Orient: $\{\mathbf{t} \simeq^?_{\mathcal{R},\lambda} x\} \uplus P;\ C;\ \boldsymbol{\mu} \Longrightarrow \{x \simeq^?_{\mathcal{R},\lambda} \mathbf{t}\} \cup P;\ C;\ \boldsymbol{\mu}$, if $\mathbf{t} \notin \mathcal{V}$.

(Cla) Clash: $\mathbf{F}(\overline{\mathbf{s}_n}) \simeq^?_{\mathcal{R},\lambda} \mathbf{G}(\overline{\mathbf{t}_m})\} \uplus P;\, C;\, \boldsymbol{\mu} \Longrightarrow \bot$, where $n \neq m$.

(Occ) Occur Check: $\{x \simeq^?_{\mathcal{R},\lambda} \mathbf{t}\} \uplus P;\, C;\, \boldsymbol{\mu} \Longrightarrow \bot$,
if there is an occurrence cycle for x in $\{x \simeq^?_{\mathcal{R},\lambda} \mathbf{t}\} \uplus P$.

Informally, in the (VE) rule, we mirror the structure of $\mathbf{t}$ in $\mathbf{t}'$ by the function ρ, replace x by $\mathbf{t}'$, and then try to bring $\mathbf{t}'$ close to $\mathbf{t}$ by solving the equation $\mathbf{t}' \simeq^?_{\mathcal{R},\lambda} \mathbf{t}$.

For proving termination of pre-unification, we will need a special complexity measure in the form of directed acyclic graphs (dag) and the corresponding well-founded relation, defined below.

Let G be an directed acyclic graph. Assume that for each vertex V in G there is a finite set $\mathit{mark}(V, G)$ associated to it. It is called the mark of V in G. The graph can be serialized in a standard way (e.g., by topological sorting), leading to a sequence of its vertices $V_1, \dots, V_k$ in which each vertex comes before all vertices to which it has outbound edges. A marked serialization of G is a sequence of pairs $((V_1, \mathit{mark}(V_1, G)), \dots, (V_k, \mathit{mark}(V_k, G)))$, where $(V_1, \dots, V_k)$ is an ordinary serialization of G.

Let G_1 and G_2 be two marked dags that may differ from each other only by vertex markings. Let $V_1, \dots, V_k$ be their serialization. Then we write $G_1 >_{\mathit{mark}} G_2$, iff

$$((V_1, \mathit{mark}(V_1, G_1)), \dots, (V_k, \mathit{mark}(V_k, G_1))) >_{\mathit{mark}}$$
$$((V_1, \mathit{mark}(V_1, G_2)), \dots, (V_k, \mathit{mark}(V_k, G_2))),$$

where the relation $>_{\mathit{mark}}$ on the marked serializations holds iff there exists $1 \leqslant i \leqslant k$ such that $\mathit{mark}(V_j, G_1) = \mathit{mark}(V_j, G_2)$ for all $1 \leqslant j < i$ and $\mathit{mark}(V_i, G_1) \supset \mathit{mark}(V_i, G_2)$ (i.e., lexicographic comparison of the marks). Obviously, $>_{\mathit{mark}}$ is a well-founded ordering on such marked serializations and, consequently, on dags. The relation $\geqslant_{\mathit{mark}}$ is $>_{\mathit{mark}} \cup =$.

Theorem 4.2.1 (Termination of pre-unification)**.** *The pre-unification algorithm terminates either with $\bot$ or with a configuration of the form $V; C; \boldsymbol{\mu}$, where V is a variables-only constraint that can be empty.*

Proof. In the process of pre-unification we maintain a marked dag (illustrating the variable dependencies in the equations). The vertices of such a dag G correspond to variables in the input problem so that each variable has a single vertex assigned. For instance, if the problem contains variables x, y, z, we will have three vertices $\mathit{Vert} = \{V_1, V_2, V_3\}$ such that $\mathit{mark}(V_1, G) = \{x\}$, $\mathit{mark}(V_2, G) = \{y\}$, and $\mathit{mark}(V_3, G) = \{z\}$. We also introduce associations of variables to vertices: $\mathit{vrt}(x, G) = V_1$, $\mathit{vrt}(y, G) = V_2$, and $\mathit{vrt}(z, G) = V_3$.

In the beginning, $E = \emptyset$, i.e., all vertices are isolated. During pre-unification, we may add or remove copies of variables to or from the marks of vertices. New edges may be also added, but the set of vertices Vert remains unchanged. If we encounter an equation of the form $x \simeq^?_{\mathcal{R},\lambda} \mathbf{t}$, such that $x \in \mathit{mark}(V, G)$ and $\mathbf{t}$ contains the variables $y_1 \in \mathit{mark}(V_1, G), \dots, y_n \in \mathit{mark}(V_n, G)$, $n \geqslant 0$, then the **(VE)** rule adds edges $(V, V_1), \dots, (V, V_n)$ to G. This justifies why we call G the variable dependency graph.

Actually, it is only **(VE)** that modifies the variable dependency graph: For instance, assume that, during the process of rule applications, we reach a configuration that is transformed by the **(VE)** rule, applied to an equation $x \simeq^?_{\mathcal{R},\lambda} \mathbf{t}$, where $\mathbf{t}$ contains variables y, z', and z'' (the latter two are copies of z). Assume $\mathit{vrt}(x, G_1) = V_1$, $\mathit{vrt}(y, G_1) = V_2$, and $\mathit{vrt}(z, G_1) = \mathit{vrt}(z', G_1) = \mathit{vrt}(z'', G_1) = V_3$, where G_1 is the current variable dependency graph. The **(VE)** rule creates a fresh copy of $\mathbf{t}$, which contains copies of variables: $\rho(y)$, $\rho(z')$, and $\rho(z'')$. In addition, a new variable dependency graph G_2 is obtained from G_1 by the following modifications: $\mathit{mark}(V_1, G_2) := \mathit{mark}(V_1, G_1) \backslash \{x\}$, $\mathit{mark}(V_2, G_2) := \mathit{mark}(V_2, G_1) \cup \{\rho(y)\}$ and $\mathit{mark}(V_3, G_2) := \mathit{mark}(V_3, G_1) \cup \{\rho(z'), \rho(z'')\}$. Moreover, $\mathit{vrt}(\rho(y), G_2) = V_2$ and $\mathit{vrt}(\rho(z'), G_2) = \mathit{vrt}(\rho(z''), G_2) = V_3$. Besides, if there was no edge connecting the vertex V_1 to the vertices V_2 and V_3, the edges are created and added to G_2. All other edges from G_1 transfer to G_2.

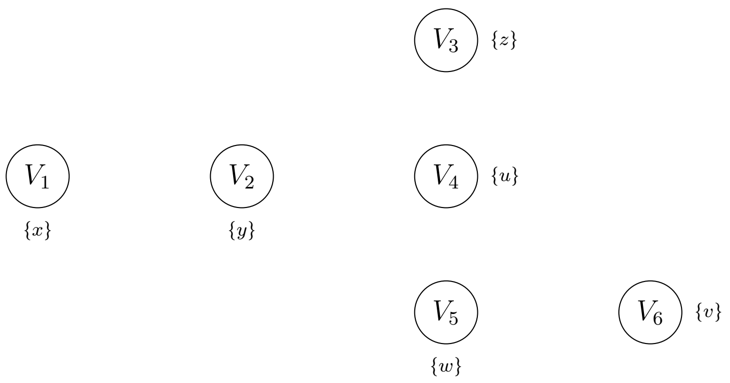
If $\mathbf{t}$ does not contain variables, then then only difference between G_1 and G_2 is in the mark of V_1: $\mathit{mark}(V_1, G_2) := \mathit{mark}(V_1, G_1) \backslash \{x\}$.

In general, if G_1 and G_2 are the variable dependency graphs before and after application of the **(VE)** rule, then either G_2 contains more edges than G_1 (as in the example above), or they as graphs are the same and $G_1 >_{\mathit{mark}} G_2$, because in their marked serializations, the mark of a vertex in G_2 is strictly smaller than the mark of the same vertex in G_1, while the marks of all earlier vertices in the serializations remain unchanged. New edges cannot be created infinitely many times (since the set of vertices does not change and edges are never removed). Their number is bounded from above by $n(n-1)/2$, where n is the number of vertices in the variable dependency graph.

As the termination criterion, consider the lexicographic combination of four measures: (1) the number of missing edges until the variable dependency graph is completed, (2) marked serializations of variable dependency graphs,

(3) the size of the set of equations, and (4) the number of equations with a non-variable term in the left and a variable in the right. If **(Occ)** or **(Cla)** are applicable, the algorithm stops immediately. Otherwise, the non-failing rules strictly decrease the measure. More precisely, **(Tri)** and **(Dec)** decrease (3) without changing (1) and (2); **(VE)** either decreases (1), or leaves it unchanged and decreases (2) with respect to $>_{mark}$; **(Ori)** decreases (4) and leaves the others unchanged. After finitely many steps, either failure will occur, or one reaches the variable-only equations (since no rule transforms the latter) and pre-unification stops. □

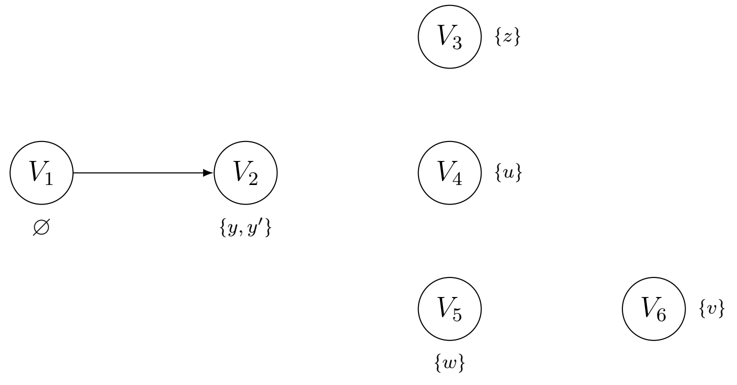
Example 4.2.1. Let $\{x \simeq^?_{\mathcal{R},\lambda} f(y), y \simeq^?_{\mathcal{R},\lambda} g(z,w,u), z \simeq^?_{\mathcal{R},\lambda} f(u), y \simeq^?_{\mathcal{R},\lambda} g(z,p(v),v)\}$ be an $(\mathcal{R},\lambda)$-unification problem, where x, y, z, u, v, w are variables. We show how the marked variable dependency graph changes during the derivation. At the same time, we also indicate the changing of the measures used in the termination proof.



$$\{x \simeq^?_{\mathcal{R},\lambda} f(y),\ y \simeq^?_{\mathcal{R},\lambda} g(z,w,u),\ z \simeq^?_{\mathcal{R},\lambda} f(u),\ y \simeq^?_{\mathcal{R},\lambda} g(z,p(v),v)\}$$

In the beginning, the number of missing edges until the variable dependency graph is completed is 15 ($= 6 \cdot 5/2$); the serialization at this stage does not really matter but later we will see that the order $V_1, \ldots, V_6$ is a relevant one; the size of the problem is 17; and the number of equations with non-variable term in the left and variable in the right is 0. Below we will not write marks in the serializations, they can be read from the graph.

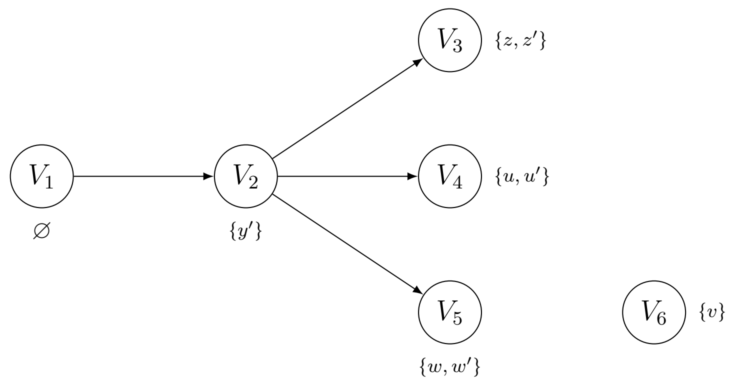
Transforming $x \simeq^?_{\mathcal{R},\lambda} f(y)$ by **(VE)** and the further decomposition gives



$$\{y' \simeq^?_{\mathcal{R},\lambda} y,\ y \simeq^?_{\mathcal{R},\lambda} g(z,w,u),\ z \simeq^?_{\mathcal{R},\lambda} f(u),\ y \simeq^?_{\mathcal{R},\lambda} g(z,p(v),v)\}$$

The maximal number of possible edges to be constructed decreased. It is $15-1=14$. Decomposition further decreases the size of the problem (it was 18 after (VE), became 16 after (Dec)), without affecting the graph.

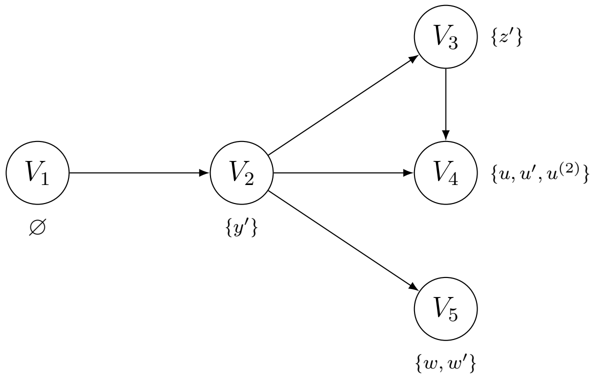
Transforming $y \simeq^?_{\mathcal{R},\lambda} g(z,w,v)$ by (VE) and further decomposition gives



$$\{y' \simeq^?_{\mathcal{R},\lambda} \mathrm{N}_1(z',w',u'),\ z' \simeq^?_{\mathcal{R},\lambda} z,\ w' \simeq^?_{\mathcal{R},\lambda} w,\ u' \simeq^?_{\mathcal{R},\lambda} u,$$
$$z \simeq^?_{\mathcal{R},\lambda} f(u),\ \mathrm{N}_1(z',w',u') \simeq^?_{\mathcal{R},\lambda} g(z,p(v),v)\}$$

The maximal number of possible edges to be constructed decreased. It is $14-3=11$. Decomposition further decreases the size of the problem (it was 25 after (VE), becomes 23 after (Dec)), without affecting the graph.

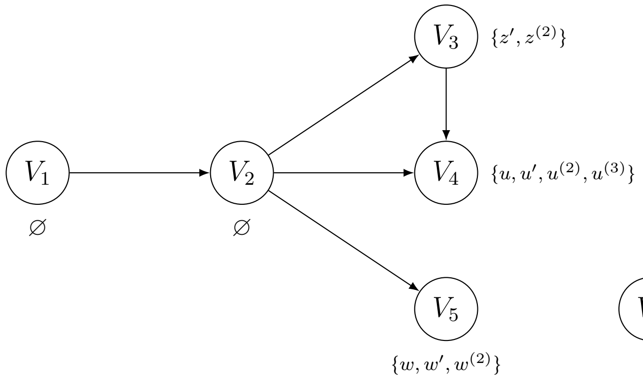
Transforming $z \simeq^?_{\mathcal{R},\lambda} f(u)$ by (VE) and further decomposition gives



$$\{y' \simeq^?_{\mathcal{R},\lambda} \mathrm{N}_1(z', w', u'),\ z' \simeq^?_{\mathcal{R},\lambda} \mathrm{N}_2(u^{(2)}),\ w' \simeq^?_{\mathcal{R},\lambda} w,\ u' \simeq^?_{\mathcal{R},\lambda} u,$$
$$u \simeq^?_{\mathcal{R},\lambda} u^{(2)},\ \mathrm{N}_1(z', w', u') \simeq^?_{\mathcal{R},\lambda} g(\mathrm{N}_2(u^{(2)}), p(v), v)\}$$

The maximal number of possible edges to be constructed decreased. It is $11-1 = 10$. Decomposition further decreases the size of the problem, without affecting the graph.

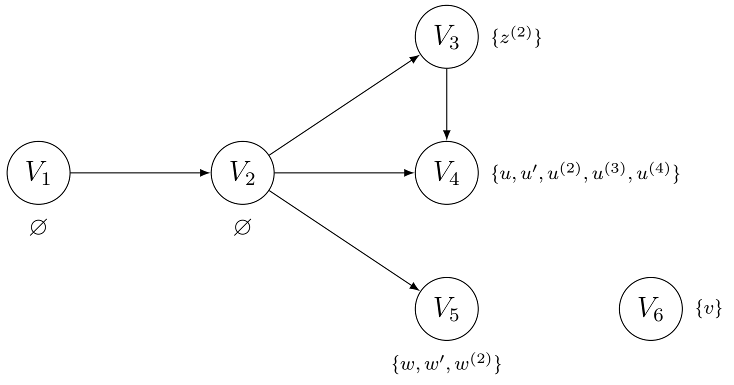
Transforming $y' \simeq^?_{\mathcal{R},\lambda} \mathrm{N}_1(z', w', u')$ by **(VE)** and further decomposition gives



$$\{z^{(2)} \simeq^?_{\mathcal{R},\lambda} z',\ w^{(2)} \simeq^?_{\mathcal{R},\lambda} w',\ u^{(3)} \simeq^?_{\mathcal{R},\lambda} u',\ z' \simeq^?_{\mathcal{R},\lambda} \mathrm{N}_2(u^{(2)}),\ w' \simeq^?_{\mathcal{R},\lambda} w,\ u' \simeq^?_{\mathcal{R},\lambda} u,$$
$$u \simeq^?_{\mathcal{R},\lambda} u^{(2)},\ \mathrm{N}_1(z', w', u') \simeq^?_{\mathcal{R},\lambda} g(\mathrm{N}_2(u^{(2)}), p(v), v)\}$$

The maximal number of possible edges to construct does not change. It is still 10. In the serialization $V_1, \dots, V_6$, the mark for V_1 is the same, but for V_2 it changed from $\{y'\}$ into $\varnothing$. Hence, the graph after the application of **(VE)** becomes strictly smaller with respect to $>_{mark}$. Decomposition further decreases the size of the problem, without affecting the graph.

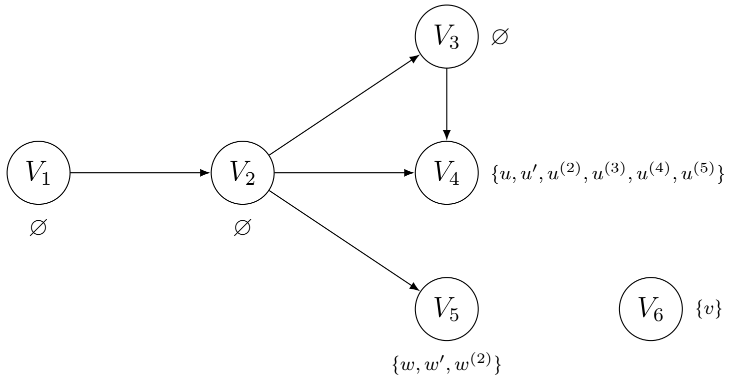
Transforming $z' \simeq^?_{\mathcal{R},\lambda} \mathrm{N}_2(u^{(2)})$ by **(VE)** and further decomposition gives



$$\{z^{(2)} \simeq^?_{\mathcal{R},\lambda} \mathrm{N}_2'(u^{(4)}),\ w^{(2)} \simeq^?_{\mathcal{R},\lambda} w',\ u^{(3)} \simeq^?_{\mathcal{R},\lambda} u',\ u^{(4)} \simeq^?_{\mathcal{R},\lambda} u^{(2)},\ w' \simeq^?_{\mathcal{R},\lambda} w,\ u' \simeq^?_{\mathcal{R},\lambda} u,\ u \simeq^?_{\mathcal{R},\lambda} u^{(2)},\ \mathrm{N}_1(\mathrm{N}_2'(u^{(4)}), w', u') \simeq^?_{\mathcal{R},\lambda} g(\mathrm{N}_2(u^{(2)}), p(v), v)\}$$

Again, the maximal number of possible edges to be constructed does not change, but the serialization decreases with this rule application: The marks for vertices V_1 and V_2 do not change, but it strictly decreases for V_3. Decomposition further decreases the size of the problem, without affecting the graph.

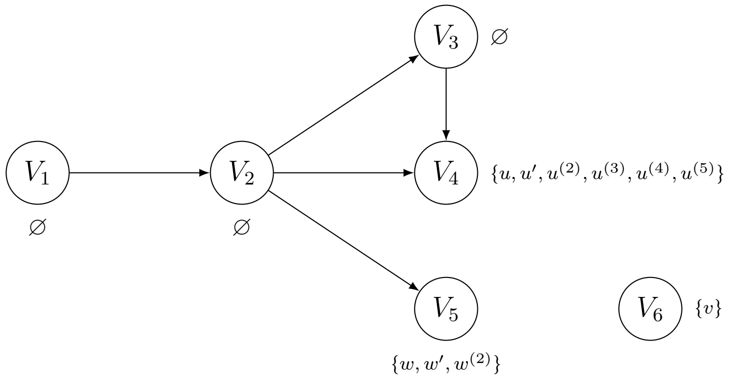
Transforming $z^{(2)} \simeq^?_{\mathcal{R},\lambda} \mathrm{N}_2'(u^{(4)})$ by **(VE)** and further decomposition gives



$$\{u^{(5)} \simeq^?_{\mathcal{R},\lambda} u^{(4)},\ w^{(2)} \simeq^?_{\mathcal{R},\lambda} w',\ u^{(3)} \simeq^?_{\mathcal{R},\lambda} u',\ u^{(4)} \simeq^?_{\mathcal{R},\lambda} u^{(2)},\ w' \simeq^?_{\mathcal{R},\lambda} w,\ u' \simeq^?_{\mathcal{R},\lambda} u,\ u \simeq^?_{\mathcal{R},\lambda} u^{(2)},\ \mathrm{N}_1(\mathrm{N}_2'(u^{(4)}), w', u') \simeq^?_{\mathcal{R},\lambda} g(\mathrm{N}_2(u^{(2)}), p(v), v)\}$$

The same reasoning as for the previous step: the maximal number of possible edges to be constructed does not change, but the serialization decreases due to the mark at V_2. Decomposition further decreases the size of the problem, without affecting the graph.

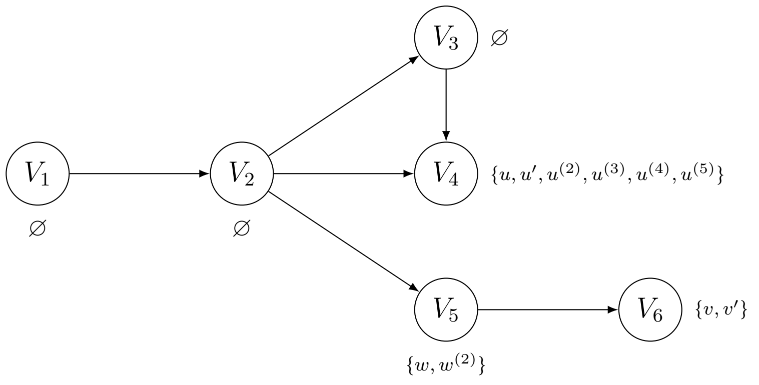
Transforming $\mathrm{N}_1(\mathrm{N}'_2(u^{(4)}), w', u') \simeq^?_{\mathcal{R},\lambda} g(\mathrm{N}_2(u^{(2)}), p(v), v)$ by **(Dec)** and further decomposition of $\mathrm{N}'_2(u^{(4)}) \simeq^?_{\mathcal{R},\lambda} \mathrm{N}_2(u^{(2)})$ gives



$$\{u^{(5)} \simeq^?_{\mathcal{R},\lambda} u^{(4)},\ w^{(2)} \simeq^?_{\mathcal{R},\lambda} w',\ u^{(3)} \simeq^?_{\mathcal{R},\lambda} u',\ u^{(4)} \simeq^?_{\mathcal{R},\lambda} u^{(2)},\ w' \simeq^?_{\mathcal{R},\lambda} w,\ u' \simeq^?_{\mathcal{R},\lambda} u,\ u \simeq^?_{\mathcal{R},\lambda} u^{(2)},\ u^{(4)} \simeq^?_{\mathcal{R},\lambda} u^{(2)},\ w' \simeq^?_{\mathcal{R},\lambda} p(v),\ u' \simeq^?_{\mathcal{R},\lambda} v\}$$

The graph is not affected: the maximal number of possible edges to be constructed and the serialization do not change. The size of the problem decreases. It was 25, and became 21.

Transforming $w' \simeq^?_{\mathcal{R},\lambda} p(v)$ by **(Dec)** and further decomposition gives



$$\{u^{(5)} \simeq^?_{\mathcal{R},\lambda} u^{(4)},\ w^{(2)} \simeq^?_{\mathcal{R},\lambda} \mathrm{N}_3(v'),\ u^{(3)} \simeq^?_{\mathcal{R},\lambda} u',\ u^{(4)} \simeq^?_{\mathcal{R},\lambda} u^{(2)},\ \mathrm{N}_3(v') \simeq^?_{\mathcal{R},\lambda} w,\ u' \simeq^?_{\mathcal{R},\lambda} u,\ u \simeq^?_{\mathcal{R},\lambda} u^{(2)},\ u^{(4)} \simeq^?_{\mathcal{R},\lambda} u^{(2)},\ v' \simeq^?_{\mathcal{R},\lambda} v,\ u' \simeq^?_{\mathcal{R},\lambda} v\}$$

The maximal number of possible edges to be constructed decreases by one. Decomposition further decreases the size of the problem.

Transforming $\mathrm{N}_3(v') \simeq^?_{\mathcal{R},\lambda} w$ by **(Ori)** decrease the number of equations with a nonvariable left-hand side and variable right-hand side by one, without affecting other measures. Further, the obtained equation can be transformed by **(VE)** and **(Dec)**, getting

$$\begin{tikzcd}
 & & V_3 \ \varnothing \arrow[d] & \\
\underset{\varnothing}{V_1} \arrow[r] & \underset{\varnothing}{V_2} \arrow[ur] \arrow[r] \arrow[dr] & V_4 \ \{u, u', u^{(2)}, u^{(3)}, u^{(4)}, u^{(5)}\} & \\
 & & \underset{\{w^{(2)}\}}{V_5} \arrow[r] & V_6 \ \{v, v', v^{(2)}\}
\end{tikzcd}$$

$$\{u^{(5)} \simeq^?_{\mathcal{R},\lambda} u^{(4)},\ w^{(2)} \simeq^?_{\mathcal{R},\lambda} \mathrm{N}_3(v'),\ u^{(3)} \simeq^?_{\mathcal{R},\lambda} u',\ u^{(4)} \simeq^?_{\mathcal{R},\lambda} u^{(2)},\ v^{(2)} \simeq^?_{\mathcal{R},\lambda} v',$$
$$u' \simeq^?_{\mathcal{R},\lambda} u,\ u \simeq^?_{\mathcal{R},\lambda} u^{(2)},\ u^{(4)} \simeq^?_{\mathcal{R},\lambda} u^{(2)},\ v' \simeq^?_{\mathcal{R},\lambda} v,\ u' \simeq^?_{\mathcal{R},\lambda} v\}$$

The edges have not changed, the serialization decreased (due to the mark at V_5), and the decomposition further decreased the size.

In the final step, we apply **(VE)** to $w^{(2)} \simeq^?_{\mathcal{R},\lambda} \mathrm{N}_3(v')$ and then decompose the result, which gives the final graph

$$\begin{tikzcd}
 & & V_3 \ \varnothing \arrow[d] & \\
\underset{\varnothing}{V_1} \arrow[r] & \underset{\varnothing}{V_2} \arrow[ur] \arrow[r] \arrow[dr] & V_4 \ \{u, u', u^{(2)}, u^{(3)}, u^{(4)}, u^{(5)}\} & \\
 & & \underset{\varnothing}{V_5} \arrow[r] & V_6 \ \{v, v', v^{(2)}, v^{(3)}\}
\end{tikzcd}$$

$$\{u^{(5)} \simeq^?_{\mathcal{R},\lambda} u^{(4)},\ v^{(3)} \simeq^?_{\mathcal{R},\lambda} v',\ u^{(3)} \simeq^?_{\mathcal{R},\lambda} u',\ u^{(4)} \simeq^?_{\mathcal{R},\lambda} u^{(2)},\ v^{(2)} \simeq^?_{\mathcal{R},\lambda} v',$$
$$u' \simeq^?_{\mathcal{R},\lambda} u,\ u \simeq^?_{\mathcal{R},\lambda} u^{(2)},\ u^{(4)} \simeq^?_{\mathcal{R},\lambda} u^{(2)},\ v' \simeq^?_{\mathcal{R},\lambda} v,\ u' \simeq^?_{\mathcal{R},\lambda} v\}$$

Also here: the edges have not changed, the serialization decreased (the mark of V_5), and the decomposition further decreased the size.

The derivation stops, because the variables-only constraint is reached.

With the termination proof done, we devote the rest of the section to soundness and completeness properties of pre-unification.

We say that a mapping-substitution pair $(\Phi, \boldsymbol{\nu})$ is a *solution of an equational* $(\mathcal{R}, \lambda)$*-configuration* $P; C; \boldsymbol{\mu}$ if the following conditions hold:

- $(\Phi, \boldsymbol{\nu})$ is an $(\mathcal{R}, \lambda)$-solution of P;
- Φ is an $(\mathcal{R}, \lambda)$-solution of C;
- For each $x \in dom(\boldsymbol{\mu})$, we have $x\boldsymbol{\nu} = x\boldsymbol{\mu}\boldsymbol{\nu}$ (syntactic equality).

Lemma 4.2.2. *1. If $P; C; \boldsymbol{\mu} \Longrightarrow \bot$ by* (Cla) *or* (Occ) *rules, then $P; C; \boldsymbol{\mu}$ does not have a solution.*

2. Let $P_1; C_1; \boldsymbol{\mu}_1 \Longrightarrow P_2; C_2; \boldsymbol{\mu}_2$ be a step performed by a pre-unification rule (except (Cla) *and* (Occ)*). Then every solution of $P_2; C_2; \boldsymbol{\mu}_2$ is a solution of $P_1; C_1; \boldsymbol{\mu}_1$.*

Proof. 1. If (Cla) applies, we have an arity mismatch in an equation in P, which cannot be repaired by substitutions: P is not unifiable. For the case of (Occ), the lemma follows from Lemma 4.2.1.

2. We shall prove the lemma for each non-failing rule. The nontrivial cases are (Dec) and (VE) rules.

(Dec): $P_1 = \{\mathbf{F}(\mathbf{s}_1, \ldots, \mathbf{s}_n) \simeq^?_{\mathcal{R},\lambda} \mathbf{G}(\mathbf{t}_1, \ldots, \mathbf{t}_n)\} \uplus P$, $P_2 = \{\mathbf{s}_1 \simeq^?_{\mathcal{R},\lambda} \mathbf{t}_1, \ldots, \mathbf{s}_n \simeq^?_{\mathcal{R},\lambda} \mathbf{t}_n\} \cup P$, $C_2 = C_1 \cup \{\mathbf{F} \approx^?_{\mathcal{R},\lambda} \mathbf{G}\}$, $\boldsymbol{\mu}_1 = \boldsymbol{\mu}_2$. Hence, the information about $\mathbf{F}$ and $\mathbf{G}$ to be $(\mathcal{R}, \lambda)$-close to each other just moves from P_1 to C_2. The rest does not change. Therefore, $P_1; C_1; \boldsymbol{\mu}_1$ and $P_2; C_2; \boldsymbol{\mu}_2$ have the same set of solutions.

(VE): Let $\boldsymbol{\xi} = \{x \mapsto \mathbf{t}'\}$. Then $P_1 = \{x \simeq^?_{\mathcal{R},\lambda} \mathbf{t}\} \uplus P$, $P_2 = \{\mathbf{t}' \simeq^?_{\mathcal{R},\lambda} \mathbf{t}\} \cup P\boldsymbol{\xi}$, $C_1 = C_2$, and $\boldsymbol{\mu}_2 = \boldsymbol{\mu}_1\boldsymbol{\xi}$.
Note that $\boldsymbol{\mu}_2$ contains $x \mapsto \mathbf{t}'$, because $x \notin dom(\boldsymbol{\mu}_1)$ by the rules. Let $(\Phi, \boldsymbol{\nu})$ be a solution of $P_2; C_2; \boldsymbol{\mu}_2$. Then we have $\Phi(x\boldsymbol{\nu}) = \Phi(\mathbf{t}'\boldsymbol{\nu})$. We also have $\Phi(\mathbf{t}'\boldsymbol{\nu}) \simeq_{\mathcal{R},\lambda} \Phi(\mathbf{t}\boldsymbol{\nu})$, since $\boldsymbol{\nu}$ solves $\mathbf{t}' \simeq^?_{\mathcal{R},\lambda} \mathbf{t}$. Hence, $\Phi(x\boldsymbol{\nu}) \simeq_{\mathcal{R},\lambda} \Phi(\mathbf{t}\boldsymbol{\nu})$ and we get that $(\Phi, \boldsymbol{\nu})$ solves the equation $x \simeq^?_{\mathcal{R},\lambda} \mathbf{t}$. For an equation $eq \in P$, we have $eq\boldsymbol{\nu} = eq\boldsymbol{\xi}\boldsymbol{\nu}$, because $x\boldsymbol{\nu} = \mathbf{t}'\boldsymbol{\nu} = x\boldsymbol{\xi}\boldsymbol{\nu}$, and for any $y \neq x$, it holds trivially that $y\boldsymbol{\nu} = y\boldsymbol{\xi}\boldsymbol{\nu}$. Hence, $(\Phi, \boldsymbol{\nu})$ solves P as well. For a $y \in dom(\boldsymbol{\mu}_1)$, we have $y\boldsymbol{\nu} = y\boldsymbol{\mu}_2\boldsymbol{\nu} = y\boldsymbol{\mu}_1\boldsymbol{\xi}\boldsymbol{\nu} = y\boldsymbol{\mu}_1\boldsymbol{\nu}$. Therefore, $(\Phi, \boldsymbol{\nu})$ is a solution of $P_1; C_1; \boldsymbol{\mu}_1$.

□

Theorem 4.2.2 (Soundness of pre-unification)**.** *Let s and t be two terms, such that the pre-unification algorithm gives $\{s \simeq^?_{\mathcal{R},\lambda} t\}; \varnothing; Id \Longrightarrow^* V; C; \boldsymbol{\mu}$, where V is a variable-only constraint. Let $(\Phi, \boldsymbol{\nu})$ be an $(\mathcal{R}, \lambda)$-solution of $V; C; \boldsymbol{\mu}$. Then the X-substitution $\Phi(\boldsymbol{\mu\nu})$ is an $(\mathcal{R}, \lambda)$-X-unifier of $\{s \simeq^?_{\mathcal{R},\lambda} t\}$.*

Proof. First, we show that $(\Phi, \boldsymbol{\mu\nu})$ is an $(\mathcal{R}, \lambda)$-solution of $V; C; \boldsymbol{\mu}$. By the construction of $\boldsymbol{\mu}$, we have (i) $dom(\boldsymbol{\mu}) \cap \mathcal{V}(V) = \varnothing$ and (ii) $\boldsymbol{\mu}$ is idempotent. Then by (i) we have that $(\Phi, \boldsymbol{\mu\nu})$ solves V, because $V\boldsymbol{\mu} = V$. By (ii) we have $x\boldsymbol{\mu\nu} = x\boldsymbol{\mu\mu\nu}$. By definition, these imply that $(\Phi, \boldsymbol{\mu\nu})$ is an $(\mathcal{R}, \lambda)$-solution of $V; C; \boldsymbol{\mu}$.

Next, from $\{s \simeq^?_{\mathcal{R},\lambda} t\}; \varnothing; Id \Longrightarrow^* V; C; \boldsymbol{\mu}$, by induction on the length of the derivation, using Lemma 4.2.2, we get that $(\Phi, \boldsymbol{\mu\nu})$ is an $(\mathcal{R}, \lambda)$-solution of $\{s \simeq^?_{\mathcal{R},\lambda} t\}; \varnothing; Id$.

Since s and t do not contain names, Φ has no effect on them: $\Phi(s\boldsymbol{\mu\nu}) = s\Phi(\boldsymbol{\mu\nu})$ and $\Phi(t\boldsymbol{\mu\nu}) = t\Phi(\boldsymbol{\mu\nu})$. From these equalities and $\Phi(s\boldsymbol{\mu\nu}) \simeq_{\mathcal{R},\lambda} \Phi(t\boldsymbol{\mu\nu})$ we get $s\Phi(\boldsymbol{\mu\nu}) \simeq_{\mathcal{R},\lambda} t\Phi(\boldsymbol{\mu\nu})$, which implies that $\Phi(\boldsymbol{\mu\nu})$ is an $(\mathcal{R}, \lambda)$-X-solution of $\{s \simeq^?_{\mathcal{R},\lambda} t\}$. □

Corollary 4.2.2.1. *Let s and t be two terms, such that the pre-unification algorithm gives $\{s \simeq^?_{\mathcal{R},\lambda} t\}; \varnothing; Id \Longrightarrow^* V; C; \boldsymbol{\mu}$. Let Φ be an $(\mathcal{R}, \lambda)$-solution of C. Let $(\Phi, \boldsymbol{\nu})$ be an $(\mathcal{R}, \lambda)$-solution of $V; C; \boldsymbol{\mu}$ such that $\boldsymbol{\nu}$ is name-free. Then every substitution in* $\mathsf{SX\text{-}substs}(\Phi(\boldsymbol{\mu\nu}))$ *is an $(\mathcal{R}, \lambda)$-unifier of s and t.*

Proof. It is a consequence of Theorem 4.2.2 and the definition of $\mathsf{SX\text{-}substs}$. Note that due to the way how the C's are constructed in pre-unification derivations, any solution of C would map all the names in C to sets of function symbols. This happens because for any name introduced by the **VE** rule, there is a connection via $\approx$ equations to a function symbol. Hence, all names that occur in the range of $\boldsymbol{\mu}$ (they all appear in C as well) are mapped by Φ to a non-name neighborhood. Further, since $\boldsymbol{\nu}$ is name-free, $\mathsf{SX\text{-}substs}(\Phi(\boldsymbol{\mu\nu}))$ is a set of substitutions, not a set of SX-substitutions, because $\Phi(\boldsymbol{\mu\nu})$ does not contain names. □

The completeness theorem below is a pretty strong statement. It says that for solvable proximity-based unification problems, the pre-unification algorithm computes an X-substitution, from which we can obtain any unifier of the given problem with the help of solutions of the variables-only and neighborhood constraints computed together with the substitution. Note the syntactic equality in the theorem, implying that any unifier can be obtained from the computed configuration exactly (in contrast to getting close to it).

Theorem 4.2.3 (Completeness of pre-unification)**.** *Let a substitution σ be an $(\mathcal{R},\lambda)$-unifier of two terms s and t and Var be the set of variables $\mathcal{V}(s) \cup \mathcal{V}(t)$. Then any maximal derivation that starts at $\{s \simeq^?_{\mathcal{R},\lambda} t\}; \emptyset; Id$ must end with an equational configuration $V; C; \boldsymbol{\mu}$ where V is a variables-only constraint, such that for some $(\mathcal{R},\lambda)$-solution Φ of C and for some $(\mathcal{R},\lambda)$-solution ν of V, we have $(\mu\nu)|_{Var} = \sigma|_{Var}$, where $\mu \in \mathsf{substs}(\Phi(\boldsymbol{\mu}))$.*

Proof. Note that since $\Phi(\boldsymbol{\mu})$ is name-free, $\mathsf{substs}(\Phi(\boldsymbol{\mu}))$ is defined.

By the assumption, $\{s \simeq^?_{\mathcal{R},\lambda} t\}$ is solvable, therefore, the derivation cannot end with $\bot$ by soundness of pre-unification. Since any equation, except variables-only ones, can be transformed by a rule, and the algorithm terminates, the final configuration should have a form $V; C; \boldsymbol{\mu}$, where V is a variables-only constraint.

We assume without loss of generality that σ is idempotent and show now how to construct the desired Φ step by step, using the following proposition:

Proposition: Assume $P_1; C_1; \boldsymbol{\mu}_1 \Longrightarrow P_2; C_2; \boldsymbol{\mu}_2$ is a single step rule application in the above mentioned derivation. Let $(\Phi_1, \boldsymbol{\vartheta}_1)$ be an idempotent $(\mathcal{R},\lambda)$-solution of $P_1; C_1; \boldsymbol{\mu}_1$ such that $\sigma|_{Var} = \vartheta_1|_{Var}$ for some substitution $\vartheta_1 \in \mathsf{substs}(\Phi_1(\boldsymbol{\vartheta}_1))$. Then there exists an idempotent $(\mathcal{R},\lambda)$-solution $(\Phi_2, \boldsymbol{\vartheta}_2)$ of $P_2; C_2; \boldsymbol{\mu}_2$ such that $\sigma|_{Var} = \vartheta_2|_{Var}$ for some $\vartheta_2 \in \mathsf{substs}(\Phi_2(\boldsymbol{\vartheta}_2))$.

Proof of the proposition: We assume that $dom(\sigma) \subseteq \mathcal{V}(s) \cup \mathcal{V}(t) = Var$.

For technical reasons, we assume that substitutions and X-substitutions are represented in triangular form [8] as a sequential list of bindings, e.g., $[x_1 \mapsto \mathbf{t}_1; \dots; x_n \mapsto \mathbf{t}_n]$, which represents the composition of n substitutions each consisting of a single binding: $\{x_1 \mapsto \mathbf{t}_1\}\{x_2 \mapsto \mathbf{t}_2\} \cdots \{x_n \mapsto \mathbf{t}_n\}$. The constructor $[\cdot]$ is assumed to be flat.

Application of a neighborhood mapping to triangular X-substitutions is defined as $\Phi([\boldsymbol{\rho}_1; \dots; \boldsymbol{\rho}_n]) = [\Phi(\boldsymbol{\rho}_1); \dots; \Phi(\boldsymbol{\rho}_1)]$. Given a triangular X-substitution $\boldsymbol{\tau} = [\boldsymbol{\rho}_1; \dots; \boldsymbol{\rho}_i; \dots; \boldsymbol{\rho}_n]$, by $\mathsf{replace}(\boldsymbol{\tau}, \boldsymbol{\rho}_i \rightsquigarrow \boldsymbol{\rho}')$ we understand the triangular X-substitution $[\boldsymbol{\rho}_1; \dots; \boldsymbol{\rho}'; \dots; \boldsymbol{\rho}_n]$.

We consider each of the rules separately.

(Tri): In this case we take $\Phi_2 = \Phi_1$ and $\boldsymbol{\vartheta}_2 = \boldsymbol{\vartheta}_1$.

(Dec): We have

- $P_1 = \{\mathbf{F}(\mathbf{t}_1, \ldots, \mathbf{t}_n) \simeq^?_{\mathcal{R},\lambda} \mathbf{G}(\mathbf{s}_1, \ldots, \mathbf{s}_n)\} \uplus P$,
- $P_2 = \{\mathbf{t}_1 \simeq^?_{\mathcal{R},\lambda} \mathbf{s}_1, \ldots, \mathbf{t}_n \simeq^?_{\mathcal{R},\lambda} \mathbf{s}_n\} \cup P$,
- $C_2 = \{\mathbf{F} \approx^?_{\mathcal{R},\lambda} \mathbf{G}\} \cup C_1$,
- $\boldsymbol{\mu}_2 = \boldsymbol{\mu}_1$.

Then, since $(\Phi_1, \boldsymbol{\vartheta}_1)$ is an $(\mathcal{R}, \lambda)$-solution of $P_1; C_1; \boldsymbol{\mu}_1$, it is also an $(\mathcal{R}, \lambda)$-solution of the configuration $P_2; C_2; \boldsymbol{\mu}_2$. So, we take $\Phi_2 = \Phi_1$ and $\boldsymbol{\vartheta}_2 = \boldsymbol{\vartheta}_1$.

(VE): We have

- $P_1 = \{x \simeq^?_{\mathcal{R},\lambda} \mathbf{s}\} \uplus P$ where there is no occurrence cycle for x and $\mathbf{s} \notin \mathcal{V}$,
- $P_2 = \{\mathbf{s}' \simeq^?_{\mathcal{R},\lambda} \mathbf{s}\} \cup P\{x \mapsto \mathbf{s}'\}$, where $\mathbf{s}' = \rho(\mathbf{s})$,
- $C_2 = C_1$.
- $\boldsymbol{\mu}_2 = [\boldsymbol{\mu}_1; x \mapsto \mathbf{s}']$.

Since the pair $(\Phi_1, \boldsymbol{\vartheta}_1)$ is an $(\mathcal{R}, \lambda)$-solution of P_1, we have $\Phi_1(x\boldsymbol{\vartheta}_1) \simeq_{\mathcal{R},\lambda} \Phi_1(\mathbf{s}\boldsymbol{\vartheta}_1)$.

To construct Φ_2 and $\boldsymbol{\vartheta}_2$, we start from Φ_1 and $\boldsymbol{\vartheta}_1$ and modify them. Since $x \simeq^?_{\mathcal{R},\lambda} \mathbf{s} \in P_1$ with $\mathbf{s} \notin \mathcal{V}$ and $(\Phi_1, \boldsymbol{\vartheta}_1)$ is a solution of P_1, we know that $x \in \mathit{dom}(\boldsymbol{\vartheta}_1)$. Consider $\mathit{pos}(\mathbf{s}')$. Each object in positions from $\mathit{pos}(\mathbf{s}')$ is either name or a variable. Let $\mathbf{r} = \Phi_1(x\boldsymbol{\vartheta}_1)$. Then $\mathit{pos}(\mathbf{s}') \subseteq \mathit{pos}(\mathbf{r})$, because $\mathbf{s}'$ is the minimal skeleton a term unifiable to $\mathbf{s}$ can have. Let $\mathit{pos}_{\mathcal{N}}(\mathbf{s}')$ (resp. $\mathit{pos}_{\mathcal{V}}(\mathbf{s}')$) be the set of positions in $\mathbf{s}'$ where names (resp. variables) occur.

Let Φ' and $\boldsymbol{\vartheta}'$ be defined as

$$\begin{aligned}
\Phi' &:= \{\mathrm{N} \mapsto \mathbf{F} \mid \mathrm{N} = \mathbf{s}'|_p,\ \mathbf{F} = \mathbf{r}|_p,\ p \in \mathit{pos}_{\mathcal{N}}(\mathbf{s}')\},\\
\boldsymbol{\vartheta}' &:= [y_1 \mapsto \mathbf{t}_1; \ldots; y_k \mapsto \mathbf{t}_k],\ y_i = \mathbf{s}'|_{p_i},\ \mathbf{t}_i = \mathbf{r}|_{p_i},\\
&\quad\ \{p_1, \ldots, p_k\} = \mathit{pos}_{\mathcal{V}}(\mathbf{s}').
\end{aligned}$$

Define Φ_2 and $\boldsymbol{\vartheta}_2$ as

$$\Phi_2 := \Phi_1 \cup \Phi',$$
$$\boldsymbol{\vartheta}_2 := [\mathsf{replace}(\boldsymbol{\vartheta}_1, \{x \mapsto x\boldsymbol{\vartheta}_1\} \rightsquigarrow \{x \mapsto \mathsf{s}'\}); \boldsymbol{\vartheta}'].$$

Φ_2 is a well-defined name-neighborhood mapping, since both Φ_1 and Φ' are name-neighborhood mappings with $dom(\Phi_1) \cap dom(\Phi') = \emptyset$. Note that $\boldsymbol{\vartheta}_2$ is an idempotent substitution, because $\boldsymbol{\vartheta}_1$ is idempotent by assumption, and for the same reason variables in the range of $\boldsymbol{\vartheta}'$ (i.e., those from $\mathbf{r}$) do not occur in the domain of $\boldsymbol{\vartheta}_1$.

We have to show that

- $(\Phi_2, \boldsymbol{\vartheta}_2)$ is a solution of P_2,
- Φ_2 is a solution of C_2,
- $y\boldsymbol{\vartheta}_2 = y\boldsymbol{\mu}_2\boldsymbol{\vartheta}_2$ for all $y \in dom(\boldsymbol{\mu}_2)$,
- $\vartheta_2|_{Var} = \sigma|_{Var}$ for some $\vartheta_2 \in \mathsf{substs}(\Phi_2(\boldsymbol{\vartheta}_2))$.

Proving $(\Phi_2, \boldsymbol{\vartheta}_2)$ is a solution of P_2. From the definition of Φ_2 and $\boldsymbol{\vartheta}_2$ it is clear $\Phi_2(y\boldsymbol{\vartheta}_2) = \Phi_1(y\boldsymbol{\vartheta}_1)$ for all $y \in dom(\boldsymbol{\vartheta}_1)\backslash\{x\}$. As for x, we have $\Phi_2(x\boldsymbol{\vartheta}_2) = \mathbf{r}$, from s', $\mathbf{r}$ and the definitions of Φ' and $\boldsymbol{\vartheta}'$. But at the same time we have $\mathbf{r} = \Phi_1(x\boldsymbol{\vartheta}_1)$. Hence, also for x we have $\Phi_2(x\boldsymbol{\vartheta}_2) = \Phi_1(x\boldsymbol{\vartheta}_1)$. Therefore, we have

$$\Phi_2(y\boldsymbol{\vartheta}_2) = \Phi_1(y\boldsymbol{\vartheta}_1) \text{ for all } y \in dom(\boldsymbol{\vartheta}_1). \tag{4.1}$$

As for those variables that belong to $dom(\boldsymbol{\vartheta}_2)\backslash dom(\boldsymbol{\vartheta}_1)$, they are exactly $\{y_1, \dots, y_n\}$, which form $dom(\boldsymbol{\vartheta}')$. In P_2, they appear only there, where s' has been introduced, i.e., where in P_1 the variable x was located. By definition of $\boldsymbol{\vartheta}_2$, we have $\Phi_2(\mathsf{s}'\boldsymbol{\vartheta}_2) = \mathbf{r}$. But since $\mathbf{r} = \Phi_1(x\boldsymbol{\vartheta}_1)$, we get

$$\Phi_2(\mathsf{s}'\boldsymbol{\vartheta}_2) = \Phi_1(x\boldsymbol{\vartheta}_1). \tag{4.2}$$

The set P_2 differs from P_1 by replacing x by s'. Since $(\Phi_1, \boldsymbol{\vartheta}_1)$ solves P_1, from (4.1) and (4.2) we obtain that $(\Phi_2, \boldsymbol{\vartheta}_2)$ solves P_2.

Proving Φ_2 is a solution of C_2. Trivial, since $C_2 = C_1$, Φ_1 solves C_1, and $\Phi_2 = \Phi_1 \cup \Phi'$.

Proving $y\boldsymbol{\vartheta}_2 = y\boldsymbol{\mu}_2\boldsymbol{\vartheta}_2$ for all $y \in dom(\boldsymbol{\mu}_2)$. Take $y \in dom(\boldsymbol{\mu}_2)$. First, assume it is not x. Then we have:

$$\begin{aligned}
y\boldsymbol{\vartheta}_2 &= y[\mathsf{replace}(\boldsymbol{\vartheta}_1, \{x \mapsto x\boldsymbol{\vartheta}_1\} \rightsquigarrow \{x \mapsto \mathsf{s}'\}); \boldsymbol{\vartheta}']\\
&\qquad \text{(Since } \boldsymbol{\vartheta}' \text{ affects only the new variables occurring in } \mathsf{s}')\\
&= y[\mathsf{replace}(\boldsymbol{\vartheta}_1, \{x \mapsto x\boldsymbol{\vartheta}_1\} \rightsquigarrow \{x \mapsto \mathsf{s}'\boldsymbol{\vartheta}'\})]. \qquad (4.3)
\end{aligned}$$

$y[\mathsf{replace}(\boldsymbol{\vartheta}_1, \{x \mapsto x\boldsymbol{\vartheta}_1\} \rightsquigarrow \{x \mapsto \mathsf{s}'\boldsymbol{\vartheta}'\})]$ differs from $y\boldsymbol{\vartheta}_1$ only at the places where function symbols occurring in $x\boldsymbol{\vartheta}_1$ have been uniquely replaced by new names in $\mathsf{s}'\boldsymbol{\vartheta}'$.

$$\begin{aligned}
y\boldsymbol{\mu}_2\boldsymbol{\vartheta}_2 &= y\boldsymbol{\mu}_1\{x \mapsto \mathsf{s}'\}[\mathsf{replace}(\boldsymbol{\vartheta}_1, \{x \mapsto x\boldsymbol{\vartheta}_1\} \rightsquigarrow \{x \mapsto \mathsf{s}'\}); \boldsymbol{\vartheta}']\\
&\qquad \text{(By definition of substitutions in the triangular form)}\\
&= y\boldsymbol{\mu}_1\{x \mapsto \mathsf{s}'\}\mathsf{replace}(\boldsymbol{\vartheta}_1, \{x \mapsto x\boldsymbol{\vartheta}_1\} \rightsquigarrow \{x \mapsto \mathsf{s}'\})\boldsymbol{\vartheta}'\\
&\qquad \text{(Since } \boldsymbol{\vartheta}' \text{ affects only the new variables occurring in } \mathsf{s}')\\
&= y\boldsymbol{\mu}_1[\mathsf{replace}(\boldsymbol{\vartheta}_1, \{x \mapsto x\boldsymbol{\vartheta}_1\} \rightsquigarrow \{x \mapsto \mathsf{s}'\boldsymbol{\vartheta}'\})]. \qquad (4.4)
\end{aligned}$$

Also here, the difference between $y\boldsymbol{\mu}_1[\mathsf{replace}(\boldsymbol{\vartheta}_1, \{x \mapsto x\boldsymbol{\vartheta}_1\} \rightsquigarrow \{x \mapsto \mathsf{s}'\boldsymbol{\vartheta}'\})]$ and $y\boldsymbol{\mu}_1\boldsymbol{\vartheta}_1$ is only at the places where function symbols occurring in $x\boldsymbol{\vartheta}_1$ have been uniquely replaced by new names in $\mathsf{s}'\boldsymbol{\vartheta}'$. Therefore, since $y\boldsymbol{\vartheta}_1 = y\boldsymbol{\mu}_1\boldsymbol{\vartheta}_1$, from (4.3) and (4.4) we get $y\boldsymbol{\vartheta}_2 = y\boldsymbol{\mu}_2\boldsymbol{\vartheta}_2$.

Now assume $y = x$. Then we have

$$\begin{aligned}
x\boldsymbol{\vartheta}_2 &= x[\mathsf{replace}(\boldsymbol{\vartheta}_1, \{x \mapsto x\boldsymbol{\vartheta}_1\} \rightsquigarrow \{x \mapsto \mathsf{s}'\}); \boldsymbol{\vartheta}']\\
&= x[\mathsf{replace}(\boldsymbol{\vartheta}_1, \{x \mapsto x\boldsymbol{\vartheta}_1\} \rightsquigarrow \{x \mapsto \mathsf{s}'\boldsymbol{\vartheta}'\})]\\
&= \mathsf{s}'\boldsymbol{\vartheta}'.
\end{aligned}$$

$$\begin{aligned}
x\boldsymbol{\mu}_2\boldsymbol{\vartheta}_2 &= x\{x \mapsto \mathsf{s}'\}[\mathsf{replace}(\boldsymbol{\vartheta}_1, \{x \mapsto x\boldsymbol{\vartheta}_1\} \rightsquigarrow \{x \mapsto \mathsf{s}'\}); \boldsymbol{\vartheta}']\\
&= x\{x \mapsto \mathsf{s}'\boldsymbol{\vartheta}'\}\mathsf{replace}(\boldsymbol{\vartheta}_1, \{x \mapsto x\boldsymbol{\vartheta}_1\} \rightsquigarrow \{x \mapsto \mathsf{s}'\boldsymbol{\vartheta}'\})\\
&\qquad \text{(By the idempotence of } \boldsymbol{\vartheta}_1 \text{ and the definition of } \boldsymbol{\vartheta}',\\
&\qquad dom(\boldsymbol{\vartheta}_1) \cap \mathcal{V}(\mathsf{s}'\boldsymbol{\vartheta}') = \varnothing)\\
&= \mathsf{s}'\boldsymbol{\vartheta}'.
\end{aligned}$$

Hence, also for x we have $x\boldsymbol{\vartheta}_2 = x\boldsymbol{\mu}_2\boldsymbol{\vartheta}_2$, which finished the proof of this part.

Proving $\vartheta_2|_{Var} = \sigma|_{Var}$ for some $\vartheta_2 \in \mathsf{substs}(\Phi_2(\boldsymbol{\vartheta}_2))$. We have already seen that $\Phi_2(y\boldsymbol{\vartheta}_2) = \Phi_1(y\boldsymbol{\vartheta}_1)$ for all $y \in dom(\boldsymbol{\vartheta}_1)$. No variable from $dom(\boldsymbol{\vartheta}')$ belongs to Var, because they have been freshly introduced at the current step. Hence, $\vartheta_2|_{Var} = \vartheta_1|_{Var} = \sigma|_{Var}$.

(Ori): Straightforward.

Hence, the proposition is proved.

In the constructed derivation, the initial configuration is $P_0; C_0; \boldsymbol{\mu}_0 = \{s \simeq^?_{\mathcal{R},\lambda} t\}; \varnothing; Id$. We take $\Phi_0 = \varnothing$ and $\boldsymbol{\vartheta}_0$ as the SX-version of σ. Then $(\Phi_0, \boldsymbol{\vartheta}_0)$ solves P_0, Φ_0 solves $C_0 = \varnothing$, and for all x, we have $x\boldsymbol{\mu}_0\boldsymbol{\vartheta}_0 = xId\boldsymbol{\vartheta}_0 = x\boldsymbol{\vartheta}_0$. Hence, $(\Phi_0, \boldsymbol{\vartheta}_0)$ is an $(\mathcal{R}, \lambda)$-solution of $P_0; C_0; \boldsymbol{\mu}_0$. Moreover, we can take ϑ_0 to be σ. Then we have $\sigma|_{Var} = \vartheta_0|_{Var}$ with $\vartheta_0 \in \Phi_0(\boldsymbol{\vartheta}_0)$.

Hence, the conditions of the proposition are satisfied. Then for the final configuration $P_n; C_n; \boldsymbol{\mu}_n = V; C_n; \boldsymbol{\mu}_n$, by the n-fold application of the proposition we can find a name-neighborhood mapping Φ_n and an (idempotent) X-substitution $\boldsymbol{\vartheta}_n$ so that

- $(\Phi_n, \boldsymbol{\vartheta}_n)$ solves $V; C_n; \boldsymbol{\mu}_n$,
- Φ_n solves C_n,
- for all $x \in dom(\boldsymbol{\vartheta}_n)$, $x\boldsymbol{\vartheta}_n = x\boldsymbol{\mu}_n\boldsymbol{\vartheta}_n$, and
- there exists $\vartheta_n \in \mathsf{substs}(\Phi_n(\boldsymbol{\vartheta}_n))$ such that $\vartheta_n|_{Var} = \sigma|_{Var}$.

By the construction of pre-unification derivations, variables in the range of $\boldsymbol{\mu}_n$ appear in V. Therefore, $\boldsymbol{\vartheta}_n$ can be represented as the composition $\boldsymbol{\mu}_n\boldsymbol{\nu}$, where $\boldsymbol{\nu}$ is a solution of V. Note that $dom(\boldsymbol{\nu})$ may contain a variable that does not appear in V. This may happen if the variable has been eliminated from the derivation by the trivial rule (Tri), but it is present in $\boldsymbol{\nu}$ since we started from σ, which may contain the variable in its domain.

Hence, we get $\mathsf{substs}(\Phi_n(\boldsymbol{\vartheta}_n)) = \mathsf{substs}(\Phi_n(\boldsymbol{\mu}_n\boldsymbol{\nu}))$ and it contains a substitution $\vartheta_n = \mu\nu$, where $\mu \in \mathsf{substs}(\Phi_n(\boldsymbol{\mu}_n))$ and $\nu \in \mathsf{substs}(\Phi_n(\boldsymbol{\nu}))$. (the latter implies that ν is an $(\mathcal{R}, \lambda)$-solution of V.) Thus $(\mu\nu)|_{Var} = \sigma|_{Var}$, and the proof is finished. □

Of course, we can always get rid of the variables-only constraint and eliminate variables by the (VO) rule, but it has its pros and cons:

(VO) Variables Only:

$$\{x \simeq^?_{\mathcal{R},\lambda} y,\ \overline{x_n \simeq^?_{\mathcal{R},\lambda} y_n}\}; C; \boldsymbol{\mu} \Longrightarrow \{\overline{x_n \simeq^?_{\mathcal{R},\lambda} y_n}\}\{x \mapsto y\}; C; \boldsymbol{\mu}\{x \mapsto y\}.$$

The advantage of permitting the (VO) rule is that we get a definite solution, a substitution, instead of a substitution-constraint pair. The disadvantage is that we lose completeness which is to see with the following example: If $(a, b) \in \mathcal{R}_\lambda$, then $\boldsymbol{\sigma} = \{x \mapsto a, y \mapsto b\}$ is an $(\mathcal{R}, \lambda)$-unifier of $x \simeq^?_{\mathcal{R},\lambda} y$, but if we solve the latter by $\boldsymbol{\mu} = \{x \mapsto y\}$, then we are not able to obtain $\boldsymbol{\sigma}$ from this solution, since $\boldsymbol{\mu}$ is not more general than $\boldsymbol{\sigma}$.

We introduce a neighborhood constraint solving algorithm in the next subsection. Before that we illustrate the pre-unification rules with a couple of examples:

Example 4.2.2. Let $s = p(x, y, x)$ and $t = q(f(a), g(d), y)$. Then the pre-unification algorithm gives $\emptyset; C, \boldsymbol{\mu}$, where $C = \{p \approx^?_{\mathcal{R},\lambda} q,\ \mathrm{N}_1 \approx^?_{\mathcal{R},\lambda} f,\ \mathrm{N}_2 \approx^?_{\mathcal{R},\lambda} a,\ \mathrm{N}_3 \approx^?_{\mathcal{R},\lambda} g,\ \mathrm{N}_4 \approx^?_{\mathcal{R},\lambda} d,\ \mathrm{N}_1 \approx^?_{\mathcal{R},\lambda} \mathrm{N}_3,\ \mathrm{N}_2 \approx^?_{\mathcal{R},\lambda} \mathrm{N}_4\}$ and $\boldsymbol{\mu} = \{x \mapsto \mathrm{N}_1(\mathrm{N}_2), y \mapsto \mathrm{N}_3(\mathrm{N}_4)\}$.

Assume that for the given λ-cut, the proximity relation consists of pairs $\mathcal{R}_\lambda = \{(a, b), (b, c), (c, d), (a, b'), (b', c'), (c', d), (f, g), (p, q)\}$. The constraint C obtained above can be solved, e.g., by the name-neighborhood mappings $\Phi = [\mathrm{N}_1 \mapsto \{f, g\},\ \mathrm{N}_2 \mapsto \{b\}, \mathrm{N}_3 \mapsto \{f, g\},\ \mathrm{N}_4 \mapsto \{c\}]$ and $\Phi' = [\mathrm{N}_1 \mapsto \{f, g\},\ \mathrm{N}_2 \mapsto \{b'\}, \mathrm{N}_3 \mapsto \{f, g\},\ \mathrm{N}_4 \mapsto \{c'\}]$. From them and $\boldsymbol{\mu}$ we can get the sets $\Phi(\boldsymbol{\mu})$ and $\Phi'(\boldsymbol{\mu})$ of $(\mathcal{R}, \lambda)$-unifiers of s and t.

Example 4.2.3. Let $s = p(x, x)$ and $t = q(f(y, y), f(a, c))$. The pre-unification algorithm stops with $\emptyset; C; \boldsymbol{\mu}$, where $C = \{p \approx^?_{\mathcal{R},\lambda} q,\ \mathrm{N}_1 \approx^?_{\mathcal{R},\lambda} f,\ \mathrm{N}_2 \approx^?_{\mathcal{R},\lambda} a,\ \mathrm{N}_3 \approx^?_{\mathcal{R},\lambda} c,\ \mathrm{M} \approx^?_{\mathcal{R},\lambda} \mathrm{N}_2,\ \mathrm{N}_3 \approx^?_{\mathcal{R},\lambda} \mathrm{M}\}$ and $\boldsymbol{\mu} = \{x \mapsto \mathrm{N}_1(\mathrm{N}_2, \mathrm{N}_3),\ y_1 \mapsto \mathrm{N}_2,\ y_2 \mapsto \mathrm{N}_3,\ y \mapsto \mathrm{M}\}$. Let $\mathcal{R}_\lambda = \{(a, a_1), (a_1, b), (b, c_1), (c_1, c), (p, q)\}$. Then C is solved by $\Phi = [\mathrm{N}_1 \mapsto \{f\},\ \mathrm{N}_2 \mapsto \{a_1\},\ \mathrm{M} \mapsto \{b\},\ \mathrm{N}_3 \mapsto \{c_1\}]$ and $\Phi(\boldsymbol{\mu}|_{\mathcal{V}(s)\cup\mathcal{V}(t)})$ contains only one element, an $(\mathcal{R}, \lambda)$-unifier $\sigma = \{x \mapsto f(a_1, c_1),\ y \mapsto b\}$ of s and t. Indeed, $s\sigma = p(f(a_1, c_1), f(a_1, c_1)) \simeq_{\mathcal{R},\lambda} q(f(b, b), f(a, c)) = t\sigma$.

This example illustrates the necessity of introducing a fresh variable for *each occurrence* of a variable by the renaming function in the VE rule. If we used the same new variable, say y', for both occurrences of y in $f(y, y)$ (instead of using y_1 and y_2 as above), we would get the configuration $\emptyset; \{p \approx^?_{\mathcal{R},\lambda} q,\ \mathrm{N}_1 \approx^?_{\mathcal{R},\lambda} f,\ \mathrm{N}_2 \approx^?_{\mathcal{R},\lambda} a,\ \mathrm{N}_2 \approx^?_{\mathcal{R},\lambda} c,\ \mathrm{N}_2 \approx^?_{\mathcal{R},\lambda} \mathrm{N}_3\}; \{x \mapsto \mathrm{N}_1(\mathrm{N}_2, \mathrm{N}_2),\ y' \mapsto \mathrm{N}_2,\ y \mapsto \mathrm{N}_3\}$. But for the given $\mathcal{R}_\lambda$, the constraint $\{p \approx^?_{\mathcal{R},\lambda}$

q, $\mathrm{N}_1 \approx^?_{\mathcal{R},\lambda} f$, $\mathrm{N}_2 \approx^?_{\mathcal{R},\lambda} a$, $\mathrm{N}_2 \approx^?_{\mathcal{R},\lambda} c$, $\mathrm{N}_2 \approx^?_{\mathcal{R},\lambda} \mathrm{N}_3\}$ does not have a solution (because the neighborhoods of a and c do not intersect). Hence, we would lose a unifier.

4.2.4 Rules for solving neighborhood constraints

Let Φ be a *name-neighborhood mapping*. The *combination* of two mappings Φ and Ψ, denoted by $\Phi \odot \Psi$, is defined as

$$\begin{aligned}\Phi \odot \Psi := &\{\mathrm{N} \mapsto \Phi(\mathrm{N}) \mid \mathrm{N} \in \mathit{dom}(\Phi)\backslash \mathit{dom}(\Psi)\} \cup \\ &\{\mathrm{N} \mapsto \Psi(\mathrm{N}) \mid \mathrm{N} \in \mathit{dom}(\Psi)\backslash \mathit{dom}(\Phi)\} \cup \\ &\{\mathrm{N} \mapsto \Phi(\mathrm{N}) \cap \Psi(\mathrm{N}) \mid \mathrm{N} \in \mathit{dom}(\Psi) \cap \mathit{dom}(\Phi)\}.\end{aligned}$$

We call Φ and Ψ *compatible*, if $(\Phi \odot \Psi)(\mathrm{N}) \neq \varnothing$ for all $\mathrm{N} \in \mathit{dom}(\Phi \odot \Psi)$. Otherwise they are *incompatible*.

A *constraint configuration* is a pair $C; \Phi$, where C is a set of $(\mathcal{R}, \lambda)$-neighborhood constraints to be solved, and Φ is a name-neighborhood mapping (as a set of rules), representing the $(\mathcal{R}, \lambda)$-solution computed so far. We say that Ψ is an $(\mathcal{R}, \lambda)$*-solution of a constraint configuration* $C; \Phi$ if Ψ is an $(\mathcal{R}, \lambda)$-solution to C, and Ψ and Φ are compatible.

The constraint simplification algorithm $\mathcal{CS}$ transforms constraint configurations, exhaustively applying the following rules ($\perp$ indicates failure):

(FFS) Function Symbols: $\{f \approx^?_{\mathcal{R},\lambda} g\} \uplus C;\ \Phi; \Longrightarrow C;\ \Phi,$ if $\mathcal{R}(f, g) \geqslant \lambda$.

(NFS) Name vs Function Symbol:
$\{\mathrm{N} \approx^?_{\mathcal{R},\lambda} g\} \uplus C;\ \Phi \Longrightarrow C;\ \Phi \odot \{\mathrm{N} \mapsto \mathbf{pc}(g, \mathcal{R}, \lambda)\}$.

(FSN) Function Symbol vs Name:$\{g \approx^?_{\mathcal{R},\lambda} \mathrm{N}\} \uplus C; \Phi \Longrightarrow \{\mathrm{N} \approx^?_{\mathcal{R},\lambda} g\} \cup C; \Phi$.

(NN1) Name vs Name 1:
$\{\mathrm{N} \approx^?_{\mathcal{R},\lambda} \mathrm{M}\} \uplus C;\ \Phi \Longrightarrow C;\ \Phi \odot \{\mathrm{N} \mapsto \{f\}, \mathrm{M} \mapsto \mathbf{pc}(f, \mathcal{R}, \lambda)\}$,
where $\mathrm{N} \in \mathit{dom}(\Phi)$, $f \in \Phi(\mathrm{N})$.

(NN2) Name vs Name 2: $\{\mathrm{M} \approx^?_{\mathcal{R},\lambda} \mathrm{N}\} \uplus C;\ \Phi \Longrightarrow \{\mathrm{N} \approx^?_{\mathcal{R},\lambda} \mathrm{M}\} \cup C;\ \Phi$,
where $\mathrm{M} \notin \mathit{dom}(\Phi)$, $\mathrm{N} \in \mathit{dom}(\Phi)$.

(Fail1) Failure 1: $\{f \approx^?_{\mathcal{R},\lambda} g\} \uplus C;\ \Phi \Longrightarrow \perp,$ if $\mathcal{R}(f, g) < \lambda$.

(Fail2) Failure 2: $C;\, \Phi \Longrightarrow \bot$, if there exists $\mathrm{N} \in dom(\Phi)$ with $\Phi(\mathrm{N}) = \varnothing$.

The **NN1** rule causes branching, generating n branches where n is the number of elements in $\Phi(\mathrm{N})$. (Remember that by definition, the proximity class of each symbol is finite.) When the derivation does not fail, the terminal configuration has the form $\{\mathrm{N}_1 \approx^?_{\mathcal{R},\lambda} \mathrm{M}_1, \dots, \mathrm{N}_n \approx^?_{\mathcal{R},\lambda} \mathrm{M}_n\};\, \Phi$, where for each $1 \leqslant i \leqslant n$, $\mathrm{N}_i, \mathrm{M}_i \notin dom(\Phi)$. Such a constraint is trivially solvable.

Theorem 4.2.4. *The constraint simplification algorithm $\mathcal{CS}$ is terminating.*

Proof. With each configuration $C; \Phi$ we associate a complexity measure, which is a triple of natural numbers (n_1, n_2, n_3): n_1 is the number of symbols occurrences in C, n_2 is the number of equations of the form $g \approx^?_{\mathcal{R},\lambda} \mathrm{N}$ in C, and n_3 is the number of equations of the form $\mathrm{M} \approx^?_{\mathcal{R},\lambda} \mathrm{N}$ in C, where $\mathrm{M} \notin dom(\Phi)$ and $\mathrm{N} \in dom(\Phi)$. Measures are compared by the lexicographic extension of the ordering $>$ on natural numbers. It is a well-founded ordering. The rules **(FFS)**, **(NFS)**, **(NN1)** decrease n_1. The rule **(FSN)** does not change n_1 and decreases n_2. The rule **(NN2)** does not change n_1 and n_2 and decreases n_3. The failing rules cause termination immediately. Hence, each rule reduces the measure or terminates. It implies the termination of the algorithm. $\square$

In the statements below, we assume $\mathcal{R}$ and λ to be given and the problems are to be solved with respect to them.

Lemma 4.2.3. *1. If $C; \Phi \Longrightarrow \bot$ by* **(Fail1)** *or* **(Fail2)** *rules, then (C, Φ) does not have an $(\mathcal{R}, \lambda)$-solution.*

2. Let $C_1; \Phi_1 \Longrightarrow C_2; \Phi_2$ be a step performed by a constraint solving (non-failing) rule. Then any $(\mathcal{R}, \lambda)$-solution of $C_1; \Phi_2$ is also an $(\mathcal{R}, \lambda)$-solution of $C_1; \Phi_1$.

Proof. 1. For **(Fail1)**, the lemma follows from the definition of $(\mathcal{R}, \lambda)$-solution. For **(Fail2)**, no Ψ is compatible with a Φ that maps a name to the empty set.

2. The lemma is straightforward for **(FFS)**, **(FSN)**, and **(NN2)**. To show it for **(NFS)**, we take Ψ, which solves $C;\, \Phi \odot \{\mathrm{N} \mapsto \mathbf{pc}(g, \mathcal{R}, \lambda)\}$. By the definition of $\odot$, we get $\Psi(\mathrm{N}) \subseteq \mathbf{pc}(g, \mathcal{R}, \lambda)$. But then Ψ is a solution to $\{\mathrm{N} \approx^?_{\mathcal{R},\lambda} g\} \uplus C;\, \Phi$. To show the lemma holds for **(NN1)**, we take

a solution Ψ of $C; \Phi \odot \{\mathrm{N} \mapsto \{f\}, \mathrm{M} \mapsto \mathbf{pc}(f, \mathcal{R}, \lambda)\}$. It implies that $\Psi(\mathrm{N}) = \{f\}$ and $\Psi(\mathrm{M}) \subseteq \mathbf{pc}(f, \mathcal{R}, \lambda)$. But then we immediately get that Ψ solves $\{\mathrm{N} \approx^?_{\mathcal{R},\lambda} \mathrm{M}\} \uplus C; \Phi$.

□

Theorem 4.2.5 (Soundness of $\mathcal{CS}$). *Let C be an $(\mathcal{R}, \lambda)$-neighborhood constraint such that $\mathcal{CS}$ produces a maximal derivation $C; \varnothing \Longrightarrow^* C'; \Phi$. Then Φ is an $(\mathcal{R}, \lambda)$-solution of $C \backslash C'$, and C' is a set of constraints between names which is trivially $(\mathcal{R}, \lambda)$-satisfiable.*

Proof. If a neighborhood equation is not between names, there is a rule in $\mathcal{CS}$ which applies to it. Hence, a maximal derivation cannot stop with a C' that contains such an equation. As for neighborhood equations between names, only two rules deal with them: **(NN1)** and **(NN2)**. But they apply only if at least one of the involved names belongs to the domain of the corresponding mapping. Hence, it can happen that an equation of the form $\mathrm{N} \approx^?_{\mathcal{R},\lambda} \mathrm{M}$ is never transformed by $\mathcal{CS}$. When the algorithm stops, such equations remain in C' and are trivially solvable. We can remove C' from each configuration in $C; \varnothing \Longrightarrow^* C'; \Phi$ without affecting any step, getting a derivation $C \backslash C'; \varnothing \Longrightarrow^* \varnothing; \Phi$. Obviously, Φ is an $(\mathcal{R}, \lambda)$-solution of $\varnothing; \Phi$. By induction on the length of the derivation and Lemma 4.2.3, we get that Φ is an $(\mathcal{R}, \lambda)$-solution of $C \backslash C'; \varnothing$ and hence, it solves $C \backslash C'$. □

Remark 4.2.1. When a neighborhood constraint C is produced by the pre-unification algorithm, then every maximal $\mathcal{CS}$-derivation starting from $C; \varnothing$ ends either in $\perp$ or in the pair of the form $\varnothing; \Phi$. This is due to the fact that the **VE** rule (which introduces names in pre-unification problems) and the subsequent decomposition steps always produce chains of neighborhood equations of the form $\mathrm{N}_1 \approx_{\mathcal{R},\lambda} \mathrm{N}_2, \mathrm{N}_2 \approx_{\mathcal{R},\lambda} \mathrm{N}_3, \ldots, \mathrm{N}_n \approx_{\mathcal{R},\lambda} f$, $n \geqslant 1$, for the introduced N's and for some f.

Theorem 4.2.6 (Completeness of $\mathcal{CS}$). *Let C be an $(\mathcal{R}, \lambda)$-neighborhood constraint produced by the pre-unification algorithm, and Φ be one of its solutions. Let $dom(\Phi) = \{\mathrm{N}_1, \ldots, \mathrm{N}_n\}$. Then for each n-tuple $c_1 \in \Phi(\mathrm{N}_1), \ldots, c_n \in \Phi(\mathrm{N}_n)$ there exists a $\mathcal{CS}$-derivation $C; \varnothing \Longrightarrow^* \varnothing; \Psi$ with $c_i \in \Psi(\mathrm{N}_i)$ for each $1 \leqslant i \leqslant n$.*

Proof. We fix $c_1, \ldots, c_n$ such that $c_1 \in \Phi(\mathrm{N}_1), \ldots, c_n \in \Phi(\mathrm{N}_n)$.

First, note that $dom(\Phi)$ coincides with $\mathcal{N}(C)$. It is implied by the assumption that C is produced by pre-unification, and Remark 4.2.1 above.

The desired derivation is constructed recursively, where the important step is to identify a single inference. To see how such a single step is made, we consider a configuration $C_i; \Phi_i$ in this derivation ($i \geqslant 0$, $C_0 = C$, $\Phi_0 = \varnothing$). We have $dom(\Phi_i) \subseteq dom(\Phi)$. During construction, we will maintain the following two invariants for each $i \geqslant 0$ (easy to check that they hold for $i = 0$):

(I1) Φ is an $(\mathcal{R}, \lambda)$-solution of (C_i, Φ_i), and

(I2) for all $1 \leqslant j \leqslant n$, if $\mathrm{N}_j \in dom(\Phi_i)$, then $c_j \in \Phi_i(\mathrm{N}_j)$.

We consider the following cases:

- C_i contains an equation of the form $f \approx^?_{\mathcal{R},\lambda} g$. By **(I1)**, $\mathcal{R}(f, g) \geqslant \lambda$. Then we make the **(FFS)** step with $f \approx^?_{\mathcal{R},\lambda} g$, obtaining $C_{i+1} = C_i \backslash \{f \approx^?_{\mathcal{R},\lambda} g\}$ and $\Phi_{i+1} = \Phi_i$. Obviously, **(I1)** and **(I2)** hold also for the new configuration.

- Otherwise, assume C_i contains an equation $\mathrm{N}_k \approx^?_{\mathcal{R},\lambda} g$, where $1 \leqslant k \leqslant n$. Since Φ solves $C_; \Phi_i$, we have $\Phi_i(\mathrm{N}) \neq \varnothing$ for all $\mathrm{N} \in dom(\Phi_i)$ and there is only one choice to make the step: the **(NFS)** rule. It gives $C_{i+1} = C_i \backslash \{\mathrm{N}_k \approx^?_{\mathcal{R},\lambda} g\}$ and $\Phi_{i+1} = \Phi_i \odot \{\mathrm{N}_k \mapsto \mathbf{pc}(g, \mathcal{R}, \lambda)\}$. Since Φ solves, in particular, $\mathrm{N}_k \approx^?_{\mathcal{R},\lambda} g$, we have $\Phi(\mathrm{N}_k) \subseteq \mathbf{pc}(g, \mathcal{R}, \lambda)$ and, hence, $c_k \in \mathbf{pc}(g, \mathcal{R}, \lambda)$. First, assume $\mathrm{N}_k \notin dom(\Phi_i)$. Then $\Phi_{i+1}(\mathrm{N}_k) = \mathbf{pc}(g, \mathcal{R}, \lambda)$ and both **(I1)** and **(I2)** hold for $i + 1$. Now, assume $\mathrm{N}_k \in dom(\Phi_i)$. Then $\Phi_{i+1}(\mathrm{N}_k) = \Phi_i(\mathrm{N}_k) \cap \mathbf{pc}(g, \mathcal{R}, \lambda)$. Besides, **(I2)** implies $c_k \in \Phi_i(\mathrm{N}_k)$. Hence, $c_k \in \Phi_{i+1}(\mathrm{N}_k)$, which implies that **(I2)** holds for $i + 1$. From $c_k \in \Phi_{i+1}(\mathrm{N}_k)$ and $c_k \in \Phi(\mathrm{N}_k)$ we get that Φ is compatible with Φ_{i+1}. Moreover, Φ solves C_i, therefore, it solves C_{i+1}. Hence, Φ solves $C_{i+1}; \Phi_{i+1}$ and **(I1)** holds for $i + 1$ as well.

- Otherwise, assume C_i contains an equation of the form $\mathrm{N}_k \approx^?_{\mathcal{R},\lambda} N_j$, where $1 \leqslant k, j \leqslant n$ and $\mathrm{N}_k \in dom(\Phi_i)$. By **(I1)**, we have $\mathrm{N}_k, \mathrm{N}_j \in dom(\Phi)$, $\Phi(\mathrm{N}_k) \cap \Phi(\mathrm{N}_j) \neq \varnothing$, and $\Phi(\mathrm{N}_k) \cap \Phi_i(\mathrm{N}_k) \neq \varnothing$. By **(I2)**, we have $c_k \in \Phi_i(\mathrm{N}_k)$. But since $c_k \in \Phi(\mathrm{N}_k)$, we have $c_k \in \Phi(\mathrm{N}_k) \cap \Phi_i(\mathrm{N}_k)$. We make the step with the **(NN1)** rule, choosing the mapping $\mathrm{N}_k \mapsto \{c_k\}$. It gives $C_{i+1} = C_i \backslash \{\mathrm{N}_k \approx^?_{\mathcal{R},\lambda} \mathrm{N}_j\}$ and $\Phi_{i+1} = \Phi_i \odot \{\mathrm{N}_k \mapsto \{c_k\}, \mathrm{N}_j \mapsto \mathbf{pc}(c_k, \mathcal{R}, \lambda)\}$.

  To see that **(I1)** holds for $i + 1$, the only nontrivial thing is to check that Φ and Φ_{i+1} are compatible. For this, $\Phi_{i+1}(\mathrm{N}_k) \cap \Phi(\mathrm{N}_k) \neq \varnothing$ and $\Phi_{i+1}(\mathrm{N}_j) \cap \Phi(\mathrm{N}_j) \neq \varnothing$ should be shown.

Proving $\Phi_{i+1}(\mathrm{N}_k) \cap \Phi(\mathrm{N}_k) \neq \varnothing$: By construction, $\Phi_{i+1}(\mathrm{N}_k) = \{c_k\}$. By assumption, $c_k \in \Phi(\mathrm{N}_k)$. Hence, $\Phi_{i+1}(\mathrm{N}_k) \cap \Phi(\mathrm{N}_k) \neq \varnothing$.

Proving $\Phi_{i+1}(\mathrm{N}_j) \cap \Phi(\mathrm{N}_j) \neq \varnothing$: Since Φ solves $\mathrm{N}_k \approx^?_{\mathcal{R},\lambda} N_j$ and $c_k \in \Phi(\mathrm{N}_k)$, we have $\Phi(\mathrm{N}_j) \subseteq \mathbf{pc}(c_k, \mathcal{R}, \lambda)$. If $\mathrm{N}_j \notin \mathit{dom}(\Phi_i)$, then $\Phi_{i+1}(\mathrm{N}_j) = \mathbf{pc}(c_k, \mathcal{R}, \lambda)$ and $\Phi_{i+1}(\mathrm{N}_j) \cap \Phi(\mathrm{N}_j) \neq \varnothing$. If $\mathrm{N}_j \in \mathit{dom}(\Phi_i)$, then by **(I2)**, $c_j \in \Phi_i(\mathrm{N}_j)$. On the other hand, $c_j \in \Phi(\mathrm{N}_j)$ and, therefore, $c_j \in \mathbf{pc}(c_k, \mathcal{R}, \lambda)$. Since $\Phi_{i+1}(\mathrm{N}_j) = \Phi_i(\mathrm{N}_j) \cap \mathbf{pc}(c_k, \mathcal{R}, \lambda)$, we get $c_j \in \Phi_{i+1}(\mathrm{N}_j)$ and, hence, $\Phi_{i+1}(\mathrm{N}_j) \cap \Phi(\mathrm{N}_j) \neq \varnothing$.

To see that **(I2)** holds is easier. In fact, we have already shown above that $\Phi_{i+1}(\mathrm{N}_k) = \{c_k\}$. As for N_j, we need to show that $c_j \in \Phi_{i+1}(N_j)$. According to the way how we defined $\Phi_{i+1}(N_j)$, we have $\mathbf{pc}(c_k, \mathcal{R}, \lambda) \subseteq \Phi_{i+1}(N_j)$. From the proof of **(I1)**, we know that $\Phi(N_j) \subseteq \mathbf{pc}(c_k, \mathcal{R}, \lambda)$. From these inclusions and the assumption $c_j \in \Phi(N_j)$ we get $c_j \in \Phi_{i+1}(N_j)$.

- The other cases will be dealt by the rules **(FSN)** and **(NN2)**. The invariants for them trivially hold, since these rules do not change the problem.

By **(I1)**, the configurations in our derivation are solvable. Therefore, the failing rules do not apply. Hence, the derivation ends with $\varnothing; \Psi$ for some Ψ. By construction, $\mathit{dom}(\Psi) = \{\mathrm{N}_1, \dots, \mathrm{N}_n\}$. By **(I2)**, $c_i \in \Psi(\mathrm{N}_i)$ for each $1 \leqslant i \leqslant n$.

□

Example 4.2.4. The pre-unification derivation in Example 4.2.2 gives the neighborhood constraint $C = \{p \approx^?_{\mathcal{R},\lambda} q, \mathrm{N}_1 \approx^?_{\mathcal{R},\lambda} f,\ \mathrm{N}_2 \approx^?_{\mathcal{R},\lambda} a,\ \mathrm{N}_3 \approx^?_{\mathcal{R},\lambda} g,\ \mathrm{N}_4 \approx^?_{\mathcal{R},\lambda} d,\ \mathrm{N}_1 \approx^?_{\mathcal{R},\lambda} \mathrm{N}_3,\ \mathrm{N}_2 \approx^?_{\mathcal{R},\lambda} \mathrm{N}_4\}$. For $\mathcal{R}_\lambda = \{(a, b), (b, c), (c, d), (a, b'), (b', c'), (c', d), (f, g), (p, q)\}$, the algorithm $\mathcal{CS}$ gives four solutions:

$$
\begin{aligned}
\Phi_1 &= \{\mathrm{N}_1 \mapsto \{f\}, \mathrm{N}_2 \mapsto \{b\},\ \mathrm{N}_3 \mapsto \{f, g\}, \mathrm{N}_4 \mapsto \{c\}\}\\
\Phi_2 &= \{\mathrm{N}_1 \mapsto \{f\}, \mathrm{N}_2 \mapsto \{b'\}, \mathrm{N}_3 \mapsto \{f, g\}, \mathrm{N}_4 \mapsto \{c'\}\}.\\
\Phi_3 &= \{\mathrm{N}_1 \mapsto \{g\}, \mathrm{N}_2 \mapsto \{b\},\ \mathrm{N}_3 \mapsto \{f, g\}, \mathrm{N}_4 \mapsto \{c\}\}\\
\Phi_4 &= \{\mathrm{N}_1 \mapsto \{g\}, \mathrm{N}_2 \mapsto \{b'\}, \mathrm{N}_3 \mapsto \{f, g\}, \mathrm{N}_4 \mapsto \{c'\}\}.
\end{aligned}
$$

Referring to the mappings Φ and Φ' and the substitution $\boldsymbol{\mu}$ in Example 4.2.2, it is easy to observe that $\Phi(\boldsymbol{\mu}) \cup \Phi'(\boldsymbol{\mu}) = \Phi_1(\boldsymbol{\mu}) \cup \Phi_2(\boldsymbol{\mu}) \cup \Phi_3(\boldsymbol{\mu}) \cup \Phi_4(\boldsymbol{\mu})$.

4.3 Matching

In a matching equation, one side is always ground. Therefore, no chains of proximity between terms appear, which allows us to solve the problem in one single stage. Names are not needed anymore, and hence, the X-terms are defined as in section 4.1.

4.3.1 Problem statement

Definition 4.3.1. *An X-substitution* $\boldsymbol{\sigma}$ *is an* $(\mathcal{R},\lambda)$-X-matcher *of* t *to* s*, if every* $\sigma \in \mathsf{substs}(\boldsymbol{\sigma})$ *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.*

Usually one would formulate the matching problem in the fuzzy context 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 formulated in this way has finitely many solutions (when proximity classes of symbols are finite, as in our case). Instead of trying to compute all of them, we will be aiming at computing their compact representations in the form of X-substitutions. Hence, our algorithm will solve the following reformulated version of the problem:

Given: a proximity relation $\mathcal{R}$, a cut value λ, and two terms t and s.

Find: an X-substitution $\boldsymbol{\sigma}$ such that each $\sigma \in \mathsf{substs}(\boldsymbol{\sigma})$ is an $(\mathcal{R},\lambda)$-matcher of t to s.

Such a reformulation will help us compute a single X-substitution instead of multiple matchers.

Example 4.3.1. Let the proximity relation $\mathcal{R}$ be defined as

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

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 $\mathsf{substs}(\boldsymbol{\sigma})$ contains all relevant $(\mathcal{R},\lambda)$-matchers of t to s for different values of λ:

$$0 < \lambda \leqslant 0.5: \qquad \boldsymbol{\sigma} = \{x \mapsto \{g_1,g_2,h_1,h_2\}(\{a_1,a_2,b\})\}.$$

	$\mathsf{substs}(\boldsymbol{\sigma})$ contains 12 substitutions.
$0.5 < \lambda \leqslant 0.6$:	$\boldsymbol{\sigma} = \{x \mapsto \{h_1, h_2\}(\{b\})\}$.
	$\mathsf{substs}(\boldsymbol{\sigma}) = \{\{x \mapsto h_1(b)\},\ \{x \mapsto h_2(b)\}\}$.
$0.6 < \lambda \leqslant 0.7$:	$\boldsymbol{\sigma} = \{x \mapsto \{h_2\}(\{b\})\}$.
	$\mathsf{substs}(\boldsymbol{\sigma}) = \{\{x \mapsto h_2(b)\}\}$.
$0.7 < \lambda \leqslant 1$:	No substitution matches t to s.

4.3.2 The algorithm

Given $\mathcal{R}$, λ, t, and s (where s does not contain variables), to solve an $(\mathcal{R}, \lambda)$-matching problem $t \precsim^?_{\mathcal{R},\lambda} s$, we create the initial pair $\{t \precsim^?_{\mathcal{R},\lambda} s\}; \varnothing$ 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

$\{f(t_1, \ldots, t_n) \precsim^?_{\mathcal{R},\lambda} g(s_1, \ldots, s_n)\} \uplus M; S \Longrightarrow M \cup \{t_i \precsim^?_{\mathcal{R},\lambda} s_i \mid 1 \leqslant i \leqslant n\}; S,$
if $n \geqslant 0$, $\mathcal{R}(f, g) \geqslant \lambda$.

VE-M: Variable elimination

$\{x \precsim^?_{\mathcal{R},\lambda} s\} \uplus M;\ S \Longrightarrow M;\ S \cup \{x \approx \mathbf{xpc}(s, \mathcal{R}, \lambda)\}.$

Mer-M: Merging

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

Cla-M: Clash

$\{f(t_1, \ldots, t_n) \precsim^?_{\mathcal{R},\lambda} g(s_1, \ldots, s_m)\} \uplus M;\ S \Longrightarrow \bot,\qquad$ where $\mathcal{R}(f, g) < \lambda$.

Inc-M: Inconsistency

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

The matching algorithm $\mathfrak{M}$ uses the rules to transform pairs as long as possible, returning either $\bot$ (indicating failure), or the pair $\varnothing; S$ (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 $x\sigma \in \mathsf{terms}(\mathbf{t})$. We also assume that $\bot$ has no solution.

Lemma 4.3.1. *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 **xpc**, we have $x\sigma \in \mathbf{xpc}(s, \mathcal{R}, \lambda)$ iff $\mathcal{R}(x\sigma, s) \geqslant \lambda$, which is equivalent to the fact that σ is a $(\mathcal{R}, \lambda)$-matcher of $x \precsim^?_{\mathcal{R},\lambda} s$. $\square$

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: $size(x) = 1$, $size(f(t_1, \dots, t_n)) = 1 + \sum_{i=1}^{n} size(t_i)$, $size(t \precsim^?_{\mathcal{R},\lambda} s) = size(t \approx s) = size(t) + size(s)$, and $size(S) = \sum_{p \in S} size(p)$, where S is a set of matching problems or equations.

Theorem 4.3.1. *Given an $(\mathcal{R}, \lambda)$-matching problem $t \precsim^?_{\mathcal{R},\lambda} s$, the matching algorithm $\mathfrak{M}$ terminates and computes an X-substitution $\boldsymbol{\sigma}$ such that* $\mathsf{substs}(\boldsymbol{\sigma})$ *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 $\langle size(M), size(S) \rangle$ of sizes of M and S, which implies termination.

Soundness. If $\sigma \in \mathsf{substs}(\boldsymbol{\sigma})$, then σ is a relevant $(\mathcal{R}, \lambda)$-matcher of t to s.

Let $\{t \precsim^?_{\mathcal{R},\lambda} s\}; \varnothing \Longrightarrow^+ \varnothing; S$ be the derivation in $\mathfrak{M}$ that computes $\boldsymbol{\sigma}$. By definition of computed X-substitution, we can conclude that $\sigma \in \mathsf{substs}(\boldsymbol{\sigma})$ iff σ is a solution of $\varnothing; S$. By induction on the length of the given derivation, using Lemma 4.3.1, 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 $\sigma \in \mathsf{substs}(\boldsymbol{\sigma})$.

Since $t \precsim^?_{\mathcal{R},\lambda} s$ is solvable, we can construct a derivation $\{t \precsim^?_{\mathcal{R},\lambda} s\}; \varnothing \Longrightarrow^+ \varnothing; S$ 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 4.3.1, we would never apply Cla-M and Inc-M rules, because it would contradict the solvability of $t \precsim^?_{\mathcal{R},\lambda} s$. Hence, we can construct the mentioned derivation, for which, again by Lemma 4.3.1, we have that σ is a $(\mathcal{R}, \lambda)$-solution of $\varnothing; S$. By definitions of computed X-substitution $\boldsymbol{\sigma}$ and substs, it implies that $\sigma \in \mathsf{substs}(\boldsymbol{\sigma})$.

□

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

Example 4.3.2. We illustrate the steps of the algorithm $\mathfrak{M}$ for the matching problem in Example 4.3.1 for $\lambda = 0.6$ and $\lambda = 0.8$.

$$\begin{aligned}
\lambda = 0.6 :& \\
&\{f(x,x) \precsim^?_{\mathcal{R},\lambda} f(g_1(a_1), g_2(a_2))\}; \varnothing \Longrightarrow_{\text{DEC-M}} \\
&\{x \precsim^?_{\mathcal{R},\lambda} g_1(a_1),\ x \precsim^?_{\mathcal{R},\lambda} g_2(a_2)\}; \varnothing \Longrightarrow_{\text{VE-M}} \\
&\{x \precsim^?_{\mathcal{R},\lambda} g_2(a_2)\}; \{x \approx \{g_1, h_1, h_2\}(\{a_1, b\})\} \Longrightarrow_{\text{VE-M}} \\
&\varnothing; \{x \approx \{g_1, h_1, h_2\}(\{a_1, b\}),\ x \approx \{g_2, h_1, h_2\}(\{a_2, b\})\} \Longrightarrow_{\text{MER-M}} \\
&\varnothing; \{x \approx \{h_1, h_2\}(\{b\})\}.
\end{aligned}$$

$$\begin{aligned}
\lambda = 0.8 :& \\
&\{f(x,x) \precsim^?_{\mathcal{R},\lambda} f(g_1(a_1), g_2(a_2))\}; \varnothing \Longrightarrow_{\text{DEC-M}} \\
&\{x \precsim^?_{\mathcal{R},\lambda} g_1(a_1),\ x \precsim^?_{\mathcal{R},\lambda} g_2(a_2)\}; \varnothing \Longrightarrow_{\text{VE-M}} \\
&\{x \precsim^?_{\mathcal{R},\lambda} g_2(a_2)\}; \{x \approx \{g_1\}(\{a_1\})\} \Longrightarrow_{\text{VE-M}} \\
&\varnothing; \{x \approx \{g_1\}(\{a_1\}),\ x \approx \{g_2, h_2\}(\{a_2, b\})\} \Longrightarrow_{\text{INC-M}} \\
&\bot.
\end{aligned}$$

If $\boldsymbol{\sigma}$ is a computed X-matcher of an $(\mathcal{R}, \lambda)$-matching problem $t \precsim^?_{\mathcal{R},\lambda} s$, then we can compute the minimal and maximal approximation degrees $\mathfrak{d}_{\min}$ and $\mathfrak{d}_{\max}$ of this solution. Intuitively, these are the minimal and maximal distances between a solution instance of t and s. More formally:

$$\begin{aligned}
\mathfrak{d}_{\min}(t, s, \mathcal{R}, \boldsymbol{\sigma}) &:= \min\{\mathcal{R}(t\sigma, s) \mid \sigma \in \mathsf{substs}(\boldsymbol{\sigma})\}, \\
\mathfrak{d}_{\max}(t, s, \mathcal{R}, \boldsymbol{\sigma}) &:= \max\{\mathcal{R}(t\sigma, s) \mid \sigma \in \mathsf{substs}(\boldsymbol{\sigma})\}.
\end{aligned}$$

For instance, if $\mathcal{R}$ is the proximity relation from Example 4.3.1, $\lambda = 0.4$ and the matching problem is $g_2(x) \lesssim^?_{\mathcal{R},\lambda} h_2(a_1)$, then its X-matcher is $\boldsymbol{\sigma} = \{x \mapsto \{a_1, a_2, b\}\}$, $\mathsf{substs}(\boldsymbol{\sigma}) = \{\{x \mapsto a_1\}, \{x \mapsto a_2\}, \{x \mapsto b\}\}$ and we have

- $\mathfrak{d}_{\min}(t, s, \mathcal{R}, \boldsymbol{\sigma}) = 0.5$, because $\mathcal{R}(g_2(x)\{x \mapsto a_2\}, h_2(a_1)) = 0.5$ is the minimum.
- $\mathfrak{d}_{\max}(t, s, \mathcal{R}, \boldsymbol{\sigma}) = 0.8$, because $\mathcal{R}(g_2(x)\{x \mapsto a_1\}, h_2(a_1)) = 0.8$ is the maximum.

The proximity relation $\mathcal{R}$ can be represented as a weighted undirected graph, whose vertices form a (finite) subset of $\mathcal{F}$ and where there is an edge of weight $\mathfrak{d}$ between vertices f and g iff $\mathcal{R}(f, g) = \mathfrak{d} > 0$. 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 below 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 4.3.2. *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 matching 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 some cut value λ.*

Proof. In the representation of the graph for $\mathcal{R}$, the 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 [16]. From the given matching problem $t \lesssim^?_{\mathcal{R},\lambda} s$ we can also construct its directed acyclic graph (dag) representation with shared variables (see, e.g., [8]). At each node g of s, we add a pointer to the adjacency list of g in the graph representation of $\mathcal{R}$. The nodes in the representation of t are labeled by function symbols and variables occurring in t. In fact, we have a graph representation *dag*(t) of t and a tree representation *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 *dag*(t) and *tree*(s), comparing the node labels pairwise. Assume the label of a nonvariable node f in *dag*(t) is an element of the adjacency list of the corresponding node g in *tree*(s), and $\mathfrak{d} \geqslant \lambda$ 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 *dag*(t) and a node g in *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 *subtree*(s, g) of *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 *dag*(t) and *tree*(s) (VE-M rule). If *tree*$_h$ to which x points already exists, we go top-down in the trees *tree*$_h$ and *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 *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 [16]. 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 [8]. 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 $O(\mathit{degree}(g))$ 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 $O(n|V_\mathcal{R}| + n|E_\mathcal{R}|)$. □

4.4 Anti-unification

Similarly to matching, anti-unification does not need to store information about proximity chains between terms. We can thus solve the problem in one single stage. For keeping the consistency in the notation, our algorithm will work not on the terms to be generalized, but on their **xpc**'s.

4.4.1 Problem statement

Definition 4.4.1. *An X-term* $\mathbf{t}$ *is an* $(\mathcal{R}, \lambda)$-X-generalization *of a term* s*, if every* $t \in \mathsf{terms}(\mathbf{t})$ *is an* $(\mathcal{R}, \lambda)$-generalization *of* s*.*

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

Usually, the anti-unification problem for terms in a fuzzy setting is formulated as follows: Given a proximity relation $\mathcal{R}$, a cut value λ, and two terms t and s, find an $(\mathcal{R}, \lambda)$-lgg of t and s.

As it was the case for matching, the anti-unification problem formulated as above has finitely many solutions. Therefore, instead of computing all of them, we will again be aiming at computing their compact representations, this time in the form of X-terms. The reformulated version of the problem solved by our algorithm will then be:

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 $r \in \mathsf{terms}(\mathbf{r})$ is an $(\mathcal{R}, \lambda)$-lgg of t and s.

Unlike matching, there is not a single answer to this reformulated problem. But still, we will compute fewer X-lggs than lggs. Moreover, if we restrict ourselves to linear lggs (i.e., those with a single occurrence of generalization variables), then we get a single answer.

Example 4.4.1. 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 x is the syntactic lgg of t and s. As for proximity-based generalizations, for each of the following

X-terms $\mathbf{r}$, the set $\mathsf{terms}(\mathbf{r})$ contains all $(\mathcal{R}, \lambda)$-lggs of t and s for different values of λ:

$$\begin{aligned}
0 < \lambda \leqslant 0.5 : \quad & \mathbf{r}_1 = \{f, g\}(x_1, x_1, x_3). \\
& \mathsf{terms}(\mathbf{r}_1) = \{f(x_1, x_1, x_3),\ g(x_1, x_1, x_3)\}. \\
& \mathbf{r}_2 = \{f, g\}(x_1, x_2, x_2). \\
& \mathsf{terms}(\mathbf{r}_2) = \{f(x_1, x_2, x_2),\ g(x_1, x_2, x_2)\}. \\
0.5 < \lambda \leqslant 0.6 : \quad & \mathbf{r} = \{f, g\}(x_1, x_2, x_2). \\
& \mathsf{terms}(\mathbf{r}) = \{f(x_1, x_2, x_2),\ g(x_1, x_2, x_2)\}. \\
0.6 < \lambda \leqslant 0.7 : \quad & \mathbf{r} = \{f, g\}(x_1, x_2, x_3). \\
& \mathsf{terms}(\mathbf{r}) = \{f(x_1, x_2, x_3),\ g(x_1, x_2, x_3)\}. \\
0.7 < \lambda \leqslant 1 : \quad & \mathbf{r} = x. \\
& \mathsf{terms}(\mathbf{r}) = \{x\}.
\end{aligned}$$

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

$$\begin{aligned}
0 < \lambda \leqslant 0.7 : \quad & \mathbf{r} = \{f, g\}(x_1, x_2, x_3). \\
& \mathsf{terms}(\mathbf{r}) = \{f(x_1, x_2, x_3),\ g(x_1, x_2, x_3)\}. \\
0.7 < \lambda \leqslant 1 : \quad & \mathbf{r} = x. \\
& \mathsf{terms}(\mathbf{r}) = \{x\}.
\end{aligned}$$

4.4.2 The algorithm

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) $x : \mathbf{xpc}(t, \mathcal{R}, \lambda) \triangleq \mathbf{xpc}(s, \mathcal{R}, \lambda)$ where x is a fresh variable. Then we put it in the initial tuple $\{x : \mathbf{xpc}(t, \mathcal{R}, \lambda) \triangleq \mathbf{xpc}(s, \mathcal{R}, \lambda)\}; \varnothing; x$, 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 $\varnothing; S; \mathbf{r}$. 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**

$\{x : \mathbf{f}(\mathbf{t}_1, \ldots, \mathbf{t}_n) \triangleq \mathbf{g}(\mathbf{s}_1, \ldots, \mathbf{s}_n)\} \uplus A;\ S;\ \mathbf{r} \Longrightarrow$
$\{y_1 : \mathbf{t}_1 \triangleq \mathbf{s}_1, \ldots, y_n : \mathbf{t}_n \triangleq \mathbf{s}_n\} \cup A;\ S;\ \mathbf{r}\{x \mapsto (\mathbf{f} \cap \mathbf{g})(y_1, \ldots, y_n)\}$,
where $n \geqslant 0$, $\mathbf{f} \cap \mathbf{g} \neq \varnothing$.

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

Mer-AU: **Merging**

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

Note that 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 4.4.1. *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 $\mathbf{r}_1, \ldots, \mathbf{r}_n$, $n \geqslant 1$, such that $\cup_{i=1}^{n}\mathsf{terms}(\mathbf{r}_i)$ consists of all $(\mathcal{R}, \lambda)$-least general generalizations of t and s (modulo variable renaming).*

Proof. Like in Theorem 4.3.1, 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 $r \in \cup_{i=1}^{n}\mathsf{terms}(\mathbf{r}_i)$, then r is an $(\mathcal{R}, \lambda)$-generalization of t and s.

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

$$\{x : \mathbf{xpc}(t, \mathcal{R}, \lambda) \triangleq \mathbf{xpc}(s, \mathcal{R}, \lambda)\}; \varnothing; x\boldsymbol{\vartheta}_0 \Longrightarrow^k$$

$$\emptyset; S; x\boldsymbol{\vartheta}_0\boldsymbol{\vartheta}_1 \cdots \boldsymbol{\vartheta}_k, \tag{4.5}$$

where $\boldsymbol{\vartheta}_0 = Id$, $k \geqslant 1$ and $\mathbf{r}_j = x\boldsymbol{\vartheta}_0\boldsymbol{\vartheta}_1 \cdots \boldsymbol{\vartheta}_k$. For 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 ($0 \leqslant l < k$), then we have $x' \in \mathcal{V}(x\boldsymbol{\vartheta}_0 \cdots \boldsymbol{\vartheta}_l)$, $\mathbf{t}' = \mathbf{xpc}(t, \mathcal{R}, \lambda)|_{p'}$ for some position p' in $\mathbf{xpc}(t, \mathcal{R}, \lambda)$, and $\mathbf{s}' = \mathbf{xpc}(s, \mathcal{R}, \lambda)|_{p'}$ for the same position p' in $\mathbf{xpc}(s, \mathcal{R}, \lambda)$;
- 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 ($0 \leqslant l < k$) 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 ($0 < l \leqslant k$) with $A_l = \emptyset$, we have $x' \in \mathcal{V}(x\boldsymbol{\vartheta}_0 \cdots \boldsymbol{\vartheta}_l)$, $\mathsf{terms}(\mathbf{t}') \subseteq \mathsf{terms}(\mathbf{xpc}(t, \mathcal{R}, \lambda)|_{p'})$ for some position p' in $\mathbf{xpc}(t, \mathcal{R}, \lambda)$, and $\mathsf{terms}(\mathbf{s}') \subseteq \mathsf{terms}(\mathbf{xpc}(s, \mathcal{R}, \lambda)|_{p'})$ for the same position p' in $\mathbf{xpc}(s, \mathcal{R}, \lambda)$.

Coming back to the derivation (4.5), we prove that for all $0 \leqslant i < k$, 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 $0 \leqslant i < k$ 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 $x_i : \mathbf{f}_i(\mathbf{t}_{i_1}, \ldots, \mathbf{t}_{i_{n_i}}) \triangleq \mathbf{g}_i(\mathbf{s}_{i_1}, \ldots, \mathbf{s}_{i_{n_i}})$ with $\mathbf{f}_i \cap \mathbf{g}_i \neq \emptyset$, which is replaced in A_{i+1} by new AUTs $y_1 : \mathbf{t}_{i_1} \triangleq \mathbf{s}_{i_1}, \ldots, y_{n_i} : \mathbf{t}_{i_{n_i}} \triangleq \mathbf{s}_{i_{n_i}}$, and $\boldsymbol{\vartheta}_{i+1} = \{x_i \mapsto (\mathbf{f}_i \cap \mathbf{g}_i)(y_1, \ldots, y_{n_i})\}$. There is a position p in both $\mathbf{xpc}(t, \mathcal{R}, \lambda)$ and $\mathbf{xpc}(s, \mathcal{R}, \lambda)$ such that $\mathbf{xpc}(t, \mathcal{R}, \lambda)|_p = \mathbf{f}_i(\mathbf{t}_{i_1}, \ldots, \mathbf{t}_{i_{n_i}})$ and $\mathbf{xpc}(s, \mathcal{R}, \lambda)|_p = \mathbf{g}_i(\mathbf{s}_{i_1}, \ldots, \mathbf{s}_{i_{n_i}})$. In the same position p in t and s, we have respectively $t_p = f_i(t_{i_1}, \ldots, t_{i_{n_i}}) \in \mathsf{terms}(\mathbf{f}_i(\mathbf{t}_{i_1}, \ldots, \mathbf{t}_{i_{n_i}}))$ and $s_p = g_i(s_{i_1}, \ldots, s_{i_{n_i}}) \in \mathsf{terms}(\mathbf{s}_i(\mathbf{s}_{i_1}, \ldots, \mathbf{s}_{i_{n_i}}))$. 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, it is obvious that $x_i\boldsymbol{\vartheta}_{i+1} = (\mathbf{f}_i \cap \mathbf{g}_i)(y_1, \ldots, y_{n_i})$ is an $(\mathcal{R}, \lambda)$-generalization of $f_i(t_{i_1}, \ldots, t_{i_{n_i}})$

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 $\mathbf{t}_{i_1} \sqcap \mathbf{t}_{i_2} \neq \varnothing$ and $\mathbf{s}_{i_1} \sqcap \mathbf{s}_{i_2} \neq \varnothing$. 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 $\mathbf{xpc}(t, \mathcal{R}, \lambda)$ such that $\mathsf{terms}(\mathbf{t}_{i_j}) \subseteq \mathsf{terms}(\mathbf{xpc}(t, \mathcal{R}, \lambda)|_{p_j})$, $j = 1, 2$. From $\mathbf{t}_{i_1} \sqcap \mathbf{t}_{i_2} \neq \varnothing$ we have $\mathsf{terms}(\mathbf{t}_{i_1}) \cap \mathsf{terms}(\mathbf{t}_{i_2}) \neq \varnothing$ and, as a consequence, $\mathsf{terms}(\mathbf{xpc}(t, \mathcal{R}, \lambda)|_{p_1}) \cap \mathsf{terms}(\mathbf{xpc}(t, \mathcal{R}, \lambda)|_{p_2}) \neq \varnothing$. Similarly, we get $\mathsf{terms}(\mathbf{s}_{i_1}) \cap \mathsf{terms}(\mathbf{s}_{i_2}) \neq \varnothing$.

  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 $q \in \mathsf{terms}(x\boldsymbol{\vartheta}_0\boldsymbol{\vartheta}_1 \cdots \boldsymbol{\vartheta}_i)$ 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 $dom(\varphi_t) = dom(\sigma_t)\backslash\{x_{i_2}\}$, $x_{i_1}\varphi_t \in \mathsf{terms}(\mathbf{xpc}(t, \mathcal{R}, \lambda)|_{p_1}) \cap \mathsf{terms}(\mathbf{xpc}(t, \mathcal{R}, \lambda)|_{p_2})$, and $y\varphi_t = y\sigma_t$ for all $y \in dom(\varphi_t)\backslash\{x_{i_1}\}$. Such a φ_t exists, since we have shown that $\mathsf{terms}(\mathbf{xpc}(t, \mathcal{R}, \lambda)|_{p_1}) \cap \mathsf{terms}(\mathbf{xpc}(t, \mathcal{R}, \lambda)|_{p_2}) \neq \varnothing$. By definition, we know that every element of the set $\mathsf{terms}(\mathbf{xpc}(t, \mathcal{R}, \lambda)|_{p_1}) \cap \mathsf{terms}(\mathbf{xpc}(t, \mathcal{R}, \lambda)|_{p_2})$ 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 $q \in \mathsf{terms}(x\boldsymbol{\vartheta}_0\boldsymbol{\vartheta}_1 \cdots \boldsymbol{\vartheta}_{i+1})$, we get $q\varphi_t \simeq_{\mathcal{R},\lambda} t$ and $q\varphi_s \simeq_{\mathcal{R},\lambda} s$, which implies 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 the derivation (4.5), $x\boldsymbol{\vartheta}_0\boldsymbol{\vartheta}_1 \cdots \boldsymbol{\vartheta}_k$ 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 $r \in \mathsf{terms}(\mathbf{r}_j)$, 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 $r' \in \cup_{i=1}^{n}\mathsf{terms}(\mathbf{r}_i)$ 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 $\mathsf{terms}(\mathit{head}(\mathbf{xpc}(t, \mathcal{R}, \lambda))) \cap \mathsf{terms}(\mathit{head}(\mathbf{xpc}(s, \mathcal{R}, \lambda))) = \emptyset$. 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 $r = h(r_1, \dots, r_m)$. Then we have that $t = f(t_1, \dots, t_m)$, $s = g(s_1, \dots, s_m)$, and $h \in \mathbf{f} \cap \mathbf{g}$, where $\mathbf{f} = \mathbf{xpc}(f, \mathcal{R}, \lambda)$ and $\mathbf{g} = \mathbf{xpc}(g, \mathcal{R}, \lambda)$. We apply Dec-AU to the AUT $x : \mathbf{xpc}(t, \mathcal{R}, \lambda) \triangleq \mathbf{xpc}(s, \mathcal{R}, \lambda)$ and obtain new AUTs $y_k : \mathbf{xpc}(t_k, \mathcal{R}, \lambda) \triangleq \mathbf{xpc}(s_k, \mathcal{R}, \lambda)$, $1 \leqslant k \leqslant m$. Note that each r_k, $1 \leqslant k \leqslant m$, is an $(\mathcal{R}, \lambda)$-lgg of t_k and s_k. Then by the induction hypothesis, for each $1 \leqslant k \leqslant m$ we compute $\mathbf{r}'_k$ so that there exists $r'_k \in \mathsf{terms}(\mathbf{r}'_k)$ 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 which computes $(\mathbf{f} \cap \mathbf{g})(\mathbf{r}'_1, \dots, \mathbf{r}'_m)$.

However, this does not yet guarantee that $(\mathbf{f} \cap \mathbf{g})(\mathbf{r}'_1, \dots, \mathbf{r}'_m)$ 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 $(\mathbf{f} \cap \mathbf{g})(\mathbf{r}'_1, \dots, \mathbf{r}'_m)$. Since r is a generalization of t and s, having the same variable in p_i and p_j implies that $\mathsf{terms}(\mathbf{xpc}(t, \mathcal{R}, \lambda)|_{p_i}) \cap \mathsf{terms}(\mathbf{xpc}(t, \mathcal{R}, \lambda)|_{p_j}) \neq \emptyset$. Therefore, we get $\mathbf{xpc}(t, \mathcal{R}, \lambda)|_{p_i} \sqcap \mathbf{xpc}(t, \mathcal{R}, \lambda)|_{p_j} \neq \emptyset$. Similarly, $\mathbf{xpc}(s, \mathcal{R}, \lambda)|_{p_i} \sqcap \mathbf{xpc}(s, \mathcal{R}, \lambda)|_{p_j} \neq \emptyset$. Having different y_i and y_j in positions p_i and p_j in $(\mathbf{f} \cap \mathbf{g})(\mathbf{r}'_1, \dots, \mathbf{r}'_m)$ implies that we have $y_i : \mathbf{xpc}(t, \mathcal{R}, \lambda)|_{p_i} \triangleq \mathbf{xpc}(s, \mathcal{R}, \lambda)|_{p_i}$ and $y_j : \mathbf{xpc}(t, \mathcal{R}, \lambda)|_{p_j} \triangleq \mathbf{xpc}(s, \mathcal{R}, \lambda)|_{p_j}$ 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 $(\mathbf{f} \cap \mathbf{g})(\mathbf{r}'_1, \dots, \mathbf{r}'_i, \dots, \mathbf{r}''_j, \dots, \mathbf{r}'_m)$ 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 $r' \in \mathsf{terms}(\mathbf{r}')$ 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 $\mathsf{terms}(\mathbf{r})$ contains all linear lggs of s and t (modulo variable renaming).

Example 4.4.2. Let $\mathcal{R}$ be a proximity relation defined as

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

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:

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

The store shows how to obtain terms which are $(\mathcal{R},\lambda)$-close to the original terms. For instance, when $0 < \lambda \leqslant 0.5$, store$_1$ tells us that for any substitution σ from the set $\mathsf{substs}(\{x_1 \mapsto \{a\}, x_3 \mapsto \{a_3, a'\}\})$, 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, i.e., $f(x_1,x_1,x_3)\sigma \simeq_{\mathcal{R},\lambda} t$ and $g(x_1,x_1,x_3)\sigma \simeq_{\mathcal{R},\lambda} t$.

Similarly, for any substitution ϑ from the set $\mathsf{substs}(\{x_1 \mapsto \{b\}, x_3 \mapsto \{b_3, b'\}\})$ (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, i.e., we have $f(x_1,x_1,x_3)\vartheta \simeq_{\mathcal{R},\lambda} s$ and $g(x_1,x_1,x_3)\vartheta \simeq_{\mathcal{R},\lambda} s$.

Now we illustrate how the first two X-lggs have been computed. Let $\lambda = 0.5$. For the initial problem we take $\mathbf{xpc}(t,\mathcal{R},\lambda) = \{f,g\}(\{a_1,a\},\{a_2,a,a'\},$

$\{a_3, a'\})$ and $\mathbf{xpc}(s, \mathcal{R}, \lambda) = \{g, f\}(\{b_1, b\}, \{b_2, b, b'\}, \{b_3, b'\})$ and proceed as follows:

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

Now there are two alternatives: to merge either x_1 and x_2 or x_2 and x_3. In the first case, we get

$$\varnothing;\ \{x_1 : \{a\} \triangleq \{b\}, x_3 : \{a_3, a'\} \triangleq \{b_3, b'\}\};\ \{f, g\}(x_1, x_1, x_3).$$

In the second case, the result is

$$\varnothing;\ \{x_1 : \{a_1, a\} \triangleq \{b_1, b\}, x_2 : \{a'\} \triangleq \{b'\}\};\ \{f, g\}(x_1, x_2, x_2).$$

These are exactly the stores and X-lggs we have seen above, at the beginning of this example.

Example 4.4.3. Consider again the proximity relation and the terms from Example 4.4.2, but this times assume we are interested in linear generalizations. Then the stores and X-lggs are the following:

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

$$\begin{aligned}&\quad \text{X-lgg} = \{f, g\}(x_1, x_2, x_3).\\ &0.7 < \lambda \leqslant 1:\\ &\quad \text{store} = \{x : \{f(a_1, a_2, a_3)\} \triangleq g(b_1, b_2, b_3)\}.\\ &\quad \text{X-lgg} = x.\end{aligned}$$

Theorem 4.4.2. *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 4.3.2). 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, as we have already mentioned, 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 $O(n^2|V_\mathcal{R}| + |E_\mathcal{R}|)$ bound for the running time of computing a single $(\mathcal{R}, \lambda)$-X-lgg of t and s. □

4.5 Conclusion

In this chapter we investigated three fundamental problems for approximate automated reasoning: unification, matching and anti-unification, where the approximation is expressed with fuzzy proximity relations. Fuzzy proximity is not a transitive relation, which makes these problems challenging. In general, there is no single solution to them. The same holds for unification, matching and anti-unification with tolerance relations, which are crisp counterparts of proximity.

We developed algorithms that solve the mentioned problems, aiming at computing a compact representation of solution sets. Our approach is based on proximity classes, which relaxes certain restrictions imposed by the block-based approach considered in the previous chapter. The class-based approach, in general, allows to solve more problems than it would be possible by the block-based approach, and the problems might also have more solutions. Therefore, a compact representation of solutions plays an important role here. We use so called extended terms (X-terms) as such a compact representation for a set of terms. In X-terms, instead of function symbols, finite sets of function symbols are permitted. X-substitutions map variables to X-terms, representing sets of substitutions.

When working with X-terms, the unification algorithm needs to keep information about the chains of proximal terms, in order not to lose solutions. For this purpose, we introduced copies of function symbols, called names, and allowed them to be used in the X-terms. Our unification algorithm has two stages. The first one - pre-unification - ends with a set of variable only constraints, a set of neighborhood constraints, and a substitution. This stage does not depend on the λ-cut. The second stage solves the neighborhood constraints by computing a finite set of name-neighborhood mappings. The composition of each such mapping with the substitution obtained in the pre-unification and each X-unifier of the variable only constraints set leads to the set of X-unifiers of the original problem.

Our matching algorithm computes a single X-substitution solution for solvable proximity (and tolerance) matching problems.

Like unification, 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 the problem becomes unitary in the sense that there exists a single linear X-lgg (which still represents a finite set of lggs as standard terms), and our algorithm computes it.

All three algorithms are terminating, sound, and complete. Time and space complexities of the algorithms are also analyzed.

Since all algorithms work with compact representations of the terms, we have not included the explicit computation of the actual approximation degree of the solutions, but it is easy. Each X-solution is a set of solutions, for which a range of approximation degrees can be computed. We could therefore store in the tuples on which the algorithms work, also the minimum and the maximum approximation degrees computed so far, in the way we store the actual computed approximation degree in the next chapters.

The current chapter allows only symbols with the same arity to be proximal. The next chapter extends the algorithms presented above to permit proximal function symbols with possibly different arities, similarly to the analogous extension of similarity-based unification described in [1].

Chapter 5

Proximity Relations for Fully Fuzzy Signatures

The algorithms presented until now are powerful and cover many different cases, but there is still room for improvement, as they can be extended in various directions. An extension we discuss in this chapter tolerates mismatches for functional symbols not only in their names, but also in the arity and argument order, thus considering fully fuzzy signatures. This extension has been investigated for similarity relations by Aït-Kaci and Pasi [3], but so far, no work has been done for proximity. This chapter covers exactly this blank spot. Our algorithms for unification, matching, and anti-unification generalize both our work described in the previous chapter, and the work done for similarity, extended now to proximity. Below, we will first briefly review the related work from in [3], followed by a detailed description of our contribution in the rest of the chapter.

5.1 Notions and terminology

Argument relations and mappings. Given two sets $N = \{1, \dots, n\}$ and $M = \{1, \dots, m\}$, a binary *argument relation* over $N \times M$ is a (possibly empty) subset of $N \times M$. We denote argument relations by ρ.

An *argument mapping* is an argument relation that is a partial injective function. In other words, an argument mapping π from $N = \{1, \dots, n\}$ to $M = \{1, \dots, m\}$ is a function $\pi : I_n \mapsto I_m$, where $I_n \subseteq N$, $I_m \subseteq M$ and $|I_n| = |I_m|$. Note that it can be also the empty mapping: $\pi : \emptyset \mapsto \emptyset$. Usually, for $\pi : I_n \mapsto I_m$ we write $\pi = \{i \mapsto \pi(i) \mid i \in I_n\}$. The inverse of an argument mapping is again an argument mapping.

An argument relation $\rho \subseteq N \times M$ is (i) *left-total* if for all $i \in N$ there exists $j \in M$ such that $(i, j) \in \rho$; (ii) *right-total* if for all $j \in M$ there exists $i \in N$ such that $(i, j) \in \rho$. *Correspondence relations* are those that are both left- and right-total.

Given a proximity relation $\mathcal{R}$ over $\mathcal{F}$, we assume in this chapter that for each pair of function symbols f and g with $\mathcal{R}(f,g) = \alpha > 0$, where f is n-ary and g is m-ary, there is also given an argument relation ρ over $\{1,\dots,n\} \times \{1,\dots,m\}$. We use the notation $f \sim^{\rho}_{\mathcal{R},\alpha} g$. These argument relations satisfy the following conditions: ρ is the empty relation if f or g is a constant; ρ is the identity if $f = g$; $f \sim^{\rho}_{\mathcal{R},\alpha} g$ iff $g \sim^{\rho^{-1}}_{\mathcal{R},\alpha} f$, where ρ^{-1} is the inverse of ρ.

Proximity relations over terms. Recall that each proximity relation $\mathcal{R}$ considered in this thesis is defined on $\mathcal{F} \cup \mathcal{V}$ such that $\mathcal{R}(f,x) = 0$ for all $f \in \mathcal{F}$ and $x \in \mathcal{V}$, and $\mathcal{R}(x,y) = 0$ for all $x \neq y$, $x,y \in \mathcal{V}$. It is assumed that for each $f \in \mathcal{F}$, its $(\mathcal{R},\lambda)$-proximity class $\{g \mid \mathcal{R}(f,g) \geqslant \lambda\}$ is finite for any $\mathcal{R}$ and λ.

We extended such an $\mathcal{R}$ to terms from $\mathcal{T}(\mathcal{F},\mathcal{V})$. In order to include our newly defined argument relations, we extend it further in this chapter as follows:

(a) $\mathcal{R}(t,s) := 0$ if $\mathcal{R}(head(t), head(s)) = 0$;

(b) $\mathcal{R}(t,s) := 1$ if $t = s$ and $t,s \in \mathcal{V}$;

(c) $\mathcal{R}(t,s) := \mathcal{R}(f,g) \wedge \mathcal{R}(t_{i_1}, s_{j_1}) \wedge \cdots \wedge \mathcal{R}(t_{i_k}, s_{j_k})$, if $t = f(t_1,\dots,t_n)$, $s = g(s_1,\dots,s_m)$, $f \sim^{\rho}_{\mathcal{R},\lambda} g$, and $\rho = \{(i_1,j_1),\dots,(i_k,j_k)\}$.

If $\mathcal{R}(t,s) \geqslant \lambda$, we write $t \simeq_{\mathcal{R},\lambda} s$.

5.2 Related work: similarity-based symbolic techniques in fully fuzzy signatures

Unification for similarity relations in fully fuzzy signatures. First, we review the unification rules introduced by Aït-Kaci and Pasi in [2]. They work on configurations of the form $P; \sigma; \alpha$, where P is a unification problem to be solved, σ is a substitution computed so far, and α is the unification approximation degree computed so far.

Tri-U-AKP: **Trivial**

$\{x \simeq^{?}_{\mathcal{R},\lambda} x\} \uplus P; \sigma; \alpha \Longrightarrow P; \sigma; \alpha.$

Dec-U-AKP: **Decomposition**

$\{f(t_1,\dots,t_n) \simeq^?_{\mathcal{R},\lambda} g(s_1,\dots,s_m)\} \uplus P; \sigma; \alpha \Longrightarrow$
$\quad P \cup \{t_i \simeq^?_{\mathcal{R},\lambda} s_{\rho(i)} \mid 1 \leqslant i \leqslant n\}; \sigma; \alpha \wedge \beta,$
where $0 \leqslant n \leqslant m$, $f \sim^{\rho}_{\mathcal{R},\beta} g$, and $\beta \geqslant \lambda$.

Elim-U-AKP: **Variable elimination**

$\{x \simeq^?_{\mathcal{R},\lambda} t\} \uplus P; \sigma; \alpha \Longrightarrow P\{x \mapsto t\}; \sigma\{x \mapsto t\}; \alpha,$ where $x \notin \mathcal{V}(t)$.

Ori1-U-AKP: **Orient 1**

$\{t \simeq^?_{\mathcal{R},\lambda} x\} \uplus P; \sigma; \alpha \Longrightarrow \{x \simeq^?_{\mathcal{R},\lambda} t\} \uplus P; \sigma; \alpha,$ where t is not a variable.

Ori2-U-AKP: **Orient 2**

$\{g(s_1,\dots,s_m) \simeq^?_{\mathcal{R},\lambda} f(t_1,\dots,t_n)\} \uplus P; \sigma; \alpha \Longrightarrow$
$\quad \{f(t_1,\dots,t_n) \simeq^?_{\mathcal{R},\lambda} g(s_1,\dots,s_m)\} \uplus P; \sigma; \alpha$
where $0 \leqslant n < m$.

As usual, to $(\mathcal{R},\lambda)$-unify t and s for a given similarity relation $\mathcal{R}$ and the cut value λ, one creates the initial unification problem $\{t \simeq^?_{\mathcal{R},\lambda} s\}$ and applies the rules as long as possible. The computed most general unifier σ together with its approximation degree α is collected from the final configuration $\emptyset; \sigma; \alpha$, if such a configuration is reached. Otherwise, the process ends with the failure $\perp$, indicating that the given terms are not $(\mathcal{R},\lambda)$-unifiable.

Example 5.2.1. We illustrate similarity-based unification in fully fuzzy signatures using an example taken from [2]. Let a, b, c, d be constants, f, g, l be binary function symbol, h be a ternary function symbols and the relation $\mathcal{R}$ be defined as $a \sim_{\mathcal{R},0.7} b$, $c \sim_{\mathcal{R},0.6} d$, $f \sim^{\{(1,2),(2,1)\}}_{\mathcal{R},0.9} g$, $g \sim^{\{(1,2),(2,1)\}}_{\mathcal{R},0.9} f$, and $l \sim^{\{(1,2),(2,3)\}}_{\mathcal{R},0.8} h$. We take the cut value $\lambda = 0.6$. The algorithm performs the following steps to unify $h(x, g(y,b), f(y,c))$ and $l(f(a,z), g(d,c))$:

$$\begin{aligned}
&\{h(x, g(y,b), f(y,c)) \simeq^?_{\mathcal{R},0.6} l(f(a,z), g(d,c))\}; Id; 1 \Longrightarrow_{\text{Ori2-U-AKP}}\\
&\{l(f(a,z), g(d,c)) \simeq^?_{\mathcal{R},0.6} h(x, g(y,b), f(y,c))\}; Id; 1 \Longrightarrow_{\text{Dec-U-AKP}}\\
&\{f(a,z) \simeq^?_{\mathcal{R},0.6} g(y,b),\ g(d,c) \simeq^?_{\mathcal{R},0.6} f(y,c)\}; Id; 0.8 \Longrightarrow_{\text{Dec-U-AKP}}\\
&\{a \simeq^?_{\mathcal{R},0.6} b,\ z \simeq^?_{\mathcal{R},0.6} y,\ g(d,c) \simeq^?_{\mathcal{R},0.6} f(y,c)\}; Id; 0.8 \Longrightarrow_{\text{Dec-U-AKP}}\\
&\{z \simeq^?_{\mathcal{R},0.6} y,\ g(d,c) \simeq^?_{\mathcal{R},0.6} f(y,c)\}; Id; 0.7 \Longrightarrow_{\text{Elim-U-AKP}}\\
&\{g(d,c) \simeq^?_{\mathcal{R},0.6} f(y,c)\}; \{z \mapsto y\}; 0.7 \Longrightarrow_{\text{Dec-U-AKP}}
\end{aligned}$$

$$\{d \simeq^?_{\mathcal{R},0.6} c,\ c \simeq^?_{\mathcal{R},0.6} y\}; \{z \mapsto y\}; 0.7 \Longrightarrow_{\text{Dec-U-AKP}}$$
$$\{c \simeq^?_{\mathcal{R},0.6} y\}; \{z \mapsto y\}; 0.6 \Longrightarrow_{\text{Ori-U-AKP}}$$
$$\{y \simeq^?_{\mathcal{R},0.6} c\}; \{z \mapsto y\}; 0.6 \Longrightarrow_{\text{Elim-U-AKP}}$$
$$\varnothing; \{z \mapsto c, y \mapsto c\}; 0.6.$$

To verify, we have

$$h(x, g(y, b), f(y, c))\{z \mapsto c, y \mapsto c\} =$$
$$h(x, g(c, b), f(c, c)) \simeq_{\mathcal{R},0.6} l(f(a, c), g(d, c)) =$$
$$l(f(a, z), g(d, c))\{z \mapsto c, y \mapsto c\}.$$

Anti-unification for similarity relations in fully fuzzy signatures. The anti-unification rules given in [2] are formulated differently from those that we show below. We chose to present them in the notation that is closer to ours. There are a couple of other differences in the presentation, namely:

- we do not show rules that deal with variables occurring in the input terms. They can be easily modeled by constants that are proximal only to themselves;
- we assume the λ-cut given and compare the symbol proximity degrees to λ, while in [2] there is no such cut and the rules proceed if the proximity between symbols is positive.

The rules are the following (the name abbreviations are combined with AU for anti-unification and with AKP for the last names of the authors of [2], although the rules do not follow their notation):

DL-AU-AKP: Decomposition left

$$\{x : f(t_1, \dots, t_n) \triangleq g(s_1, \dots, s_m)\} \uplus A; S; r; \alpha \Longrightarrow$$
$$\{y_i : t_i \triangleq s_{\rho(i)} \mid 1 \leqslant i \leqslant n\} \cup A; S; r\{x \mapsto f(y_1, \dots, y_n)\}; \alpha \wedge \beta,$$

where $f \sim^{\rho}_{\mathcal{R},\beta} g$, $\beta > \lambda$, and $0 \leqslant n \leqslant m$.

DR-AU-AKP: Decomposition right

$$\{x : f(t_1, \dots, t_n) \triangleq g(s_1, \dots, s_m)\} \uplus A; S; r; \alpha \Longrightarrow$$
$$\{y_i : t_{\rho(i)} \triangleq s_i \mid 1 \leqslant i \leqslant m\} \cup A; S; r\{x \mapsto g(y_1, \dots, y_m)\}; \alpha \wedge \beta,$$

where $g \sim^{\rho}_{\mathcal{R},\beta} f$, $\beta > \lambda$, and $0 \leqslant m \leqslant n$.

Sol-AU-AKP: Solve
$\{x : t \triangleq s\} \uplus A; S; r; \alpha \Longrightarrow A; \{x : t \triangleq s\} \cup S; r; \alpha \wedge \beta$,
where $\mathcal{R}(head(t), head(s)) < \lambda$.

Mer-AU-AKP: Merge
$A; S \uplus \{x_1 : t_1 \triangleq s_1,\ x_2 : t_2 \triangleq s_2\}; r; \alpha \Longrightarrow$
$\quad A; \{x_1 : t_1 \triangleq s_1\} \cup S; r\{x_2 \mapsto x_1\}; \alpha \wedge \beta_1 \wedge \beta_2$,
where $\mathcal{R}(t_1, t_2) = \beta_1 \geqslant \lambda$ and $\mathcal{R}(s_1, s_2) = \beta_2 \geqslant \lambda$.

To anti-unify t and s, as usual, one creates the initial configuration $\{x : t \triangleq s\}; \varnothing; x; 1$ and applies the rules as long as possible. The terminal configuration has the form $\varnothing; S; r; \alpha$, from which one can extract

- the generalization r of t and s,
- the substitutions σ_t and σ_s that match r respectively to t and s, defined as $\sigma_t = \{y \mapsto t' \mid y : t' \triangleq s' \in S\}$ and $\sigma_s = \{y \mapsto s' \mid y : t' \triangleq s' \in S\}$,
- and the generalization degree α

such that $r\sigma_t \simeq_{\mathcal{R},\alpha} t$ and $r\sigma_s \simeq_{\mathcal{R},\alpha} s$.

Example 5.2.2. We illustrate the described anti-unification algorithm with an example which is a modified version of a similar example from [2]. Assume a, b, c, d, e are constants, f, g, and l are binary function symbols, and h is a ternary function symbol. Let the similarity relation $\mathcal{R}$ be defined as $a \sim_{\mathcal{R},0.7} b$, $c \sim_{\mathcal{R},0.6} d$, $f \sim^{\{(1,1),(2,2)\}}_{\mathcal{R},0.8} g$, $g \sim^{\{(1,1),(2,2)\}}_{\mathcal{R},0.8} f$, and $l \sim^{\{(1,1),(2,2)\}}_{\mathcal{R},0.9} h$. Let the cut value be $\lambda = 0.5$. Then the algorithm performs the following steps to $(\mathcal{R}, \lambda)$-anti-unify the terms $t = h(g(b, e), f(e, c), e)$ and $s = l(f(a, d), g(c, c))$:

$$
\begin{aligned}
&\{x : h(g(b,e), f(e,c), e) \triangleq l(f(a,d), g(c,c))\}; \varnothing; x; 1 \Longrightarrow_{\text{DR-AU-AKP}}\\
&\{y_1 : g(b,e) \triangleq f(a,d),\ y_2 : f(e,c) \triangleq g(c,c)\}; \varnothing; l(y_1, y_2); 0.9 \Longrightarrow_{\text{DL-AU-AKP}}\\
&\{z_1 : b \triangleq a,\ z_2 : e \triangleq d,\ y_2 : f(e,c) \triangleq g(c,c)\}; \varnothing;\\
&\qquad l(g(z_1, z_2), y_2); 0.8 \Longrightarrow_{\text{DL-AU-AKP}}\\
&\{z_2 : e \triangleq d,\ y_2 : f(e,c) \triangleq g(c,c)\}; \varnothing; l(g(b, z_2), y_2); 0.7 \Longrightarrow_{\text{Sol-AU-AKP}}\\
&\{y_2 : f(e,c) \triangleq g(c,c)\}; \{z_2 : e \triangleq d\}; l(g(b, z_2), y_2); 0.7 \Longrightarrow_{\text{DL-AU-AKP}}\\
&\{u_1 : e \triangleq c,\ u_2 : c \triangleq c\}; \{z_2 : e \triangleq d\}; l(g(b, z_2), f(u_1, u_2)); 0.7 \Longrightarrow_{\text{Sol-AU-AKP}}\\
&\{u_2 : c \triangleq c\}; \{z_2 : e \triangleq d,\ u_1 : e \triangleq c\}; l(g(b, z_2), f(u_1, u_2)); 0.7 \Longrightarrow_{\text{DL-AU-AKP}}
\end{aligned}
$$

$\emptyset; \{z_2 : e \triangleq d,\ u_1 : e \triangleq c\}; l(g(b, z_2), f(u_1, c)); 0.7 \Longrightarrow_{\text{Mer-AU-AKP}}$
$\emptyset; \{u_1 : e \triangleq c\}; l(g(b, u_1), f(u_1, c)); 0.6.$

Hence, the computed generalization is $l(g(b, u_1), f(u_1, c))$. We extract the substitutions $\sigma_t = \{u_1 \mapsto e\}$ and $\sigma_s = \{u_1 \mapsto c\}$ from the computed store $\{u_1 : e \triangleq c\}$. The computed approximation degree is 0.6. We have:

- $l(g(b, u_1), f(u_1, c))\sigma_t = l(g(b, e), f(e, c)) \simeq_{\mathcal{R},0.6} t = h(g(b, e), f(e, c), e)$: Note that $l(g(b, e), f(e, c)) \simeq_{\mathcal{R},0.6} h(g(b, e), f(e, c), e)$ holds because of $\mathcal{R}(l(g(b, e), f(e, c)), h(g(b, e), f(e, c), e)) = 0.9 \geqslant 0.6$;

- $l(g(b, u_1), f(u_1, c))\sigma_s = l(g(b, c), f(c, c)) \simeq_{\mathcal{R},0.6} s = l(f(a, d), g(c, c))$: Note that $l(g(b, c), f(c, c)) \simeq_{\mathcal{R},0.6} l(f(a, d), g(c, c))$ holds because of $\mathcal{R}(l(g(b, c), f(c, c)), l(f(a, d), g(c, c))) = 0.6$.

5.3 Unification in fully fuzzy signatures

In this section we assume that all argument relations are correspondence relations. First, we formulate two technical lemmas that will be useful later, when we discuss the algorithm.

Lemma 5.3.1. *If all argument relations in $\mathcal{R}$ are correspondence relations, then for any λ-cut: (a) $t \simeq_{\mathcal{R},\lambda} s$ implies $\mathcal{V}(t) = \mathcal{V}(s)$; (b) no term is $(\mathcal{R}, \lambda)$-close to its proper subterm.*

Proof. We prove (a) by structural induction for terms. If both t and s are variables, then $t \simeq_{\mathcal{R},\lambda} s$ implies $t = s$. If they are nonvariable terms $t = f(t_1 \dots, t_n)$ and $s = g(s_1, \dots, s_n)$ with $f \sim^{\rho}_{\mathcal{R},\lambda} g$, then the correspondence property of ρ implies the following: for each t_i there is s_j such that $t_i \simeq_{\mathcal{R},\lambda} s_j$ and, hence, by the induction hypothesis $\mathcal{V}(t_i) = \mathcal{V}(s_j)$, and vice versa: for each s_j there is t_i such that $s_j \simeq_{\mathcal{R},\lambda} t_i$ and, hence, by the induction hypothesis $\mathcal{V}(s_j) = \mathcal{V}(t_i)$. Therefore, $\mathcal{V}(t) = \cup_{i=1}^{n} \mathcal{V}(t_i) = \cup_{j=1}^{m} \mathcal{V}(s_j) = \mathcal{V}(s)$.

To prove (b), by the definition of correspondence relations, a non-constant term cannot be $(\mathcal{R}, \lambda)$-close to a constant. According to the definition of proximity, no nonvariable term is $(\mathcal{R}, \lambda)$-close to a variable. By structural induction over terms we get that no term is $(\mathcal{R}, \lambda)$-close to its proper subterm. □

A set of $(\mathcal{R}, \lambda)$-equations $\{x \simeq_{\mathcal{R},\lambda} s\} \uplus P$ contains an *occurrence cycle* for x if $s \notin \mathcal{V}$ and there exist term-pairs (x_0, s_0), $(x_1, s_1), \ldots, (x_n, s_n)$ such that $x_0 = x$, $s_0 = s$, and for each $0 \leqslant i \leqslant n$ the set P contains an equation $x_i \simeq_{\mathcal{R},\lambda} s_i$ or $s_i \simeq_{\mathcal{R},\lambda} x_i$ with $x_{i+1} \in \mathcal{V}(s_i)$, where $x_{n+1} = x_0$.

Lemma 5.3.2. *Let all argument relations in $\mathcal{R}$ be correspondence relations. If a set of $(\mathcal{R}, \lambda)$-equations P contains an occurrence cycle for some variable, then P has no solution.*

Proof. By Lemma 5.3.1, no term can be $(\mathcal{R}, \lambda)$-close to its proper subterm. Therefore, equations containing an occurrence cycle cannot have a solution. □

5.3.1 Unification rules

Now we formulate a unification algorithm in a rule-based way. The rules work, as usual, on unification configurations, which here are triples $P; \sigma; \alpha$, where P is a unification problem, σ is the substitution computed so far, and α is the approximation degree, also computed so far.

Tri-U: **Trivial**

$\{x \simeq^?_{\mathcal{R},\lambda} x\} \uplus P; \sigma; \alpha \Longrightarrow P; \sigma; \alpha.$

Dec-U: **Decomposition**

$\{f(t_1, \ldots, t_n) \simeq^?_{\mathcal{R},\lambda} g(s_1, \ldots, s_m)\} \uplus P; \sigma; \alpha \Longrightarrow$
$\quad P \cup \{t_i \simeq^?_{\mathcal{R},\lambda} s_j \mid (i, j) \in \rho\}; \sigma; \alpha \wedge \beta,$
where $n, m \geqslant 0$, $f \sim^{\rho}_{\mathcal{R},\beta} g$, and $\beta \geqslant \lambda$.

Cla-U: **Clash**

$\{f(t_1, \ldots, t_n) \simeq^?_{\mathcal{R},\lambda} g(s_1, \ldots, s_m)\} \uplus P; \sigma; \alpha \Longrightarrow \bot, \qquad$ if $\mathcal{R}(f, g) < \lambda$.

Ori-U: **Orient**

$\{t \simeq^?_{\mathcal{R},\lambda} x\} \uplus P; \sigma; \alpha \Longrightarrow P \cup \{x \simeq^?_{\mathcal{R},\lambda} t\}; \sigma; \alpha, \qquad$ if t is not a variable.

Occ-U: **Occurrence check**

$\{x \simeq^?_{\mathcal{R},\lambda} g(s_1, \ldots, s_n)\} \uplus P; \sigma; \alpha \Longrightarrow \bot,$
if $\{x \simeq^?_{\mathcal{R},\lambda} g(s_1, \ldots, s_n)\} \uplus P$ has an occurrence cycle for x.

Var-E-U: Variable elimination

$\{x \simeq^?_{\mathcal{R},\lambda} g(s_1,\ldots,s_n)\} \uplus P; \sigma; \alpha \Longrightarrow P\vartheta \cup \{v_i \simeq^?_{\mathcal{R},\lambda} s_j \mid (i,j) \in \rho\}; \sigma\vartheta; \alpha \wedge \beta$,

where:

$\{x \simeq^?_{\mathcal{R},\lambda} g(s_1,\ldots,s_n)\} \uplus P$ does not contain an occurrence cycle for x,

$\vartheta = \{x \mapsto f(v_1,\ldots,v_m)\}$ with fresh variables $v_1,\ldots,v_m$,

$f \sim^{\rho}_{\mathcal{R},\beta} g$ with $\beta \geqslant \lambda$,

$n, m \geqslant 0$.

Given a unification problem P, we create the initial system $P; Id; 1$ and start applying the unification rules in all possible ways, generating a complete tree of derivations in the standard way. The **Var-E-U** rule causes branching, since there can be multiple f's satisfying the condition there. No rule applies to $\perp$ or to a *variables-only* configuration $\{x_1 \simeq^?_{\mathcal{R},\lambda} y_1,\ldots,x_n \simeq^?_{\mathcal{R},\lambda} y_n\}$. In the latter case we say that α is the computed approximation degree, $\sigma|_{\mathcal{V}(P)}$ is the computed substitution, and $\{x_1 \simeq_{\mathcal{R},\lambda} y_1,\ldots,x_n \simeq_{\mathcal{R},\lambda} y_n\}$ is the computed constraint. We denote the obtained unification algorithm by $\mathfrak{U}_{\mathit{full}}$.

5.3.2 Examples

In the examples below it is assumed that $\mathcal{R}(sym_1, sym_2) = 0$ for any pair of distinct symbols sym_1 and sym_2 except those for which the proximity is explicitly given.

Example 5.3.1. Let f, h be binary, and g unary function symbols, and let a, b, c be constants with:

$f \sim^{\{(1,1),(2,1)\}}_{\mathcal{R},0.6} g$,

$f \sim^{\{(1,1),(2,2)\}}_{\mathcal{R},0.7} h$,

$a \sim_{\mathcal{R},0.5} b$.

$b \sim_{\mathcal{R},0.4} c$.

Let the unification problem be $P = \{f(x,x) \simeq_{\mathcal{R},0.4} f(g(a), h(a,c))\}$. Then the algorithm $\mathfrak{U}_{\mathit{full}}$ starts with decomposition:

$\{f(x,x) \simeq^?_{\mathcal{R},0.4} f(g(a), h(a,c))\}; Id; 1 \Longrightarrow_{\text{Dec-U}}$

$$\{x \simeq^?_{\mathcal{R},0.4} g(a),\, x \simeq^?_{\mathcal{R},0.4} h(a,c)\}; Id; 1.$$

From here there are two ways to proceed by **Var-E-U** on $x \simeq^?_{\mathcal{R},0.4} g(a)$: by choosing the variable eliminating substitution either $\{x \mapsto g(v)\}$ or $\{x \mapsto f(v_1, v_2)\}$. The former one leads to failure, since $\mathcal{R}(g,h) = 0$. Therefore, we show here only the second derivation:

$$\{x \simeq^?_{\mathcal{R},0.4} g(a),\, x \simeq^?_{\mathcal{R},0.4} h(a,c)\}; Id; 1 \Longrightarrow_{\text{Var-E-U}}$$
$$\{v_1 \simeq^?_{\mathcal{R},0.4} a, v_2 \simeq^?_{\mathcal{R},0.4} a,\, f(v_1,v_2) \simeq^?_{\mathcal{R},0.4} h(a,c)\}; \{x \mapsto f(v_1,v_2)\}; 0.6.$$

Here we have two alternatives by **Var-E-U**, on each of v_1 and v_2. The alternatives for v_1, both $\{v_1 \mapsto a\}$, and $\{v_1 \mapsto b\}$, may lead to success. For v_2, the first alternative chooses $\{v_2 \mapsto a\}$, and ends up in failure since $\mathcal{R}(a,c) = 0$, while the second alternative, $\{v_2 \mapsto b\}$, is successful.

From all possible 4 branches, we directly present the final configuration obtained on the branch with $\{v_1 \mapsto b, v_2 \mapsto b\}$, and show here only the other successful derivation, for $\{v_1 \mapsto a, v_2 \mapsto b\}$:

$$\{v_1 \simeq^?_{\mathcal{R},0.4} a, v_2 \simeq^?_{\mathcal{R},0.4} a,\, f(v_1,v_2) \simeq^?_{\mathcal{R},0.4} h(a,c)\};$$
$$\quad \{x \mapsto f(v_1,v_2)\}; 0.6 \Longrightarrow^2_{\text{Var-E-U}}$$
$$\{f(a,b) \simeq^?_{\mathcal{R},0.4} h(a,c)\}; \{x \mapsto f(a,b), v_1 \mapsto a, v_2 \mapsto b\}; 0.5 \Longrightarrow_{\text{Dec-U}}$$
$$\{a \simeq^?_{\mathcal{R},0.4} a, b \simeq^?_{\mathcal{R},0.4} c\}; \{x \mapsto f(a,b), v_1 \mapsto a, v_2 \mapsto b\}; 0.5 \Longrightarrow^2_{\text{Dec-U}}$$
$$\emptyset; \{x \mapsto f(a,b), v_1 \mapsto a, v_2 \mapsto b\}; 0.4.$$

The substitution computed in this derivation is $\sigma_1 = \{x \mapsto f(a,b)\}$. It solves P, because $f(f(a,b), f(a,b)) \simeq_{\mathcal{R},0.4} f(g(a), h(a,c))$.

On the branch where we take $\{v_1 \mapsto b, v_2 \mapsto b\}$ we get $\sigma_2 = \{x \mapsto f(b,b)\}$ and $\alpha = 0.4$. It also solves P, because $f(f(b,b), f(b,b)) \simeq_{\mathcal{R},0.4} f(g(a), h(a,c))$.

If we took the $\lambda > 0.4$, there would be no solution.

This example is, in fact, a matching problem since variables did not appear in the right side. Now we consider a case when variables appear in both sides.

Example 5.3.2. Assume p is a unary function symbol, q, g, and h are binary, f is ternary, and a, b, c are constants such that:

$$p \sim^{\{(1,1),(1,2)\}}_{\mathcal{R},0.7} q,$$

$$f \sim^{\{(1,1),(2,2),(3,1)\}}_{\mathcal{R},0.6} g,$$

$$f \sim^{\{(1,2),(2,1),(3,2)\}}_{\mathcal{R},0.5} h,$$

$$b \sim^{\varnothing}_{\mathcal{R},0.4} c.$$

Consider the unification problem $P = \{p(x) \simeq^?_{\mathcal{R},0.4} q(g(u,y), h(z,u))\}$. Then $\mathfrak{U}_{\mathit{full}}$ stops with the configuration $S; \sigma; \alpha$ where:

$$S = \{v_1 \simeq^?_{\mathcal{R},0.4} u, v_2 \simeq^?_{\mathcal{R},0.4} y, v_2 \simeq^?_{\mathcal{R},0.4} z, v_3 \simeq^?_{\mathcal{R},0.4} u\},$$

$$\sigma = \{x \mapsto f(v_1, v_2, v_3)\},$$

$$\alpha = 0.5.$$

For illustration, we take three unifiers of P: ϑ_1, ϑ_2, and ϑ_3 together with their approximation degrees β_1, β_2, and β_3, and show how they can be obtained from $S; \sigma$:

1. $\vartheta_1 = \{x \mapsto f(u,z,u), y \mapsto z\}$ and $\beta_1 = 0.5$.

   The instance of $S; \sigma$ under $\varphi = \{y \mapsto z, v_1 \mapsto u, v_2 \mapsto z, v_3 \mapsto u\}$: $S\varphi = \{u \simeq^?_{\mathcal{R},0.4} u, z \simeq^?_{\mathcal{R},0.4} z\}$ and $\sigma\varphi = \{x \mapsto f(u,z,u), y \mapsto z, v_1 \mapsto u, v_2 \mapsto z, v_3 \mapsto u\}$.

   $S\varphi$ is solved and $(\sigma\varphi)|_{\mathcal{V}(P)} = \vartheta_1$. Besides, $\alpha \geqslant \beta_1$.

2. $\vartheta_2 = \{x \mapsto f(u,b,u), y \mapsto b, z \mapsto b\}$ and $\beta_2 = 0.5$.

   The instance of $S; \sigma$ under $\varphi = \{y \mapsto b, z \mapsto b, v_1 \mapsto u, v_2 \mapsto b, v_3 \mapsto u\}$: $S\varphi = \{u \simeq^?_{\mathcal{R},0.4} u, b \simeq^?_{\mathcal{R},0.4} b\}$ and $\sigma\varphi = \{x \mapsto f(u,b,u), y \mapsto b, z \mapsto b, v_1 \mapsto u, v_2 \mapsto b, v_3 \mapsto u\}$.

   $S\varphi$ is solved and $(\sigma\varphi)|_{\mathcal{V}(P)} = \vartheta_2$. Besides, $\alpha \geqslant \beta_2$.

3. $\vartheta_3 = \{x \mapsto f(u,c,u), y \mapsto b, z \mapsto c\}$ and $\beta_3 = 0.4$.

   The instance of $S; \sigma$ under $\varphi = \{v_1 \mapsto u, v_2 \mapsto c, y \mapsto b, z \mapsto c, v_3 \mapsto u\}$: $S\varphi = \{u \simeq^?_{\mathcal{R},0.4} u, c \simeq^?_{\mathcal{R},0.4} b, c \simeq^?_{\mathcal{R},0.4} c\}$ and $\sigma\varphi = \{x \mapsto f(u,c,u), v_1 \mapsto u, v_2 \mapsto c, y \mapsto b, z \mapsto c, v_3 \mapsto u\}$.

   $S\varphi$ is solved, and $(\sigma\varphi)|_{\mathcal{V}(P)} = \vartheta_3$. Besides, $\alpha \geqslant \beta_3$.

This example explains why $\mathfrak{U}_{\mathit{full}}$ stops at variables-only configuration. If it went further from $S; \sigma; \alpha$ as usual and eliminated y, v_1, v_2, v_3 by $\{y \mapsto z, v_1 \mapsto u, v_2 \mapsto z, v_3 \mapsto u\}$, we would end up with the configuration $\emptyset; \{x \mapsto f(u, z, u), y \mapsto z, v_1 \mapsto u, v_2 \mapsto z, v_3 \mapsto u\}$, computing the unifier ϑ_1 as above, but it would not be more general than the unifier ϑ_3. (Recall: more generality is defined by equality, not by proximity.)

5.3.3 Properties of algorithm $\mathfrak{U}_{\mathit{full}}$

Theorem 5.3.1. *The decision problem of $(\mathcal{R}, \lambda)$-unifiability with arity mismatch is NP-hard.*

Proof. By reduction from positive 1-in-3-SAT. Consider the argument correspondence relations[1] $\rho_1 = \{(1,1), (2,2), (3,3)\}$, $\rho_2 = \{(1,3), (2,1), (3,2)\}$, $\rho_3 = \{(1,2), (2,3), (3,1)\}$, and assume $h_i \simeq^{\rho_1}_{\mathcal{R},\lambda} f$ and $h_i \simeq^{\rho_i}_{\mathcal{R},\lambda} g$ for each $1 \leqslant i \leqslant 3$. Then each positive 3-SAT clause $x_1 \vee x_2 \vee x_3$ can be encoded as two proximity equations $y \simeq^{?}_{\mathcal{R},\lambda} f(x_1, x_2, x_3)$ and $y \simeq^{?}_{\mathcal{R},\lambda} g(1, 0, 0)$, where 1 and 0 are constants. Their unifiers force exactly one x to be mapped to 1, and the other two to 0 ($\{y \mapsto h_1(1,0,0), x_1 \mapsto 1, x_2 \mapsto 0, x_3 \mapsto 0\}$, $\{y \mapsto h_2(0,1,0), x_1 \mapsto 0, x_2 \mapsto 1, x_3 \mapsto 0\}$, and $\{y \mapsto h_3(0,0,1), x_1 \mapsto 0, x_2 \mapsto 0, x_3 \mapsto 1\}$). The reduction is polynomial and preserves solvability in both directions. □

Below we state the properties of the algorithm $\mathfrak{U}_{\mathit{full}}$.

Theorem 5.3.2. *$\mathfrak{U}_{\mathit{full}}$ terminates for any input.*

Proof. According to [41], for a syntactic unification problem P, the maximal depth of terms in an mgu of P does not exceed the size of P (i.e., the number of symbols in P, denoted by $\mathit{size}(P)$). Due to the definition of proximity between terms, no proximal mgu can be deeper than a syntactic mgu. Therefore, the same bound applies to $(\mathcal{R}, \lambda)$-unification problems.

Given a variable v and a substitution σ, let $\mathit{md}_\sigma(v)$ be the natural number that denotes the *maximal depth* at which this variable occurs in the range of σ. If v does not appear in the range of σ, then $\mathit{md}_\sigma(v) = 0$.

To an $(\mathcal{R}, \lambda)$-unification configuration $P_i; \sigma_i; \alpha_i$, we associate the multiset $M_i := \{\!\{\mathit{md}_{\sigma_i}(v) \mid v \in \mathcal{V}(P_i)\}\!\}$. Then, for the initial configuration $P_0; \sigma_0; \alpha_0$, where $\sigma_0 = \mathit{Id}$, we have $M_0 = \{\!\{0, \dots, 0\}\!\}$ with $|M_0| = |\mathcal{V}(P_0)|$. These multisets are ordered by the multiset extension $<_{\mathit{mul}}$ of the standard natural

[1]Actually, these relations are total bijective functions.

number ordering, defined in [18] as follows: for two multisets M and M', we say that $M' <_{mul} M$ if $M' = (M \backslash X) \cup Y$, where X and Y are multisets such that $\emptyset \neq X \subseteq M$, and for all $y \in Y$ there exists $x \in X$ with $y < x$.

When **Var-E-U** transforms $P_i; \sigma_i; \alpha_i$ into $P_{i+1}; \sigma_{i+1}; \alpha_{i+1}$ with the substitution $\sigma_{i+1} = \sigma_i\{x \mapsto f(v_1, \ldots, v_m)\}$, we get $\mathcal{V}(P_{i+1}) = \big(\mathcal{V}(P_i) \backslash \{x\}\big) \cup \{v_1, \ldots, v_m\}$ and $md_{\sigma_{i+1}}(v_1) = \cdots = md_{\sigma_{i+1}}(v_m) = 1 + md_{\sigma_i}(x)$. Hence, we have $M_{i+1} = \big(M_i \backslash \{\!\{md_{\sigma_i}(x)\}\!\}\big) \cup \{\!\{md_{\sigma_{i+1}}(v_1), \ldots, md_{\sigma_{i+1}}(v_m)\}\!\}$. Therefore, $M_i <_{mul} M_{i+1}$ after the application of **Var-E-U**.

On the other hand, occurrence cycle check in **Var-E-U** prevents an uncontrolled growth of the multisets. Thus, with each derivation we get the chain $M_0 = \cdots = M_{i_1} <_{mul} M_{i_1+1} = \cdots = M_{i_2} <_{mul} M_{i_2+1} = \cdots <_{mul} \{\!\{size(P) + 1\}\!\}$, where $i_1, i_2 \ldots$ are the steps when **Var-E-U** is used. Since the chain is bounded, **Var-E-U** cannot be applied infinitely often.

From the other rules, **Tri-U** and **Dec-U** do not affect the multisets and strictly decrease $size(P)$. **Var-E-U** may increase the size but, as we said above, it may be applied only finitely many times. Therefore, **Tri-U** and **Dec-U** cannot be applied infinitely often. **Ori-U** does not change the multisets and the size, but strictly decreases the number of equations of the form $t \simeq^?_{\mathcal{R},\lambda} x$, where t is not a variable. The number of such equations may grow after the application of **Dec-U** or **Var-E-U**, but it can happen only finitely many times. Therefore, **Ori-U** cannot be applied infinitely often either. The failure rules stop immediately. Hence, $\mathfrak{U}_{full}$ is terminating. □

Lemma 5.3.3. *Given $\mathcal{R}$, λ, and an $(\mathcal{R}, \lambda)$-unification problem P:*

1. *If $P; \sigma; \alpha \Longrightarrow \bot$ in $\mathfrak{U}_{full}$, then P has no solution.*
2. *If $P; \sigma; \alpha \Longrightarrow P'; \sigma\vartheta; \alpha \wedge \beta$ in $\mathfrak{U}_{full}$ and φ is a solution of P' with the approximation degree γ, then $\vartheta\varphi$ is a solution of P with the approximation degree $\beta \wedge \gamma$.*

Proof. In 1), if the step is made by the **Cla-U** rule, then it is obvious that P has no solution. If the **Occ-U** rule is used, then the theorem follows from Lemma 5.3.2.

To prove 2), we shall consider each non-failing rule. The nontrivial cases are **Dec-U** and **Var-E-U**.

In **Dec-U**, the transformation is $\{f(t_1, \ldots, t_n) \simeq^?_{\mathcal{R},\lambda} g(s_1, \ldots, s_m)\} \uplus P; \sigma; \alpha \Longrightarrow P \cup \{t_i \simeq^?_{\mathcal{R},\lambda} s_j \mid (i, j) \in \rho\}; \sigma; \alpha \wedge \beta$, where $n, m \geqslant 0$, $f \sim^{\rho}_{\mathcal{R},\beta} g$, and $\beta \geqslant \lambda$. If φ is a solution of $P \cup \{t_i \simeq^?_{\mathcal{R},\lambda} s_j \mid (i, j) \in \rho\}$, we have

$deg(P\varphi) \wedge \bigwedge_{(i,j)\in\rho} \mathcal{R}(t_i\varphi, s_j\varphi) = \gamma \geqslant \lambda$. But then $deg(P\varphi) \wedge \mathcal{R}(f(t_1,\dots,t_n)\varphi, g(s_1,\dots,s_m)\varphi) = \beta \wedge \gamma$. Hence, φ is a solution of $\{f(t_1,\dots,t_n) \simeq^?_{\mathcal{R},\lambda} g(s_1,\dots,s_m)\} \uplus P$ with the approximation degree $\beta \wedge \gamma \geqslant \lambda$.

In **Var-E-U**, the step is $\{x \simeq^?_{\mathcal{R},\lambda} g(s_1,\dots,s_n)\} \uplus P; \sigma; \alpha \Longrightarrow P\vartheta \cup \{v_i \simeq^?_{\mathcal{R},\lambda} s_j \mid (i,j) \in \rho\}; \sigma\vartheta; \alpha\wedge\beta$, where $\vartheta = \{x \mapsto f(v_1,\dots,v_m)\}$ with $v_1,\dots,v_m$ fresh variables, and $f \sim^{\rho}_{\mathcal{R},\beta} g$ with $\beta \geqslant \lambda$. If φ solves $P\vartheta \cup \{v_i \simeq^?_{\mathcal{R},\lambda} s_j \mid (i,j) \in \rho\}$, we have $deg(P\vartheta\varphi) \wedge \bigwedge_{(i,j)\in\rho} \mathcal{R}(v_i\varphi, s_j\varphi) = \gamma \geqslant \lambda$. But then $deg(P\vartheta\varphi) \wedge \mathcal{R}(x\vartheta\varphi, g(s_1,\dots,s_n)\vartheta\varphi) = deg(P\vartheta\varphi) \wedge \mathcal{R}(f(v_1,\dots,v_m)\varphi, g(s_1,\dots,s_n)\varphi) = \beta \wedge \gamma$, and thus, φ is a solution of $\{x \simeq^?_{\mathcal{R},\lambda} g(s_1,\dots,s_n)\} \uplus P$ with the approximation degree $\beta \wedge \gamma \geqslant \lambda$. □

Theorem 5.3.3 (Soundness of $\mathfrak{U}_{full}$). *Let $P; Id; 1 \Longrightarrow^* S; \sigma; \alpha$ be a derivation in $\mathfrak{U}_{full}$ where $S; \sigma; \alpha$ is a variables-only configuration. Let φ be a unifier of S with the approximation degree β. Then $\sigma\varphi$ is a unifier of P with the approximation degree $\alpha \wedge \beta$.*

Proof. Induction on the derivation length, using Lemma 5.3.3. □

Theorem 5.3.4 (Completeness of $\mathfrak{U}_{full}$). *Let P be an $(\mathcal{R},\lambda)$-unification problem and ϑ be its unifier with the approximation degree β. Then there exists a derivation $P; Id; 1 \Longrightarrow^* S; \sigma; \alpha$ in $\mathfrak{U}_{full}$, where $S; \sigma; \alpha$ is a variables-only configuration with $\alpha \geqslant \beta$ and there exists a unifier φ of S such that $(\sigma\varphi)|_{\mathcal{V}(P)} = \vartheta|_{\mathcal{V}(P)}$.*

Proof. The existence of a derivation in $\mathfrak{U}_{full}$ that ends in a variables-only configuration follows from Theorem 5.3.2, Lemma 5.3.3 and from the assumption that P is solvable.

We now construct recursively the desired derivation and the substitution φ using ϑ. For the initial configuration $P; Id; 1$ we take $\sigma = Id$, $\alpha = 1$, $\varphi = \vartheta$. Then $\alpha \geqslant \beta$ and $(\sigma\varphi)|_{\mathcal{V}(P)} = \vartheta|_{\mathcal{V}(P)}$ hold. Next, we take $C_0 = \{t \simeq^?_{\mathcal{R},\lambda} s\} \uplus P_0; \sigma_0; \alpha_0$ and assume that φ_0 is a unifier of $\{t \simeq^?_{\mathcal{R},\lambda} s\} \uplus P_0$ such that $(\sigma_0\varphi_0)|_{\mathcal{V}(P)} = \vartheta|_{\mathcal{V}(P)}$ and $\alpha_0 \geqslant \beta$. We prove that there exists a configuration $C_1 = P_1; \sigma_1; \alpha_1$ and a unifier φ_1 of P_1 such that $C_0 \Longrightarrow C_1$, $(\sigma_1\varphi_1)|_{\mathcal{V}(P)} = \vartheta|_{\mathcal{V}(P)}$ and $\alpha_1 \geqslant \beta$.

From the four non-failing rules that can perform the step $C_0 \Longrightarrow C_1$ only **Var-E-U** with $t = x$ and $s = g(s_1,\dots,s_n)$ is non-trivial. Since φ_0 solves $\{x \simeq^?_{\mathcal{R},\lambda} g(s_1,\dots,s_n)\} \uplus P_0$, we have $x\varphi_0 = f(r_1,\dots,r_m)$ for an f with $f \sim^{\rho}_{\mathcal{R},\beta_1} g$ and $r_i \simeq_{\mathcal{R},\lambda} s_j$ for all $(i,j) \in \rho$. Note that $\beta_1 \geqslant \beta$. Then $P_1 = \{v_i \simeq^?_{\mathcal{R},\lambda} s_j \mid (i,j) \in \rho\} \cup P_0\psi$, $\sigma_1 = \sigma_0\psi$ where $\psi = \{x \mapsto f(v_1,\dots,v_m)\}$, and $\alpha_1 = \alpha_0 \wedge \beta_1$. Take $\varphi_1 = \nu\varphi_0$, where $\nu = \{v_i \mapsto r_i \mid 1 \leqslant i \leqslant m\}$. Then φ_1 solves P_1, since (i) $y\varphi_1 = y\varphi_0$ for all $y \in \mathcal{V}(P_0)\backslash\{x\}$; (ii) $v_k\varphi_1 = r_k$ for all $1 \leqslant k \leqslant m$; and (iii) $v_i\varphi_1 \simeq_{\mathcal{R},\lambda} s_j$ for those v_i's for which $(i,j) \in \rho$. Moreover, $(\sigma_1\varphi_1)|_{\mathcal{V}(P)} = (\sigma_0\psi\nu\varphi_0)|_{\mathcal{V}(P)} = (\sigma_0\{x \mapsto x\varphi_0\}\nu\varphi_0)|_{\mathcal{V}(P)} = (\sigma_0\{x \mapsto x\varphi_0\}\varphi_0\nu)|_{\mathcal{V}(P)} = (\sigma_0\varphi_0\nu)|_{\mathcal{V}(P)} = (\sigma_0\varphi_0)|_{\mathcal{V}(P)} = \vartheta|_{\mathcal{V}(P)}$. Finally, $\alpha_1 \geqslant \beta$, because $\alpha_0 \geqslant \beta$, $\beta_1 \geqslant \beta$, and $\alpha_1 = \alpha_0 \wedge \beta_1$. □

5.4 Matching in fully fuzzy signatures

For the matching with arity mismatch problem we do not need restrictions on argument relations. This leads to a peculiarity, namely the matchers do not have to be ground substitutions. For instance, if f is binary, and g and h are unary symbols with $f \sim^{\{(2,1)\}}_{\mathcal{R},0.7} g$ and $f \sim^{\{(2,1)\}}_{\mathcal{R},0.6} h$, then $\sigma = \{x \mapsto f(y,a)\}$ is a matcher of $f(x,x) \lesssim_{\mathcal{R},0.5} f(g(a),h(a))$ with the degree 0.6. In fact, σ is more general than any other solution of this problem. Analogously to unification, we will use the notion of most general solution for matching problems and write $(\mathcal{R},\lambda)$-*mgm* for most general $(\mathcal{R},\lambda)$-matchers.

In this section we need to extend the definition of the $(\mathcal{R},\lambda)$-proximity class $\mathbf{pc}_{\mathcal{R},\lambda}(s)$ of a term s, to also include the argument relations, as follows:

$$\begin{aligned}
&\mathbf{pc}_{\mathcal{R},\lambda}(x) := \{x\}.\\
&\mathbf{pc}_{\mathcal{R},\lambda}(g(s_1,\dots,s_m)) := \big\{f(t_1,\dots,t_n) \,\big|\\
&\quad g \sim^{\rho}_{\mathcal{R},\beta} f,\ \beta \geqslant \lambda,\ f \text{ is } n\text{-ary, and for each } 1 \leqslant j \leqslant n,\\
&\qquad t_j \in \mathbf{pc}_{\mathcal{R},\lambda}(s_i),\ \text{if } (i,j) \in \rho,\\
&\qquad \text{or } t_j = v,\ \text{if for no } i,\ 1 \leqslant i \leqslant m,\ (i,j) \in \rho,\\
&\qquad \text{where } v \text{ is a fresh variable}\big\}.
\end{aligned}$$

We will also need the operation $\Cap$ of merging two terms, defined as:

(i) $x \Cap t = t \Cap x := t$ for any variable x and term t;

(ii) $f(t_1,\dots,t_n) \Cap f(s_1,\dots,s_n) := f(t_1 \Cap s_1,\dots,t_n \Cap s_n)$, $n \geqslant 0$;

(iii) $t \Cap s$ is undefined in any other case.

5.4.1 Matching rules

To solve a matching problem $t \lesssim^{?}_{\mathcal{R},\lambda} s$, we create the triple $\{t \lesssim^{?}_{\mathcal{R},\lambda} s\}; \varnothing; 1$ and apply the rules below. They work on triples $M; S; \alpha$, where M is a set of matching equations to be solved, S is the set of solved equations of the form $x \approx s$, and α is the approximation degree computed so far.

The matching rules are the following:

Dec-M: Decomposition

$$\{f(t_1, \dots, t_n) \lesssim^?_{\mathcal{R},\lambda} g(s_1, \dots, s_m)\} \uplus M; S; \alpha \Longrightarrow$$
$$M \cup \{t_i \lesssim^?_{\mathcal{R},\lambda} s_j \mid (i,j) \in \rho\}; S; \alpha \wedge \beta,$$

if $n, m \geqslant 0$ and $f \sim^{\rho}_{\mathcal{R},\beta} g$ with $\beta \geqslant \lambda$.

Var-E-M: Variable elimination

$$\{x \lesssim^?_{\mathcal{R},\lambda} s\} \uplus M; S; \alpha \Longrightarrow M; S \cup \{x \approx t\}; \alpha \wedge \beta,$$

where $t \in \mathbf{pc}_{\mathcal{R},\lambda}(s)$ and $\mathcal{R}(t, s) = \beta \geqslant \lambda$.

Mer-M: Merging

$$M; \{x \approx t, x \approx s\} \uplus S; \alpha \Longrightarrow M; S \cup \{x \approx t \Cap s\}; \alpha, \qquad \text{if } t \Cap s \text{ is defined.}$$

Cla-M: Clash

$$\{f(t_1, \dots, t_n) \lesssim^?_{\mathcal{R},\lambda} g(s_1, \dots, s_m)\} \uplus M; S; \alpha \Longrightarrow \perp, \qquad \text{if } \mathcal{R}(f, g) < \lambda.$$

Inc-M: Inconsistency

$$M; \{x \approx t,\ x \approx s\} \uplus S; \alpha \Longrightarrow \perp, \qquad \text{if } t \Cap s \text{ is undefined.}$$

The matching algorithm $\mathfrak{M}_{\mathit{full}}$ uses these rules to transform triples as long as possible, returning either $\perp$ (indicating failure), or $\emptyset; S; \alpha$ (indicating success). In the latter case, each variable occurs in S at most once. Therefore, from S one can obtain a substitution $\sigma_S := \{x \mapsto s \mid x \approx s \in S\}$. We call it the *computed 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 $x \approx t \in S$, we have $x\sigma = t$. We also assume that $\perp$ has no solution.

5.4.2 Example

Example 5.4.1. Assume p, g and h are unary symbols, q is binary, f is ternary, and a, b, and c are constants such that:

$$p \sim^{\{(1,1),(1,2)\}}_{\mathcal{R},0.7} q,$$

$$f \sim^{\{(1,1)\}}_{\mathcal{R},0.6} g,$$

$$f \sim^{\{(3,1)\}}_{\mathcal{R},0.5} h,$$

$b \sim^{\emptyset}_{\mathcal{R},0.4} c$.

We consider the matching problem $p(x) \lesssim^{?}_{\mathcal{R},0.4} q(g(a), h(c))$ and show only the successful derivations. They start with:

$$\{p(x) \lesssim^{?}_{\mathcal{R},0.4} q(g(a), h(c))\}; \emptyset; 1 \Longrightarrow_{\text{Dec-M}}$$
$$\{x \lesssim^{?}_{\mathcal{R},0.4} g(a),\ x \lesssim^{?}_{\mathcal{R},0.4} h(c)\}; \emptyset; 0.7 \Longrightarrow_{\text{Var-E-M}}$$
$$\{x \lesssim^{?}_{\mathcal{R},0.4} h(c)\}; \{x \approx f(a, v_1, v_2)\}; 0.6$$

and then continue by **Var-E-M** in two different ways:

$$1{:}\ \emptyset; \{x \approx f(a, v_1, v_2), x \approx f(v_3, v_4, c)\}; 0.5 \Longrightarrow_{\text{Mer-M}}$$
$$\emptyset; \{x \approx f(a, v_1, c)\}; 0.5.$$
$$2{:}\ \emptyset; \{x \approx f(a, v_1, v_2), x \approx f(v_3, v_4, b)\}; 0.4 \Longrightarrow_{\text{Mer-M}}$$
$$\emptyset; \{x \approx f(a, v_1, b)\}; 0.4.$$

The computed substitutions $\{x \mapsto f(a, v_1, c)\}$ and $\{x \mapsto f(a, v_1, b)\}$ are matchers of the original problem with the approximation degrees 0.5 and 0.4, respectively.

5.4.3 Properties of algorithm $\mathfrak{M}_{full}$

Remark 5.4.1. In Theorem 5.3.1, we proved the NP-hardness of unification. NP-hardness can be also shown for *well-moded unification problems*. They are special unification problems in which the equations can be ordered as $t_0 \simeq_{\mathcal{R},\lambda} s_0, \dots, t_n \simeq_{\mathcal{R},\lambda} s_n$, with s_0 ground and $\mathcal{V}(s_i) \subseteq \cup_{j=0}^{i-1} \mathcal{V}(t_j)$, $1 \leqslant i \leqslant n$. Hence, $t_0 \simeq_{\mathcal{R},\lambda} s_0$ is actually a matching problem $t_0 \lesssim^{?}_{\mathcal{R},\lambda} s_0$. If we solve these equations from left to right, the s's get ground as we move. Thus, we will encounter only matching equations. The encoding in the proof of Theorem 5.3.1 can be expressed as a well-moded unification problem, written as matching equations $y \lesssim^{?}_{\mathcal{R},\lambda} g(1, 0, 0), f(x_1, x_2, x_3) \lesssim^{?}_{\mathcal{R},\lambda} y$. Hence, the decision problem for well-moded proximity unification is NP-hard.

Lemma 5.4.1. *Let $M_1; S_1; \alpha \Longrightarrow M_2; S_2; \alpha \wedge \beta$ be a step made by $\mathfrak{M}_{full}$. If ϑ is an $(\mathcal{R}, \lambda)$-solution of $M_2; S_2$ with the approximation degree γ, then ϑ is an $(\mathcal{R}, \lambda)$-solution of $M_1; S_1$ with the approximation degree $\beta \wedge \gamma$.*

Proof. By the definition of a solution, the lemma holds for **Dec-M** and **Cla-M**. The definition of $\Cap$ implies it for **Mer-M** and **Inc-M**. For **Var-E-M**, by the definition of **pc**, we have $x\vartheta \in \mathbf{pc}_{\mathcal{R},\lambda}(s)$ iff $\mathcal{R}(x\vartheta, s) = \beta \geqslant \lambda$, which implies that ϑ with $x\vartheta = t$ is an $(\mathcal{R},\lambda)$-matcher of $x \lesssim^{?}_{\mathcal{R},\lambda} s$ with the degree β. Therefore ϑ is a solution of $M_1; S_1$ with the degree $\beta \wedge \gamma$. □

Theorem 5.4.1. *Given an $(\mathcal{R},\lambda)$-matching problem $t \lesssim^{?}_{\mathcal{R},\lambda} s$, the matching algorithm $\mathfrak{M}_{full}$ terminates and computes a substitution σ that is an $(\mathcal{R},\lambda)$-matcher of t to s.*

Proof. We prove termination and soundness separately.

Termination. The rules **Dec-M** and **Var-E-M** strictly reduce the size of M. **Mer-M** does the same for S, without changing M. **Cla-M** and **Inc-M** stop immediately. Hence, $\mathfrak{M}_{full}$ strictly reduces the lexicographic combination $\langle size(M), size(S)\rangle$ of sizes of M and S, which implies termination.

Soundness. Let $\{t \lesssim^{?}_{\mathcal{R},\lambda} s\}; \emptyset; 1 \Longrightarrow^{+} \emptyset; S; \alpha$ be the derivation in $\mathfrak{M}_{full}$ that computes σ. Then σ is a solution of $\emptyset; S$. By induction on the length of the derivation, using Lemma 5.4.1, we can prove that σ is an $(\mathcal{R},\lambda)$-matcher of t to s. □

Theorem 5.4.2. *Given an $(\mathcal{R},\lambda)$-matching problem M and its solution ϑ, the algorithm $\mathfrak{M}_{full}$ computes a substitution σ such that $x\vartheta \Cap x\sigma = x\vartheta$ for all x in M.*

Proof. This theorem essentially says that for an x occurring in the matching problem, if $r_1 = x\vartheta$ and $r_2 = x\sigma$, then r_1 and r_2 have exactly same structure (otherwise $r_1 \Cap r_2$ would not be defined) and they may differ from each other only at those positions where r_2 contains a variable.

We need to construct a derivation that computes σ. The construction will be guided by ϑ. The only steps in the derivation that take into account ϑ are **Var-E-M** steps. When we transform $\{x \lesssim^{?}_{\mathcal{R},\lambda} s\} \uplus M; S; \alpha$ to $M; S \cup \{x \approx t\}; \alpha \wedge \beta$, we will construct t based on $x\vartheta$: if p is a position in t where by the definition of $\mathbf{pc}_{\mathcal{R},\lambda}(s)$ we should have a function symbol, then this symbol is chosen as the one that appears in $x\vartheta$ at position p. The fact that ϑ is a solution for M ensures that the chosen symbol is included in $\mathbf{pc}_{\mathcal{R},\lambda}(s)$. When t does not need to have a function symbol in p, it will have a variable, and such positions do not play a role in the proximity of t with s. All equations for the same x in S are constructed in this way. **Mer-M** merges them by replacing some variables by terms in $x\vartheta$. Therefore, if a subterm r occurring at position p in $x\sigma$ differs from the subterm at the same position p in $x\vartheta$, then r is a variable. Hence, $x\vartheta \Cap x\sigma = x\vartheta$ for all x in M. □

Corollary 5.4.2.1. *Given an $(\mathcal{R}, \lambda)$-matching problem M and its solution ϑ, the algorithm $\mathfrak{M}_{full}$ computes a substitution σ such that $x\sigma \lesssim_{\mathcal{R},\lambda} x\vartheta$ for all x in M.*

5.5 Anti-unification in fully fuzzy signatures

Computing an $(\mathcal{R}, \lambda)$-generalization of some terms t and s, with fully fuzzy signatures, raises some unattainable challenges – as it will be seen below – if we formulate the problem as we did in the previous chapters. Therefore, we will consider in the following section slightly modified variants of the problem, that allow us to generalize the terms. We start with extending some of the notions that were previously used.

5.5.1 Additional notions

In this section we extend the set of variables $\mathcal{V}$, with a special symbol $\_$ (the anonymous variable). The set of fixed arity function symbols $\mathcal{F}$ stays the same. The set of non-anonymous variables $\mathcal{V}\backslash\{\_\}$ is denoted by $\mathcal{V}^-$. When the set of variables is not explicitly specified, we mean $\mathcal{V}$.

The set of terms $\mathcal{T}(\mathcal{F}, \mathcal{V})$ over $\mathcal{F}$ and $\mathcal{V}$ is defined in the standard way, with $\_$ being included in $\mathcal{V}$. Terms over $\mathcal{T}(\mathcal{F}, \mathcal{V}^-)$ are defined similarly, except that all variables are taken from $\mathcal{V}^-$. For a term t, $\mathcal{V}(t)$ remains the notation for the set of all variables, while $\mathcal{V}^-(t)$ denotes the set of all non-anonymous variables appearing in t. By default, variable means an element of $\mathcal{V}$ and term means an element of $\mathcal{T}(\mathcal{F}, \mathcal{V})$. We will make it explicit when we will be talking about variables from $\mathcal{V}^-$ and terms from $\mathcal{T}(\mathcal{F}, \mathcal{V}^-)$. A term is called *linear* in this context if no non-anonymous variable occurs in it more than once.

The deanonymization operation deanon replaces each occurrence of the anonymous variable in a term by a fresh variable. For instance, we have $\mathsf{deanon}(f(\_, x, g(\_))) = f(y', x, g(y'')))$, where y' and y'' are fresh. Hence, $\mathsf{deanon}(t) \in \mathcal{T}(\mathcal{F}, \mathcal{V}^-)$ is unique up to variable renaming for all $t \in \mathcal{T}(\mathcal{F}, \mathcal{V})$. If t is linear, then $\mathsf{deanon}(t)$ is linear as well and vice versa.

We restrict the definition of a *substitution* as being a mapping from $\mathcal{V}^-$ to $\mathcal{T}(\mathcal{F}, \mathcal{V}^-)$ (i.e., without anonymous variables), which is the identity almost everywhere. We also extend the definition for the *application* of a substitution σ to a term t, by adding the case $\_\sigma := \_$.

5.5.2 Generalizations

In the new context, which includes the anonymous variable, we have to adjust some of the definitions regarding generalizations.

Given $\mathcal{R}$ and λ, a term r is an $(\mathcal{R}, \lambda)$-*generalization* of (alternatively, $(\mathcal{R}, \lambda)$-*more general than*) a term t, written as $r \lesssim_{\mathcal{R},\lambda} t$, if there exists a substitution σ such that $\mathsf{deanon}(r)\sigma \simeq_{\mathcal{R},\lambda} \mathsf{deanon}(t)$. The strict part of $\lesssim_{\mathcal{R},\lambda}$ is denoted by $\prec_{\mathcal{R},\lambda}$, i.e., $r \prec_{\mathcal{R},\lambda} t$ if $r \lesssim_{\mathcal{R},\lambda} t$ and not $t \lesssim_{\mathcal{R},\lambda} r$.

Example 5.5.1. Given a proximity relation $\mathcal{R}$, a cut value $\lambda = 0.3$, constants $a \sim^{\varnothing}_{\mathcal{R},0.4} b$ and $b \sim^{\varnothing}_{\mathcal{R},0.5} c$, binary function symbols f and h, and a unary function symbol g such that $h \sim^{\{(1,1),(1,2)\}}_{\mathcal{R},0.5} f$ and $h \sim^{\{(1,1)\}}_{\mathcal{R},0.6} g$, we have

- $h(x, \_) \lesssim_{\mathcal{R},\lambda} h(a, x)$, because $h(x, x')\{x \mapsto a, x' \mapsto x\} = h(a, x) \simeq_{\mathcal{R},\lambda} h(a, x)$.
- $h(x, \_) \lesssim_{\mathcal{R},\lambda} h(\_, x)$, because $h(x, x')\{x \mapsto y', x' \mapsto x\} = h(y', x) \simeq_{\mathcal{R},\lambda} h(y', x)$.
- $h(x, x) \not\lesssim_{\mathcal{R},\lambda} h(\_, x)$, because $h(x, x) \not\lesssim_{\mathcal{R},\lambda} h(y', x)$.
- $h(x, \_) \lesssim_{\mathcal{R},\lambda} f(a, c)$, because $h(x, x')\{x \mapsto b\} = h(b, x') \simeq_{\mathcal{R},\lambda} f(a, c)$, since $\mathcal{R}(h(b, x'), f(a, c)) = 0.4$.
- $h(x, \_) \lesssim_{\mathcal{R},\lambda} g(c)$, because $h(x, x')\{x \mapsto c\} = h(c, x') \simeq_{\mathcal{R},\lambda} g(c)$, since $\mathcal{R}(h(c, x'), g(c)) = 0.6$.

As a reminder, the notion of *syntactic generalization* of a term is a special case of $(\mathcal{R}, \lambda)$-generalization for $\lambda = 1$. We write $r \lesssim t$ to indicate that r is a syntactic generalization of t. Its strict part is denoted by $\prec$.

Recall that $\mathcal{R}$ is assumed to be *strict*: for all $w_1, w_2 \in \mathcal{F} \cup \mathcal{V}$, if $\mathcal{R}(w_1, w_2) = 1$, then $w_1 = w_2$. When $\lambda = 1$, the relation $\simeq_{\mathcal{R},\lambda}$ does not depend on $\mathcal{R}$ due to strictness of the latter and is just the syntactic equality $=$. Since $\mathcal{R}$ is strict, $r \lesssim t$ is equivalent to $\mathsf{deanon}(r)\sigma = \mathsf{deanon}(t)$ for some σ (note the syntactic equality here).

Theorem 5.5.1. *If $r \lesssim t$ and $t \lesssim_{\mathcal{R},\lambda} s$, then $r \lesssim_{\mathcal{R},\lambda} s$.*

Proof. $r \lesssim t$ implies $\mathsf{deanon}(r)\sigma = \mathsf{deanon}(t)$ for some σ, while from $t \lesssim_{\mathcal{R},\lambda} s$ we have $\mathsf{deanon}(t)\vartheta \simeq_{\mathcal{R},\lambda} \mathsf{deanon}(s)$ for some ϑ. Then $\mathsf{deanon}(r)\sigma\vartheta \simeq_{\mathcal{R},\lambda} \mathsf{deanon}(s)$, which implies $r \lesssim_{\mathcal{R},\lambda} s$. □

Note that $r \lesssim_{\mathcal{R},\lambda} t$ and $t \lesssim_{\mathcal{R},\lambda} s$, in general, do not imply $r \lesssim_{\mathcal{R},\lambda} s$ due to non-transitivity of $\simeq_{\mathcal{R},\lambda}$. A simple counterexample: $a \lesssim_{\mathcal{R},0.3} b$, $b \lesssim_{\mathcal{R},0.3} c$, but not $a \lesssim_{\mathcal{R},0.3} c$.

Ideally, the precise formulation of anti-unification problem would be like in the previous chapters: Given $\mathcal{R}$, λ, t_1 and t_2, find an $(\mathcal{R},\lambda)$-lgg r of t_1 and t_2, some substitutions σ_1 and σ_2, and the approximation degrees α_1 and α_2 such that $\mathcal{R}(r\sigma_1, t_1) = \alpha_1$ and $\mathcal{R}(r\sigma_2, t_2) = \alpha_2$. A minimal and complete algorithm to solve this problem would compute exactly the elements of $(\mathcal{R},\lambda)$-mcsg of t_1 and t_2 together with their approximation degrees. However, as we see below, it is problematic to solve the problem in this form. Therefore, we will consider a slightly modified variant, taking into account anonymous variables in generalizations and relaxing bounds on approximation degrees.

We assume that the terms to be generalized are ground. It is not a restriction because we can treat variables in them as constants that are close only to themselves.

Recall that the proximity class of any alphabet symbol is finite. Also, the symbols are related to each other by finitely many argument relations. One may think that it leads to finite proximity classes of terms, but this is not the case. Consider, e.g., $\mathcal{R}$ and λ, where $h \simeq_{\mathcal{R},\lambda}^{\{(1,1)\}} f$ with binary h and unary f. Then the $(\mathcal{R},\lambda)$-proximity class of $f(a)$ is infinite: $\{f(a)\} \cup \{h(a,t) \mid t \in \mathcal{T}(\mathcal{F},\mathcal{V})\}$. Also, the $(\mathcal{R},\lambda)$-mcsg for $f(a)$ and $f(b)$ is infinite: $\{f(x)\} \cup \{h(x,t) \mid t \in \mathcal{T}(\mathcal{F},\varnothing)\}$.

Definition 5.5.1. *Given the terms $t_1,\dots,t_n$, $n \geqslant 1$, a position p in a term r is called* irrelevant for $(\mathcal{R},\lambda)$-generalizing *(resp.* for $(\mathcal{R},\lambda)$-proximity to*) $t_1,\dots,t_n$ if $r[s]_p \lesssim_{\mathcal{R},\lambda} t_i$ (resp. $r[s]_p \simeq_{\mathcal{R},\lambda} t_i$) for all $1 \leqslant i \leqslant n$ and for all terms s.*

We say that r is a relevant $(\mathcal{R},\lambda)$-generalization *(resp.* relevant $(\mathcal{R},\lambda)$-proximal term*) of $t_1,\dots,t_n$ if $r \lesssim_{\mathcal{R},\lambda} t_i$ (resp. $r \simeq_{\mathcal{R},\lambda} t_i$) for all $1 \leqslant i \leqslant n$ and $r|_p = \_$ for all positions p in r that is irrelevant for generalizing (resp. for proximity to) $t_1,\dots,t_n$. The* $(\mathcal{R},\lambda)$-relevant proximity class *of t is*

$$\mathbf{rpc}_{\mathcal{R},\lambda}(t) := \{s \mid s \textit{ is a relevant } (\mathcal{R},\lambda)\textit{-proximal term of } t\}.$$

In the example above, position 2 in $h(x,t)$ is irrelevant for generalizing $f(a)$ and $f(b)$, and $h(x,\_)$ is one of their relevant generalizations. Note that $f(x)$ is also a relevant generalization of $f(a)$ and $f(b)$, since it contains no irrelevant positions. More general generalizations like, e.g., x, are relevant as well. Similarly, position 2 in $h(a,t)$ is irrelevant for proximity to $f(a)$

and $\mathbf{rpc}_{\mathcal{R},\lambda}(f(a)) = \{f(a), h(a, \_)\}$. Generally, $\mathbf{rpc}_{\mathcal{R},\lambda}(t)$ is finite for any t due to the finiteness of proximity classes of symbols and argument relations mentioned above.

Definition 5.5.2 (Minimal complete set of relevant $(\mathcal{R}, \lambda)$-generalizations)**.** *Given $\mathcal{R}$, λ, t_1, and t_2, a set of terms T is a* complete set of relevant $(\mathcal{R}, \lambda)$-generalizations *of t_1 and t_2 if*

(a) every element of T is a relevant $(\mathcal{R}, \lambda)$-generalization of t_1 and t_2, and

(b) if r is a relevant $(\mathcal{R}, \lambda)$-generalization of t_1 and t_2, then there exists $r' \in T$ such that $r \lesssim r'$.

The minimality property is defined as usual (see Definition 2.3.3).

This definition directly extends to relevant generalizations of finitely many terms. We use $(\mathcal{R}, \lambda)$-mcsrg as an abbreviation for minimal complete set of relevant $(\mathcal{R}, \lambda)$-generalization. Like relevant proximity classes, mcsrg's are also finite.

Lemma 5.5.1. *For given $\mathcal{R}$ and λ, if all argument relations are correspondence relations, then $(\mathcal{R}, \lambda)$-mcsg's and $(\mathcal{R}, \lambda)$-proximity classes for all terms are finite.*

Proof. Under correspondence relations no term contains an irrelevant position for generalization or for proximity. □

Hence, for correspondence relations the notions of mcsg and mcsrg coincide, as well as the notions of proximity class and relevant proximity class.

For a term r, we define its *linearized version* $\mathsf{lin}(r)$ as a term obtained from r by replacing each occurrence of a non-anonymous variable in r by a fresh one. Linearized versions of terms are unique modulo variable renaming. For instance, $\mathsf{lin}(f(x, \_, g(y, x, a), b)) = f(x', \_, g(y', x'', a), b)$, where x', x'', and y' are fresh variables.

Definition 5.5.3 (Generalization degree upper bound)**.** *Given two terms r and t, a proximity relation $\mathcal{R}$, and a λ-cut, the $(\mathcal{R}, \lambda)$*-generalization degree upper bound of r and t, *denoted by $\mathsf{gdub}_{\mathcal{R},\lambda}(r, t)$, is defined as follows:*

Let $\alpha := \max\{\mathcal{R}(\mathsf{lin}(r)\sigma, t) \mid \sigma$ is a substitution$\}$. Then

$$\mathsf{gdub}_{\mathcal{R},\lambda}(r, t) = \begin{cases} \alpha, & \text{if } \alpha \geqslant \lambda, \\ 0, & \text{otherwise.} \end{cases}$$

Intuitively, $\mathsf{gdub}_{\mathcal{R},\lambda}(r,t) = \alpha$ means that no instance of r can get closer than $\alpha \geqslant \lambda$ to t in $\mathcal{R}$. From the definition it follows that if $r \lesssim_{\mathcal{R},\lambda} t$, then $0 < \lambda \leqslant \mathsf{gdub}_{\mathcal{R},\lambda}(r,t) \leqslant 1$ and if $r \not\lesssim_{\mathcal{R},\lambda} t$, then $\mathsf{gdub}_{\mathcal{R},\lambda}(r,t) = 0$. The upper bound computed by gdub is more relaxed than it would be if the linearization function were not used, but this is what we will be able to compute in our algorithms later.

Example 5.5.2. Let $\mathcal{R}(a,b) = 0.6$, $\mathcal{R}(b,c) = 0.7$, and $\lambda = 0.5$. Then:

$$\mathsf{gdub}_{\mathcal{R},\lambda}(f(x,b), f(a,c)) = 0.7 \text{ and}$$
$$\mathsf{gdub}_{\mathcal{R},\lambda}(f(x,x), f(a,c)) = \mathsf{gdub}_{\mathcal{R},\lambda}(f(x,y), f(a,c)) = 1.$$

It is not difficult to see that if σ is a substitution such that $r\sigma \simeq_{\mathcal{R},\lambda} t$, then $\mathcal{R}(r\sigma,t) \leqslant \mathsf{gdub}_{\mathcal{R},\lambda}(r,t)$. In Example 5.5.2, for $\sigma = \{x \mapsto b\}$ we have $\mathcal{R}(f(x,x)\sigma, f(a,c)) = \mathcal{R}(f(b,b), f(a,c)) = 0.6 < \mathsf{gdub}_{\mathcal{R},\lambda}(f(x,x), f(a,c)) = 1$.

Given $r \preceq_{\mathcal{R},\lambda} t$, we can compute $\mathsf{gdub}_{\mathcal{R},\lambda}(r,t)$ as follows:

- If r is a variable, then $\mathsf{gdub}_{\mathcal{R},\lambda}(r,t) = 1$.
- Otherwise, if $head(r) \sim^{\rho}_{\mathcal{R},\beta} head(t)$, then
$$\mathsf{gdub}_{\mathcal{R},\lambda}(r,t) = \beta \wedge \bigwedge_{(i,j)\in\rho} \mathsf{gdub}_{\mathcal{R},\lambda}(r|_i, t|_j).$$
- Otherwise, $\mathsf{gdub}_{\mathcal{R},\lambda}(r,t) = 0$.

5.5.3 Term set consistency

The notion of term set consistency plays an important role in the computation of proximal generalizations. Intuitively, a set of terms is $(\mathcal{R},\lambda)$-consistent if all the terms in the set have a common $(\mathcal{R},\lambda)$-proximal term. In this section, we discuss this notion and the corresponding algorithms.

Definition 5.5.4 (Consistent set of terms)**.** *A finite set of terms T is* $(\mathcal{R},\lambda)$-consistent *if there exists a term s such that $s \simeq_{\mathcal{R},\lambda} t$ for all $t \in T$.*

$(\mathcal{R},\lambda)$-consistency of a finite term set T is equivalent to $\bigcap_{t\in T} \mathbf{pc}_{\mathcal{R},\lambda}(t) \neq \emptyset$, but we cannot use this property to decide consistency, since proximity classes of terms can be infinite (when the argument relations are not restricted). For this reason, we introduce the operation $\sqcap$ on terms as follows: (i) $t \sqcap \_ = \_ \sqcap t = t$, (ii) $f(t_1, \dots, t_n) \sqcap f(s_1, \dots, s_n) = f(t_1 \sqcap s_1, \dots, t_n \sqcap s_n)$, $n \geqslant 0$. Obviously, $\sqcap$ is associative (A), commutative (C), idempotent (I), and has $\_$ as its unit element (U). It can be extended to sets of terms: $T_1 \sqcap T_2 := \{t_1 \sqcap t_2 \mid t_1 \in T_1, t_2 \in T_2\}$. It is easy to see that $\sqcap$ on sets also satisfies the ACIU properties with the set $\{\_\}$ playing the role of the unit element.

Lemma 5.5.2. *A finite set of terms T is $(\mathcal{R}, \lambda)$-consistent if and only if $\bigsqcap_{t \in T} \mathbf{rpc}_{\mathcal{R},\lambda}(t) \neq \emptyset$.*

Proof. ($\Rightarrow$) If $s \simeq_{\mathcal{R},\lambda} t$ for all $t \in T$, then $s_t \in \mathbf{rpc}_{\mathcal{R},\lambda}(t)$, where s_t is obtained from s by replacing all subterms that are irrelevant for the $(\mathcal{R}, \lambda)$-proximity to t by the anonymous variable. Assume $T = \{t_1, \dots, t_n\}$. Then we have $s_{t_1} \sqcap \cdots \sqcap s_{t_n} \in \bigsqcap_{t \in T} \mathbf{rpc}_{\mathcal{R},\lambda}(t)$.

($\Leftarrow$) Obvious, since for $s \in \bigsqcap_{t \in T} \mathbf{rpc}_{\mathcal{R},\lambda}(t)$ we have $s \simeq_{\mathcal{R},\lambda} t$ for all $t \in T$. □

Now we design an algorithm $\mathfrak{C}$ that computes $\bigsqcap_{t \in T} \mathbf{rpc}_{\mathcal{R},\lambda}(t)$ without actually computing $\mathbf{rpc}_{\mathcal{R},\lambda}(t)$ for each $t \in T$. A special version of the algorithm can be used to decide the $(\mathcal{R}, \lambda)$-consistency of T.

The algorithm is rule-based. The rules work on states, that are pairs $\mathbf{T}; s$, where s is a term and $\mathbf{T}$ is a finite set of expressions of the form $x \mathsf{~in~} T$, where T is a finite set of terms. $\mathcal{R}$ and λ are given. There are two rules ($\uplus$ stands for disjoint union):

Rem: Removing the empty set

$\{x \mathsf{~in~} \emptyset\} \uplus \mathbf{T}; s \Longrightarrow \mathbf{T}; s\{x \mapsto \_\}$.

Red: Reduce a set to new sets

$\{x \mathsf{~in~} \{t_1, \dots, t_m\}\} \uplus \mathbf{T}; s \Longrightarrow$
$\quad \{y_1 \mathsf{~in~} T_1, \dots, y_n \mathsf{~in~} T_n\} \cup \mathbf{T}; s\{x \mapsto h(y_1, \dots, y_n)\}$,

where $m > 1$, h is an n-ary function symbol such that $h \sim^{\rho_k}_{\mathcal{R},\gamma_k} head(t_k)$ with $\gamma_k \geqslant \lambda$ for all $1 \leqslant k \leqslant m$, and $T_i := \{t_k|_j \mid (i,j) \in \rho_k, 1 \leqslant k \leqslant m\}$, $1 \leqslant i \leqslant n$, is the set of all those arguments of the terms $t_1, \dots, t_m$ that are supposed to be $(\mathcal{R}, \lambda)$-proximal to the i's argument of h.

To compute $\bigsqcap_{t \in T} \mathbf{rpc}_{\mathcal{R},\lambda}(t)$, the algorithm $\mathfrak{C}$ starts with $\{x \mathsf{~in~} T\}; x$ and applies the rules as long as possible. Red cause branching. A state of the form $\varnothing; s$ is called a success state. A failure state is a state of the form $\mathbf{T}; s$ to which no rule applies and $\mathbf{T} \neq \varnothing$. In the full derivation tree, each leaf is a either success or a failure state.

Theorem 5.5.2. *Given a finite set of terms T, the algorithm $\mathfrak{C}$ always terminates starting from the state $\{x \textsf{ in } T\}; x$ (where x is a fresh variable). If S is the set of success states produced at the end, we have $\{s \mid \varnothing; s \in S\} = \bigsqcap_{t\in T} \mathbf{rpc}_{\mathcal{R},\lambda}(t)$.*

Proof. Termination: Associate to each state $\{x_1 \textsf{ in } T_1, \dots x_n \textsf{ in } T_n\}; s$ the multiset $\{d_1, \dots, d_n\}$, where d_i is the maximum depth of terms occurring in T_i. $d_i = 0$ if $T_i = \varnothing$. Compare these multisets by the Dershowitz-Manna ordering [18]. Each rule strictly reduces them, which implies termination.

By the definitions of $\mathbf{rpc}_{\mathcal{R},\lambda}$ and $\sqcap$, $h(s_1, \dots, s_n) \in \bigsqcap_{t\in\{t_1,\dots,t_m\}} \mathbf{rpc}_{\mathcal{R},\lambda}(t)$ iff $h \sim^{\rho_k}_{\mathcal{R},\gamma_k} head(t_k)$ with $\gamma_k \geqslant \lambda$ for all $1 \leqslant k \leqslant m$ and $s_i \in \bigsqcap_{t\in T_i} \mathbf{rpc}_{\mathcal{R},\lambda}(t)$, where $T_i = \{t_k|_j \mid (i,j) \in \rho_k, 1 \leqslant k \leqslant m\}$, $1 \leqslant i \leqslant n$. It makes sure that in the $\textsf{Rem}$ rule, the instance of x (which is $h(y_1, \dots, y_n)$) is in $\bigsqcap_{t\in\{t_1,\dots,t_m\}} \mathbf{rpc}_{\mathcal{R},\lambda}(t)$ iff for each $1 \leqslant i \leqslant n$ we can find an instance of y_i that is in $\bigsqcap_{t\in T_i} \mathbf{rpc}_{\mathcal{R},\lambda}(t)$. If T_i is empty, it means that the i's argument of h is irrelevant for terms in $\{t_1, \dots, t_m\}$ and can be replaced by $\_$. (The rule $\textsf{Rem}$ will take care of it in a subsequent step.) Hence, in each success branch of the derivation tree, the algorithm $\mathfrak{C}$ computes one element of $\bigsqcap_{t\in T} \mathbf{rpc}_{\mathcal{R},\lambda}(t)$. Branching at $\textsf{Red}$ help produce all elements of $\bigsqcap_{t\in T} \mathbf{rpc}_{\mathcal{R},\lambda}(t)$. □ □

It is easy to see how to use $\mathfrak{C}$ to decide the $(\mathcal{R}, \lambda)$-consistency of T: it is enough to find one successful branch in the $\mathfrak{C}$-derivation tree for $\{x \textsf{ in } T\}; x$. If there is no such branch, then T is not $(\mathcal{R}, \lambda)$-consistent. In fact, during the derivation we can even ignore the second component of the states. Only the first one matters.

Example 5.5.3. Assume a, b, c are constants, g, f, h are function symbols with the arities respectively 1, 2, and 3. Let λ be given and $\mathcal{R}$ be defined so that $\mathcal{R}(a,b) \geqslant \lambda$, $\mathcal{R}(b,c) \geqslant \lambda$, $h \sim^{\{(1,1),(1,2)\}}_{\mathcal{R},\beta} f$, $h \sim^{\{(2,1)\}}_{\mathcal{R},\gamma} g$ with $\beta \geqslant \lambda$ and $\gamma \geqslant \lambda$. Then

$$\begin{aligned}\mathbf{rpc}_{\mathcal{R},\lambda}(f(a,c)) &= \{f(a,c),\ f(b,c),\ f(a,b),\ f(b,b),\ h(b,\_,\_)\},\\ \mathbf{rpc}_{\mathcal{R},\lambda}(g(a)) &= \{g(a),\ g(b), h(\_,a,\_), h(\_,b,\_)\},\end{aligned}$$

and $\mathbf{rpc}_{\mathcal{R},\lambda}(f(a,c)) \sqcap \mathbf{rpc}_{\mathcal{R},\lambda}(g(a)) = \{h(b,a,\_), h(b,b,\_)\}$. We show how to compute this set using $\mathfrak{C}$:

$$\begin{aligned}
&x \text{ in } \{f(a,c),\, g(a)\};\ x \Longrightarrow_{\text{Red}} \\
&\{y_1 \text{ in } \{a,c\}, y_2 : \{a\}, y_3 \text{ in } \varnothing\};\ h(y_1,y_2,y_3) \Longrightarrow_{\text{Rem}} \\
&\{y_1 \text{ in } \{a,c\}, y_2 : \{a\}\};\ h(y_1,y_2,\_) \Longrightarrow_{\text{Red}} \{y_2 \text{ in } \{a\}\};\ h(b,y_2,\_).
\end{aligned}$$

Here we have two ways to apply Red to the last state, leading to two elements of $\mathbf{rpc}_{\mathcal{R},\lambda}(f(a,c)) \sqcap \mathbf{rpc}_{\mathcal{R},\lambda}(g(a))$:

$$\begin{aligned}
&\{y_2 \text{ in } \{a\}\};\ h(b,y_2,\_) \Longrightarrow_{\text{Red}} \varnothing;\ h(b,a,\_). \\
&\{y_2 \text{ in } \{a\}\};\ h(b,y_2,\_) \Longrightarrow_{\text{Red}} \varnothing;\ h(b,b,\_).
\end{aligned}$$

5.5.4 Solving generalization problems

Now we can reformulate the anti-unification problem that will be solved in the remaining part of the section.

Given: a proximity relation $\mathcal{R}$, a cut value λ, and the ground terms $t_1, \ldots, t_n$, $n \geqslant 2$.

Find: a set S of tuples $(r, \sigma_1, \ldots, \sigma_n, \alpha_1, \ldots, \alpha_n)$ such that

- $\{r \mid (r, \ldots) \in \mathsf{S}\}$ is a minimal complete set of relevant $(\mathcal{R},\lambda)$-generalizations of $t_1, \ldots, t_n$,
- $r\sigma_i \simeq_{\mathcal{R},\lambda} t_i$ and $\alpha_i = \mathsf{gdub}_{\mathcal{R},\lambda}(r, t_i)$, $1 \leqslant i \leqslant n$, for each $(r, \sigma_1, \ldots, \sigma_n, \alpha_1, \ldots, \alpha_n) \in \mathsf{S}$.

(Note that when $n = 1$, this is a problem of computing a relevant proximity class of a term.)

Below we solve the anti-unification problem for four versions of argument relations:

1. The most general (unrestricted) case; see algorithm $\mathfrak{A}^1_{\mathit{full}}$ below, the computed set of generalizations is an mcsrg;
2. Correspondence relations: using the same algorithm $\mathfrak{A}^1_{\mathit{full}}$, the computed set of generalizations is an mcsg;

3. Argument mappings: using a dedicated algorithm $\mathfrak{A}^2_{full}$, the computed set of generalizations is an mcsrg;

4. Correspondence mappings (bijections): using the algorithm $\mathfrak{A}^2_{full}$, the computed set of generalizations is an mcsg.

Each of them has also the corresponding linear variant, computing minimal complete sets of (relevant) linear $(\mathcal{R}, \lambda)$-generalizations. They are denoted $\mathfrak{A}^1_{\textit{full-lin}}$ and $\mathfrak{A}^2_{\textit{full-lin}}$, respectively.

For simplicity, we formulate the algorithms for the case $n = 2$. They can be extended for arbitrary n straightforwardly.

The main data structure in these algorithms is an anti-unification triple (AUT) $x : T_1 \triangleq T_2$, where T_1 and T_2 are finite *consistent* sets of ground terms. The idea is that x is a common generalization of all terms in $T_1 \cup T_2$.

A configuration is a tuple $A; S; r; \alpha_1; \alpha_2$, where A is a set of AUTs to be solved, S is a set of solved AUTs (the store), r is the generalization computed so far, and the α's are the current approximations of generalization degree upper bounds of r for the input terms.

Before formulating the algorithms, we discuss one peculiarity of generalizations in fully fuzzy signatures.

Example 5.5.4. For a given $\mathcal{R}$ and λ, assume $\mathcal{R}(a,b) \geqslant \lambda$, $\mathcal{R}(b,c) \geqslant \lambda$, $h \sim^{(1,1),(1,2)}_{\mathcal{R},\alpha} f$ and $h \sim^{(1,1)}_{\mathcal{R},\beta} g$, where f is binary, g, h are unary, $\alpha \geqslant \lambda$ and $\beta \geqslant \lambda$. Then

- $h(b)$ is an $(\mathcal{R}, \lambda)$-generalization of $f(a,c)$ and $g(a)$.
- x is the only $(\mathcal{R}, \lambda)$-generalization of $f(a,d)$ and $g(a)$. One may be tempted to have h as the head of the generalization, e.g., $h(x)$, but x cannot be instantiated by any term that would be $(\mathcal{R}, \lambda)$-close to both a and d, since in the given $\mathcal{R}$, d is $(\mathcal{R}, \lambda)$-close only to itself. Hence, there would be no instance of $h(x)$ that is $(\mathcal{R}, \lambda)$-close to $f(a,d)$. Since there is no other alternative (except h) for the common neighbor of f and g, the generalization should be a fresh variable x.

This example shows that generalization algorithms should take into account not only the heads of the terms to be generalized, but also should look deeper, to make sure that the arguments grouped together by the given argument relation have a common neighbor. This justifies the requirement of consistency of a set of arguments, used in the decomposition rules in the algorithms below.

Anti-unification for unrestricted argument relations

Algorithms $\mathfrak{A}^1_{\mathit{full\text{-}lin}}$ and $\mathfrak{A}^1_{\mathit{full}}$ use the rules below to transform configurations into configurations. Given $\mathcal{R}$, λ, and the ground terms t_1 and t_2, we create the initial configuration $\{x : \{t_1\} \triangleq \{t_2\}\}; \varnothing; x; 1; 1$ and apply the rules as long as possible. Note that the rules preserve consistency of AUTs. The process generates a finite complete tree of derivations, whose terminal nodes have configurations with the first component empty. We will show how from these terminal configurations one collects the result as required in the anti-unification problem statement.

Tri: **Trivial**

$\{x : \varnothing \triangleq \varnothing\} \uplus A;\ S;\ r;\ \alpha_1; \alpha_2 \Longrightarrow A;\ S;\ r\{x \mapsto \_\};\ \alpha_1; \alpha_2.$

Dec: **Decomposition**

$\{x : T_1 \triangleq T_2\} \uplus A; S; r; \alpha_1; \alpha_2 \Longrightarrow$
$\quad \{y_i : Q_{i1} \triangleq Q_{i2} \mid 1 \leqslant i \leqslant n\} \cup A; S; r\{x \mapsto h(y_1, \dots, y_n)\}; \alpha_1 \wedge \beta_1; \alpha_2 \wedge \beta_2,$

where $T_1 \cup T_2 \neq \varnothing$; h is n-ary with $n \geqslant 0$; $y_1, \dots, y_n$ are fresh; and for $j = 1, 2$, if $T_j = \{t^j_1, \dots, t^j_{m_j}\}$, then

- $h \sim^{\rho^j_k}_{\mathcal{R},\gamma^j_k} \mathit{head}(t^j_k)$ with $\gamma^j_k \geqslant \lambda$ for all $1 \leqslant k \leqslant m_j$ and $\beta_j = \gamma^j_1 \wedge \dots \wedge \gamma^j_{m_j}$ (note that $\beta_j = 1$ if $m_j = 0$),
- for all $1 \leqslant i \leqslant n$, $Q_{ij} = \cup_{k=1}^{m_j} \{t^j_k|_q \mid (i, q) \in \rho^j_k\}$ and is $(\mathcal{R}, \lambda)$-consistent.

Sol: **Solving**

$\{x : T_1 \triangleq T_2\} \uplus A;\ S;\ r;\ \alpha_1; \alpha_2 \Longrightarrow A;\ \{x : T_1 \triangleq T_2\} \cup S;\ r;\ \alpha_1; \alpha_2,$

if Tri and Dec rules are not applicable. (It means that at least one $T_i \neq \varnothing$ and either there is no h as it is required in the Dec rule, or at least one Q_{ij} from Dec is not consistent.)

Let expand be an *expansion operation* defined for sets of AUTs as

$$\mathsf{expand}(S) := \{x : \prod_{t \in T_1} \mathbf{rpc}_{\mathcal{R},\lambda}(t) \triangleq \prod_{t \in T_2} \mathbf{rpc}_{\mathcal{R},\lambda}(t) \mid x : T_1 \triangleq T_2 \in S\}.$$

Exhaustive application of the three rules above leads to configurations of the form $\varnothing; S; r; \alpha_1; \alpha_2$, where r is a linear term. These configurations are further postprocessed, replacing S by $\mathsf{expand}(S)$. We will use the letter E for expanded stores. Hence, terminal configurations obtained after the exhaustive rule application and expansion have the form $\emptyset; E; r; \alpha_1; \alpha_2$, where r is a linear term.[2] This is what Algorithm $\mathfrak{A}^1_{\textit{full-lin}}$ stops with.

To an expanded store $E = \{y_1 : Q_{11} \triangleq Q_{12}, \ldots, y_n : Q_{n1} \triangleq Q_{n2}\}$ we associate two sets of substitutions $\Sigma_L(E)$ and $\Sigma_R(E)$, defined as follows: $\sigma \in \Sigma_L(E)$ (resp. $\sigma \in \Sigma_R(E)$) iff $\mathit{dom}(\sigma) = \{y_1, \ldots, y_n\}$ and $y_i\sigma \in Q_{i1}$ (resp. $y_i\sigma \in Q_{i2}$) for each $1 \leqslant i \leqslant n$. We call them the sets of *witness substitutions*.

Configurations containing expanded stores are called *expanded configurations*. From each expanded configuration $C = \emptyset; E; r; \alpha_1; \alpha_2$, we construct the set $\mathsf{S}(C) := \{(r, \sigma_1, \sigma_2, \alpha_1, \alpha_2) \mid \sigma_1 \in \Sigma_L(E), \sigma_2 \in \Sigma_R(E)\}$.

Given an anti-unification problem $\mathcal{R}$, λ, t_1 and t_2, the *answer computed by Algorithm* $\mathfrak{A}^1_{\text{full-lin}}$ is the set $\mathsf{S} := \cup_{i=1}^m \mathsf{S}(C_i)$, where $C_1, \ldots, C_m$ are all of the final expanded configurations reached by $\mathfrak{A}^1_{\textit{full-lin}}$ for $\mathcal{R}$, λ, t_1, and t_2.[3]

Algorithm $\mathfrak{A}^1_{\textit{full}}$ is obtained by further transforming the expanded configurations produced by $\mathfrak{A}^1_{\textit{full-lin}}$. This transformation is performed by applying the **Merge** rule below as long as possible. Intuitively, its purpose is to make the linear generalization obtained by $\mathfrak{A}^1_{\textit{full-lin}}$ less general by merging some variables.

Mer: Merge

$$\emptyset; \{x_1 : R_{11} \triangleq R_{12},\, x_2 : R_{21} \triangleq R_{22}\} \uplus E;\ r;\ \alpha_1;\ \alpha_2 \Longrightarrow$$
$$\emptyset;\ \{y : Q_1 \triangleq Q_2\} \cup E;\ r\sigma;\ \alpha_1;\ \alpha_2,$$

where

- $Q_i = (R_{1i} \sqcap R_{2i}) \neq \emptyset$, $i = 1, 2$,
- y is fresh and $\sigma = \{x_1 \mapsto y,\, x_2 \mapsto y\}$.

The answer computed by $\mathfrak{A}^1_{\textit{full}}$ is defined analogously to the answer computed by $\mathfrak{A}^1_{\textit{full-lin}}$.

Example 5.5.5. Assume a, b, c and d are constants with $b \sim^{\emptyset}_{\mathcal{R},0.5} c$, $c \sim^{\emptyset}_{\mathcal{R},0.6} d$, and f, g and h are respectively binary, ternary and quaternary function symbols with $h \sim^{\{(1,1),(3,2),(4,2)\}}_{\mathcal{R},0.7} f$ and $h \sim^{\{(1,1),(3,3)\}}_{\mathcal{R},0.8} g$. For the proximity

[2]Note that no side of the AUTs in E in those configurations is empty due to the condition at the **Decomposition** rule requiring the Q_{ij}'s to be $(\mathcal{R}, \lambda)$-consistent.

[3]If we are interested only in linear generalizations *without witness substitutions*, there is no need in computing expanded configurations in $\mathfrak{A}^1_{\textit{full-lin}}$.

relation $\mathcal{R}$ given in this way and $\lambda = 0.5$, Algorithms $\mathfrak{A}^1_{\textit{full-lin}}$ and $\mathfrak{A}^1_{\textit{full}}$ perform the following steps to anti-unify $f(a,b)$ and $g(a,c,d)$:

$$
\begin{aligned}
&\{x : \{f(a,b)\} \triangleq \{g(a,c,d)\}\}; \varnothing; x; 1; 1 \Longrightarrow_{\mathsf{Dec}}\\
&\{x_1 : \{a\} \triangleq \{a\},\ x_2 : \varnothing \triangleq \varnothing,\ x_3 : \{b\} \triangleq \{d\},\\
&\qquad x_4 : \{b\} \triangleq \varnothing\}; \varnothing; h(x_1,x_2,x_3,x_4); 0.7; 0.8 \Longrightarrow_{\mathsf{Dec}}\\
&\{x_2 : \varnothing \triangleq \varnothing,\ x_3 : \{b\} \triangleq \{d\},\ x_4 : \{b\} \triangleq \varnothing\};\\
&\qquad \varnothing; h(a,x_2,x_3,x_4); 0.7; 0.8 \Longrightarrow_{\mathsf{Tri}}\\
&\{x_3 : \{b\} \triangleq \{d\},\ x_4 : \{b\} \triangleq \varnothing\};\\
&\qquad \varnothing; h(a,\_,x_3,x_4); 0.7; 0.8 \Longrightarrow_{\mathsf{Dec}}\\
&\{x_4 : \{b\} \triangleq \varnothing\}; \varnothing; h(a,\_,c,x_4); 0.5; 0.6.
\end{aligned}
$$

Here Dec applies in two different ways, with the substitutions $\{x_4 \mapsto b\}$ and $\{x_4 \mapsto c\}$, leading to two final configurations(both $\mathfrak{A}^1_{\textit{full-lin}}$ and $\mathfrak{A}^1_{\textit{full}}$ would give them):

$$\varnothing; \varnothing; h(a,\_,c,b); 0.5; 0.6, \qquad \varnothing; \varnothing; h(a,\_,c,c); 0.5; 0.6.$$

The witness substitutions are the identity substitutions. We have $\mathcal{R}(h(a,\_,c,b), f(a,b)) = 0.5$, $\mathcal{R}(h(a,\_,c,b), g(a,c,d)) = 0.6$, $\mathcal{R}(h(a,\_,c,c), f(a,b)) = 0.5$, and $\mathcal{R}(h(a,\_,c,c), g(a,c,d)) = 0.6$.

If we had $h \sim^{\{(1,1),(1,2),(4,2)\}}_{\mathcal{R},0.7} f$, then the algorithm would perform only the Sol step, because in the attempt to apply Dec to the initial configuration, the set $Q_{11} = \{a,b\}$ is inconsistent: $\mathbf{rpc}_{\mathcal{R},\lambda}(a) = \{a\}$, $\mathbf{rpc}_{\mathcal{R},\lambda}(b) = \{b,c\}$, and, hence, $\mathbf{rpc}_{\mathcal{R},\lambda}(a) \cap \mathbf{rpc}_{\mathcal{R},\lambda}(b) = \varnothing$.

Example 5.5.6. Assume $a_1, a_2, b_1, b_2, c_1, c_2$ are constants and f, g, h are ternary function symbols. Let $\lambda = 0.4$ and the proximity relation $\mathcal{R}$ be defined by $a_1 \sim^{\varnothing}_{\mathcal{R},0.5} b_1$, $b_1 \sim^{\varnothing}_{\mathcal{R},0.5} c_1$, $a_2 \sim^{\varnothing}_{\mathcal{R},0.6} b_2$, $b_2 \sim^{\varnothing}_{\mathcal{R},0.6} c_2$, $h \sim^{\{(1,1),(1,2),(2,2),(3,3)\}}_{\mathcal{R},0.7} f$ and $h \sim^{\{(1,1),(2,2),(3,2),(3,3)\}}_{\mathcal{R},0.8} g$. To anti-unify $f(a_1,b_1,c_1)$ and $g(a_2,b_2,c_2)$, we would start with the following steps:

$$
\begin{aligned}
&\{x : \{f(a_1,b_1,c_1)\} \triangleq \{g(a_2,b_2,c_2)\}\}; \varnothing; x; 1; 1 \Longrightarrow_{\mathsf{Dec}}\\
&\{x_1 : \{a_1,b_1\} \triangleq \{a_2\},\ x_2 : \{b_1\} \triangleq \{b_2\},\\
&\quad x_3 : \{c_1\} \triangleq \{b_2,c_2\}\}; \varnothing; h(x_1,x_2,x_3); 0.7; 0.8 \Longrightarrow^3_{\mathsf{Sol}}\\
&\varnothing; \{x_1 : \{a_1,b_1\} \triangleq \{a_2\},\ x_2 : \{b_1\} \triangleq \{b_2\},\\
&\quad x_3 : \{c_1\} \triangleq \{b_2,c_2\}\}; h(x_1,x_2,x_3); 0.7; 0.8.
\end{aligned}
$$

Here we expand the final store, obtaining

$$\begin{aligned}&\varnothing; \{x_1 : \{a_1, b_1\} \triangleq \{a_2, b_2\},\ x_2 : \{a_1, b_1, c_1\} \triangleq \{a_2, b_2, c_2\},\\ &\quad x_3 : \{b_1, c_1\} \triangleq \{b_2, c_2\}\};\ h(x_1, x_2, x_3);\ 0.7;\ 0.8.\end{aligned}$$

Algorithm $\mathfrak{A}^1_{\mathit{full\text{-}lin}}$ would stop here. From the computed store, one can extract 144 pairs of witness substitutions. For each of them, the generalization degrees do not exceed the computed ones. For instance, for $\sigma_1 = \{x_1 \mapsto a_1, x_2 \mapsto b_1, x_3 \mapsto c_1\}$ and $\sigma_2 = \{x_1 \mapsto a_2, x_2 \mapsto b_2, x_3 \mapsto c_2\}$ we have

$$\begin{aligned}\mathcal{R}(h(x_1, x_2, x_3)\sigma_1, f(a_1, b_1, c_1)) &= \mathcal{R}(h(a_1, b_1, c_1), f(a_1, b_1, c_1)) = 0.5 \leqslant 0.7,\\ \mathcal{R}(h(x_1, x_2, x_3)\sigma_2, g(a_2, b_2, c_2)) &= \mathcal{R}(h(a_2, b_2, c_2), g(a_2, b_2, c_2)) = 0.6 \leqslant 0.8.\end{aligned}$$

Algorithm $\mathfrak{A}^1_{\mathit{full}}$ would perform two more merging steps:

$$\begin{aligned}&\varnothing; \{x_1 : \{a_1, b_1\} \triangleq \{a_2, b_2\},\ x_2 : \{a_1, b_1, c_1\} \triangleq \{a_2, b_2, c_2\},\\ &\quad x_3 : \{b_1, c_1\} \triangleq \{b_2, c_2\}\};\ h(x_1, x_2, x_3);\ 0.7;\ 0.8 \Longrightarrow^2_{\text{Mer}}\\ &\varnothing; \{y : \{b_1\} \triangleq \{b_2\}\};\ h(y, y, y);\ 0.7;\ 0.8\end{aligned}$$

Taking the pair of two witness substitutions $\sigma_1 = \{y \mapsto b_1\}$ and $\sigma_2 = \{y \mapsto b_2\}$, we get:

$$\begin{aligned}\mathcal{R}(h(y, y, y)\sigma_1,\ f(a_1, b_1, c_1)) &= \mathcal{R}(h(b_1, b_1, b_1),\ f(a_1, b_1, c_1)) = 0.5 \leqslant 0.7,\\ \mathcal{R}(h(y, y, y)\sigma_2,\ g(a_2, b_2, c_2)) &= \mathcal{R}(h(b_2, b_2, b_2),\ g(a_2, b_2, c_2)) = 0.6 \leqslant 0.8.\end{aligned}$$

Example 5.5.7. Assume a, b are constants, f_1, f_2, g_1, and g_2 are unary function symbols, p is a binary function symbol, and h_1 and h_2 are ternary function symbols. Let λ be a cut value and $\mathcal{R}$ be defined as $f_i \sim^{\{(1,1)\}}_{\mathcal{R},\alpha_i} h_i$ and $g_i \sim^{\{(1,2)\}}_{\mathcal{R},\beta_i} h_i$ with $\alpha_i \geqslant \lambda$, $\beta_i \geqslant \lambda$, $i = 1, 2$. To generalize $p(f_1(a), g_1(b))$ and $p(f_2(a), g_2(b))$, we can use $\mathfrak{A}^1_{\mathit{full}}$. The derivation starts as

$$\begin{aligned}&\{x : \{p(f_1(a), g_1(b))\} \triangleq \{p(f_2(a), g_2(b))\}\};\ \varnothing;\ x;\ 1;\ 1 \Longrightarrow_{\text{Dec}}\\ &\{y_1 : \{f_1(a)\} \triangleq \{f_2(a)\},\ y_2 : \{g_1(b)\} \triangleq \{g_2(b)\}\};\ \varnothing;\ p(y_1, y_2);\ 1;\ 1 \Longrightarrow^2_{\text{Sol}}\\ &\varnothing;\ \{y_1 : \{f_1(a)\} \triangleq \{f_2(a)\},\ y_2 : \{g_1(b)\} \triangleq \{g_2(b)\}\};\ p(y_1, y_2);\ 1;\ 1.\end{aligned}$$

At this stage, we expand the store, obtaining

$$\varnothing;\ \{y_1 : \{f_1(a), h_1(a, \_, \_)\} \triangleq \{f_2(a), h_2(a, \_, \_)\},$$

$$y_2 : \{g_1(b), h_1(\_, b, \_)\} \triangleq \{g_2(b), h_2(\_, b, \_)\}\};\ p(y_1, y_2);\ 1;\ 1.$$

If we had the standard intersection $\cap$ in the **Mer** rule, we would not be able to merge y_1 and y_2, because the obtained sets in the corresponding AUTs are disjoint. However, **Mer** uses $\sqcap$: we have $\{f_i(a), h_i(a, \_, \_)\} \sqcap \{g_i(b), h_i(\_, b, \_)\} = \{h_i(a, b, \_)\}$, $i = 1, 2$ and, therefore, can make the step

$$\begin{aligned}&\varnothing;\ \{y_1 : \{f_1(a), h_1(a, \_, \_)\} \triangleq \{f_2(a), h_2(a, \_, \_)\},\\ &\qquad y_2 : \{g_1(b), h_1(\_, b, \_)\} \triangleq \{g_2(b), h_2(\_, b, \_)\}\};\ p(y_1, y_2);\ 1;\ 1 \Longrightarrow_{\mathsf{Mer}}\\ &\varnothing;\ \{z : \{h_1(a, b, \_)\} \triangleq \{h_2(a, b, \_)\}\};\ p(z, z);\ 1;\ 1.\end{aligned}$$

Indeed, if we take the witness substitutions $\sigma_i = \{z \mapsto h_i(a, b, \_)\}$, $i = 1, 2$, and apply them to the obtained generalization, we get

$$\begin{aligned}p(z, z)\sigma_1 &= p(h_1(a, b, \_), h_1(a, b, \_)) \simeq_{\mathcal{R},\lambda} p(f_1(a), g_1(b)),\\ p(z, z)\sigma_2 &= p(h_2(a, b, \_), h_2(a, b, \_)) \simeq_{\mathcal{R},\lambda} p(f_2(a), g_2(b)).\end{aligned}$$

Theorem 5.5.3. *Given $\mathcal{R}$, λ, and the ground terms t_1 and t_2, Algorithm $\mathfrak{A}^1_{\text{full}}$ terminates for the initial configuration $\{x : \{t_1\} \triangleq \{t_2\}\}; \varnothing; x; 1; 1$ and computes an answer set S such that*

1. *the set $\{r \mid (r, \sigma_1, \sigma_2, \alpha_1, \alpha_2) \in \mathsf{S}\}$ is an $(\mathcal{R}, \lambda)$-mcsrg of t_1 and t_2,*
2. *for each $(r, \sigma_1, \sigma_2, \alpha_1, \alpha_2) \in \mathsf{S}$ we have $\mathcal{R}(r\sigma_i, t_i) \leqslant \alpha_i = \mathsf{gdub}_{\mathcal{R},\lambda}(r, t_i)$, $i = 1, 2$.*

Proof. Termination: Define the depth of an AUT $x : \{t_1, \dots, t_m\} \triangleq \{s_1, \dots, s_n\}$ as the depth of the term $f(g(t_1, \dots, t_m), h(s_1, \dots, s_n))$. The rules **Tri**, **Dec**, and **Sol** strictly reduce the multiset of depths of AUTs in the first component of the configurations. **Mer** strictly reduces the number of distinct variables in generalizations. Hence, these rules cannot be applied infinitely often and $\mathfrak{A}^1_{full}$ terminates.

In order to prove 1), we need to verify three properties:

- Soundness: If $(r, \sigma_1, \sigma_2, \alpha_1, \alpha_2) \in \mathsf{S}$, then r is a relevant $(\mathcal{R}, \lambda)$-generalization of t_1 and t_2.
- Completeness: If r' is a relevant $(\mathcal{R}, \lambda)$-generalization of t_1 and t_2, then there exists $(r, \sigma_1, \sigma_2, \alpha_1, \alpha_2) \in \mathsf{S}$ such that $r' \lesssim r$.

- Minimality: If r and r' belong to two tuples from S such that $r \neq r'$, then neither $r \prec_{\mathcal{R},\lambda} r'$ nor $r' \prec_{\mathcal{R},\lambda} r$.

Soundness: We show that each rule transforms an $(\mathcal{R}, \lambda)$-generalization into an $(\mathcal{R}, \lambda)$-generalization. Since we start from a most general $(\mathcal{R}, \lambda)$-generalization of t_1 and t_2 (a fresh variable x), at the end of the algorithm we will get an $(\mathcal{R}, \lambda)$-generalization of t_1 and t_2. We also show that in this process all irrelevant positions are abstracted by anonymous variables, to guarantee that each computed generalization is relevant.

Dec: The computed h is $(\mathcal{R}, \lambda)$-close to the head of each term in $T_1 \cup T_2$. Q_{ij}'s correspond to argument relations between h and those heads, and each Q_{ij} is $(\mathcal{R}, \lambda)$-consistent, i.e., there exists a term that is $(\mathcal{R}, \lambda)$-close to each term in Q_{ij}. It implies that $x\sigma = h(y_1, \dots, y_n)$ $(\mathcal{R}, \lambda)$-generalizes all the terms from $T_1 \cup T_2$. Note that at this stage, $h(y_1, \dots, y_n)$ might not yet be a relevant $(\mathcal{R}, \lambda)$-generalization of T_1 and T_2: if there exists an irrelevant position $1 \leqslant i \leqslant n$ for the $(\mathcal{R}, \lambda)$-generalization of T_1 and T_2, then in the new configuration we will have an AUT $y_i : \varnothing \triangleq \varnothing$.

Tri: When Dec generates $y : \varnothing \triangleq \varnothing$, the Tri rule replaces y by $\_$ in the computed generalization, making it relevant.

Sol does not change generalizations.

Mer merges AUTs whose terms have *nonempty* intersection of **rpc**'s. Hence, we can reuse the same variable in the corresponding positions in generalizations, i.e., Mer transforms a generalization computed so far into a less general one.

Completeness: We prove a slightly more general statement. Given two finite consistent sets of ground terms T_1 and T_2, if r' is a relevant $(\mathcal{R}, \lambda)$-generalization for all $t_1 \in T_1$ and $t_2 \in T_2$, then starting from $\{x : T_1 \triangleq T_2\}; \varnothing; x; 1; 1$, Algorithm $\mathfrak{A}^1_{\mathit{full}}$ computes a $(r, \sigma_1, \sigma_2, \alpha_1, \alpha_2)$ such that $r' \lesssim r$.

We may assume w.l.o.g. that r' is a relevant $(\mathcal{R}, \lambda)$-lgg. Due to the transitivity of $\lesssim$, completeness for such an r' will imply it for all terms more general than r'.

We proceed by structural induction on r'. If r' is a (named or anonymous) variable, the statement holds. Assume $r' = h(r'_1, \dots, r'_n)$, $T_1 = \{u_1, \dots, u_m\}$, and $T_2 = \{w_1, \dots, w_l\}$. Then h is such that $h \sim^{\rho_i}_{\mathcal{R},\beta_i} \mathit{head}(u_i)$ for all $1 \leqslant i \leqslant m$ and $h \sim^{\mu_j}_{\mathcal{R},\gamma_j} \mathit{head}(w_j)$ for all $1 \leqslant j \leqslant l$. Moreover, each r'_k is a relevant $(\mathcal{R}, \lambda)$-generalization of $Q_{k1} = \cup_{i=1}^{m}\{u_i|_q \mid (k, q) \in \rho_i\}$ and $Q_{k2} = \cup_{j=1}^{l}\{w_j|_q \mid (k, q) \in \mu_j\}$ and, hence, Q_{k1} and Q_{k2} are $(\mathcal{R}, \lambda)$-consistent. Therefore, we can perform a step by Dec, choosing $h(y_1, \dots, y_k)$ as the generalization term and $y_i : Q_{i1} \triangleq Q_{i2}$ as the new AUTs. By the induction hypothesis, for each $1 \leqslant i \leqslant n$ we can compute a relevant $(\mathcal{R}, \lambda)$-generalization r_i for Q_{i_1} and Q_{i2} such that $r'_i \lesssim r_i$.

If r' is linear, then the combination of the current Dec step with the derivations that lead to those r_i's computes a tuple $(r, \dots) \in \mathsf{S}$, where $r = h(r_1, \dots, r_n)$ and, hence, $r' \lesssim r$.

If r' is non-linear, assume w.l.o.g. that all occurrences of a shared variable z appear as the direct arguments of h: $z = r'_{k_1} = \dots = r'_{k_p}$ for $1 \leqslant k_1 < \dots < k_p \leqslant n$. Since r' is an lgg, Q_{k_i1} and Q_{k_i2} cannot be generalized by a non-variable term, thus, Tri and Dec are not applicable. Therefore, the AUTs $y_i : Q_{k_i1} \triangleq Q_{k_i2}$ would be transformed by Sol. Since all pairs Q_{k_i1} and Q_{k_i2}, $1 \leqslant i \leqslant p$, are generalized by the same variable, we have $\sqcap_{t \in Q_j} \mathbf{rpc}_{\mathcal{R},\lambda}(t) \neq \varnothing$, where $Q_j = \cup_{i=1}^{p} Q_{k_ij}$, $j = 1, 2$. Additionally, $r'_{k_1}, \dots, r'_{k_p}$ are all occurrences of z in r'. Hence, the condition of Mer is satisfied and we can extend our derivation with $p-1$-fold application of this rule, obtaining $r = h(r_1, \dots, r_n)$ with $z = r_{k_1} = \dots = r_{k_p}$, implying $r' \lesssim r$.

Minimality: Alternative generalizations are obtained by branching in Dec or Mer. If the current generalization r is transformed by Dec into two generalizations r_1 and r_2 on two branches, then $r_1 = h_1(y_1, \dots, y_m)$ and $r_2 = h_2(z_1, \dots, z_n)$ for some h's, and fresh y's and z's. It may happen that $r_1 \lesssim_{\mathcal{R},\lambda} r_2$ or vice versa (if h_1 and h_2 are $(\mathcal{R}, \lambda)$-close to each other), but neither $r_1 \prec_{\mathcal{R},\lambda} r_2$ nor $r_2 \prec_{\mathcal{R},\lambda} r_1$ holds. Hence, the set of generalizations computed before applying Mer is minimal. Mer groups AUTs together maximally, and different groupings are not comparable. Therefore, variables in generalizations are merged so that distinct generalizations are not $\prec_{\mathcal{R},\lambda}$-comparable. Hence, 1) is proven.

As for 2), for $i = 1, 2$, from the construction in Dec follows $\mathcal{R}(r\sigma_i, t_i) \leqslant \alpha_i$. Mer does not change α_i, thus, $\alpha_i = \mathsf{gdub}_{\mathcal{R},\lambda}(r, t_i)$ also holds, since the way how α_i is computed corresponds exactly to the computation of $\mathsf{gdub}_{\mathcal{R},\lambda}(r, t_i)$: $r \lesssim_{\mathcal{R},\lambda} t_i$ and only the decomposition changes the degree during the computation. □

Corollary 5.5.3.1. *Given $\mathcal{R}$, λ, and the ground terms t_1 and t_2, Algorithm $\mathfrak{A}^1_{\text{full-lin}}$ terminates for the initial configuration $\{x : \{t_1\} \triangleq \{t_2\}\}; \varnothing; x; 1; 1$ and computes an answer set S such that*

1. *the set $\{r \mid (r, \sigma_1, \sigma_2, \alpha_1, \alpha_2) \in \mathsf{S}\}$ is a minimal complete set of relevant* linear *$(\mathcal{R}, \lambda)$-generalizations of t_1 and t_2,*

2. *for each* $(r, \sigma_1, \sigma_2, \alpha_1, \alpha_2) \in \mathsf{S}$ *we have* $\mathcal{R}(r\sigma_i, t_i) \leqslant \alpha_i = \mathsf{gdub}_{\mathcal{R},\lambda}(r, t_i)$, $i = 1, 2$.

Anti-unification with correspondence argument relations

Correspondence relations make sure that for a pair of proximal symbols, no argument is irrelevant for proximity. Left- and right-totality of those relations guarantee that each argument of a term is close to at least one argument of its proximal term and the inverse relation remains a correspondence relation. Consequently, in the **Dec** rule of $\mathfrak{A}^1_{\mathit{full}}$, the sets Q_{ij} never get empty. Therefore, the **Tri** rule becomes obsolete and no anonymous variable appears in generalizations. As a result, the $(\mathcal{R}, \lambda)$-mcsrg and the $(\mathcal{R}, \lambda)$-mcsg coincide, and the algorithm computes a solution from which we get an $(\mathcal{R}, \lambda)$-mcsg for the given anti-unification problem. The linear version $\mathfrak{A}^1_{\mathit{full\text{-}lin}}$ works analogously.

Anti-unification with argument mappings

When the argument relations are mappings, we are able to design a more constructive method for computing generalizations and their degree bounds. (Recall that our mappings are partial injective functions, which guarantees that their inverses are also partial mappings.) The configurations stay the same as in Algorithm $\mathfrak{A}^1_{\mathit{full}}$, but the AUTs in A will contain only empty or singleton sets of terms. In the store, we may still get (after expansion) AUTs with term sets containing more than one element. Only the **Dec** rule differs in $\mathfrak{A}^2_{\mathit{full}}$ (and in its linear variant $\mathfrak{A}^2_{\mathit{full\text{-}lin}}$):

Dec: Decomposition

$$\{x : T_1 \triangleq T_2\} \uplus A; S; r; \alpha_1; \alpha_2 \Longrightarrow$$
$$\{y_i : Q_{i1} \triangleq Q_{i2} \mid 1 \leqslant i \leqslant n\} \cup A; S;$$
$$r\{x \mapsto h(y_1, \ldots, y_n)\}; \alpha_1 \wedge \beta_1; \alpha_2 \wedge \beta_2,$$

where $T_1 \cup T_2 \neq \emptyset$; h is n-ary with $n \geqslant 0$; $y_1, \ldots, y_n$ are fresh; for $j = 1, 2$ and for all $1 \leqslant i \leqslant n$,

- if $T_j = \{t_j\}$, then $h \sim^{\pi_j}_{\mathcal{R},\beta_j} \mathit{head}(t_j)$, $Q_{ij} = \{t_j|_{\pi_j(i)}\}$,
- if $T_j = \emptyset$, then $\beta_j = 1$ and $Q_{ij} = \emptyset$.

This **Dec** rule is equivalent to the special case of **Dec** in $\mathfrak{A}^1_{\mathit{full}}$ for $m_j \leqslant 1$. The new Q_{ij}'s contain at most one element (due to mappings) and, thus, are always $(\mathcal{R}, \lambda)$-consistent. Various choices of h in Dec and alternatives in grouping AUTs in Mer cause branching in the same way as in $\mathfrak{A}^1_{\mathit{full}}$. It is easy to see that the counterpart of Theorem 5.5.3 holds for $\mathfrak{A}^2_{\mathit{full}}$ and $\mathfrak{A}^2_{\mathit{full\text{-}lin}}$ as well.

A special case of this fragment of anti-unification is anti-unification for similarity relations in full fuzzy signatures from [2] (described earlier in 5.2). The position mappings in [2] can be modeled by our argument mappings, requiring them to be total for symbols of the smaller arity and to satisfy the similarity-specific consistency restrictions from [2].

Anti-unification with correspondence argument mappings

Correspondence argument mappings are bijections between arguments of function symbols of the same arity. For such mappings, if $h \simeq^{\pi}_{\mathcal{R},\lambda} f$ and h is n-ary, then f is also n-ary and π is a permutation of $(1, \dots, n)$. Hence, $\mathfrak{A}^2_{\mathit{full}}$ combines in this case the properties of $\mathfrak{A}^1_{\mathit{full}}$ for correspondence relations and of $\mathfrak{A}^2_{\mathit{full}}$ for argument mappings: all generalizations are relevant, computed answer gives an mcsg of the input terms, and the algorithm works with term sets of cardinality at most 1.

5.5.5 Remarks about the complexity

The proximity relation $\mathcal{R}$ can be represented as a weighted undirected graph, where the vertices are function symbols and an edge between them indicates that they are proximal. Graphs induced by proximity relations are usually sparse. Therefore we choose to represent the graphs by (sorted) adjacency lists rather than by adjacency matrices. In the adjacency lists, we also accommodate the argument relations and proximity degrees. For instance, if we have $h \sim^{\rho}_{\mathcal{R},\gamma} f$ where f is n-ary and h is m-ary $(n, m \geqslant 0)$, then the adjacency list for f contains both $\langle f, \{(1,1), \dots, (n,n)\}, 1\rangle$ and $\langle h, \rho, \gamma\rangle$, and the adjacency list for h contains both $\langle h, \{(1,1), \dots, (m,m)\}, 1\rangle$ and $\langle f, \rho^{-1}, \gamma\rangle$.

In the rest of this section we use the following notation:

- n: the size of the input (number of symbols) of the corresponding algorithms,
- Δ: the maximum degree of $\mathcal{R}$ considered as a graph,
- $\mathfrak{a}$: the maximum arity of function symbols that occur in $\mathcal{R}$.

- $m^{\bullet n}$: a function defined on natural numbers m and n such that $1^{\bullet n} = n$ and $m^{\bullet n} = m^n$ for $m \neq 1$.

We assume that the given anti-unification problem is represented as a completely shared directed acyclic graph (dag). Each node of the dag has a pointer to the adjacency list (with respect to $\mathcal{R}$) of the symbol in the node.[4]

Theorem 5.5.4. *Time complexities of $\mathfrak{C}$ and the linear versions of the generalization algorithms are as follows:*

- *$\mathfrak{C}$ for argument relations and $\mathfrak{A}^1_{\text{full-lin}}$:* $O(n \cdot \Delta \cdot \Delta^{\bullet \mathfrak{a}^{\bullet n}})$,
- *$\mathfrak{C}$ for argument mappings and $\mathfrak{A}^2_{\text{full-lin}}$:* $O(n \cdot \Delta \cdot \Delta^{\bullet n})$.

The time complexities for algorithms $\mathfrak{A}^1_{\text{full}}$, and $\mathfrak{A}^2_{\text{full}}$ are the following:

- *for $\mathfrak{A}^1_{\text{full}}$:* $O\left(\binom{\mathfrak{a}^{\bullet n}}{\lfloor \mathfrak{a}^{\bullet n}/2 \rfloor} \cdot \Delta^{\bullet 2\mathfrak{a}^{\bullet n}} \cdot \mathfrak{a}^{\bullet n} \cdot n\right)$,
- *for $\mathfrak{A}^2_{\text{full}}$:* $O\left(\binom{n}{\lfloor n/2 \rfloor} \cdot \Delta^{\bullet 2n} \cdot n^2\right)$.

Proof. In $\mathfrak{C}$, in the case of argument relations (unrestricted or correspondence), an application of the Red rule to a state $\mathbf{T}; s$ replaces one element of $\mathbf{T}$ of size m by at most $\mathfrak{a}$ new elements, each of them of size $m-1$. Hence, one branch in the search tree for $\mathfrak{C}$, starting from a singleton set $\mathbf{T}$ of size n, will have the length at most $l = \sum_{i=0}^{n-1} \mathfrak{a}^i$. At each node of it there are at most Δ choices of applying Red with different h's, which gives the total size of the search tree to be at most $\sum_{i=0}^{l-1} \Delta^i$, i.e., the number of steps performed by $\mathfrak{C}$ is in the worst case $O(\Delta^{\bullet \mathfrak{a}^{\bullet n}})$. Those different h's are obtained by intersecting the proximity classes of the heads of terms $\{t_1, \dots, t_m\}$ in the Red rule. In our graph representation of the proximity relation, proximity classes of symbols are exactly the adjacency lists of those symbols which we assume are sorted. Their maximal length is Δ. Hence, the work to be done at each node of the search tree of $\mathfrak{C}$ is to find the intersection of at most n sorted lists, each containing at most Δ elements. It needs $O(n \cdot \Delta)$ time. It gives the time complexity $O(n \cdot \Delta \cdot \Delta^{\bullet \mathfrak{a}^{\bullet n}})$ of $\mathfrak{C}$ for the relation case.

[4]We assume that all pointers require constant space and all basic operations on them can be done in constant time. Also that all symbols can be represented in constant space and all basic operations on them are done in constant time. These assumptions, although not very accurate, are popular in the literature, see a remark in [10, p. 304].

In the mapping case, an application of the **Red** rule of $\mathfrak{C}$ to a state $\mathbf{T}; s$ replaces one element of $\mathbf{T}$ of size m by at most $\mathfrak{a}$ new elements of the *total* size $m-1$. Therefore, the maximal length of a branch is n, the branching factor is Δ, and the amount of work at each node, like above, is $O(n \cdot \Delta)$. Hence, the number of steps in the worst case is $O(\Delta^{\bullet n})$ and the time complexity of $\mathfrak{C}$ is $O(n \cdot \Delta \cdot \Delta^{\bullet n})$.

The fact that consistency check is incorporated in the **Dec** rule in $\mathfrak{A}^1_{\mathit{full}}$ and $\mathfrak{A}^1_{\mathit{full\text{-}lin}}$ can be used to guide the application of this rule, using the values memoized by the previous applications of **Red**. The very first time, the appropriate h in **Dec** is chosen arbitrarily. In any subsequent application of this rule, h is chosen according to the result of the **Red** rule that has already been applied to the arguments of the current AUT for their consistency check, as required by the condition of **Dec**. To reduce nondeterminism, when **Dec** triggers the application of $\mathfrak{C}$ to Q_1 and Q_2 coming from an AUT $y : Q_1 \triangleq Q_2$, we can apply Red to y in Q_1 and y in Q_2 concurrently, using the same new h in **Red** for both Q_1 and Q_2, memoizing those applications, and continuing to the next step, again working with the pairs of problems. If in this process $\mathfrak{C}$ fails in one of elements of some pair, **Sol** is applied instead of the triggering **Dec**. If at some step of $\mathfrak{C}$ no common h is found for both elements of some pair, we stop (suspend) the application of $\mathfrak{C}$ marking that pair of consistency problems as failure, and apply **Dec** along the just obtained sequence of **Red** steps as long as possible until reaching that failure step. In this way, the applications of **Dec** and **Sol** will correspond to the applications of **Red**. There is a natural correspondence between the applications of **Rem** and **Tri** rules. Therefore, the search tree for $\mathfrak{A}^1_{\mathit{full\text{-}lin}}$ will have the same structure as the search tree for $\mathfrak{C}$ (together with the intersection computations at each node). Hence its complexity of $\mathfrak{A}^1_{\mathit{full\text{-}lin}}$ is $O(n \cdot \Delta \cdot \Delta^{\bullet \mathfrak{a}^{\bullet n}})$.

For $\mathfrak{A}^1_{\mathit{full}}$, we first need to expand the store, which would correspond to resuming $\mathfrak{C}$ to the suspended pair of consistency problems, but this time separately for each of them, not concurrently. Still, the search space for this case is the same as for $\mathfrak{C}$, and the time required for the algorithm including the expansion is $\mathfrak{A}^1_{\mathit{full\text{-}lin}}$ is $O(n \cdot \Delta \cdot \Delta^{\bullet \mathfrak{a}^{\bullet n}})$. Next, we should apply **Mer** exhaustively. The number of AUTs in the store is $O(\mathfrak{a}^{\bullet n})$. A normal form of the store with respect to this rule corresponds to partitioning the store into maximal disjoint subsets so that no partition is covered by another one. By Sperner's theorem [72], the number of all such partitioning is $O\left(\binom{\mathfrak{a}^{\bullet n}}{\lfloor \mathfrak{a}^{\bullet n}/2 \rfloor}\right)$. In the computation of each such partition, we intersect $O(\mathfrak{a}^{\bullet n})$ unsorted sets of terms, where two terms can be compared in $O(n)$ time. The size of the sets is $O(\Delta^{\bullet\mathfrak{a}^{\bullet n}})$. It gives $O(\Delta^{\bullet 2\mathfrak{a}^{\bullet n}} \cdot \mathfrak{a}^{\bullet n} \cdot n)$, which leads to the overall complexity $O\left(\binom{\mathfrak{a}^{\bullet n}}{\lfloor \mathfrak{a}^{\bullet n}/2 \rfloor} \cdot \Delta^{\bullet 2\mathfrak{a}^{\bullet n}} \cdot \mathfrak{a}^{\bullet n} \cdot n\right)$.

$\mathfrak{A}^2_{\mathit{full\text{-}lin}}$ does not call the consistency check, but does the same work as $\mathfrak{C}$ and, hence, has the same complexity $O(n \cdot \Delta \cdot \Delta^{\bullet n})$.

For $\mathfrak{A}^2_{\mathit{full}}$, by reasoning similar to that for $\mathfrak{A}^1_{\mathit{full}}$ we get the complexity $O\left(\binom{n}{\lfloor n/2 \rfloor} \cdot \Delta^{\bullet 2n} \cdot n^2\right)$. □

5.5.6 An extended example

Assume a, b, c, d are constants, f and h are binary function symbols, and g is a unary function symbol. Let $\mathcal{R}$ be defined as $a \sim^{\varnothing}_{\mathcal{R},0.6} b$, $b \sim^{\varnothing}_{\mathcal{R},0.7} c$, $f \sim^{\{(1,1),(2,1)\}}_{\mathcal{R},0.8} h$, and $h \sim^{\{(1,1)\}}_{\mathcal{R},0.9} g$.

Take $\lambda = 0.5$ and consider the anti-unification problem between the terms $f(f(f(a,c), f(a,c)), g(g(a)))$ and $f(g(g(b)), f(f(b,c), f(b,c)))$. The dag representation of the problem is shown in Fig. 5.1.

Each subgraph of this graph is a compact representation of a set of terms that form the proximity class of the corresponding subterm in the problem. For instance, the subgraph at node **3** is a compact representation of the proximity class of the subterm $f(f(a,c), f(a,c))$. The label $\{\langle f, \{(1,1),(2,2)\}, 1\rangle, \langle h, \{(1,1),(1,2)\}, 0.8\rangle\}$ of **3** is the adjacency list of f in $\mathcal{R}$ (containing the argument relation and the proximity degree for each symbol proximal to f).

Algorithm $\mathfrak{A}^1_{\mathit{full\text{-}lin}}$ starts with the configuration

$$\{x : \{f(f(f(a,c), f(a,c)), g(g(a)))\} \triangleq \{f(g(g(b)), f(f(b,c), f(b,c)))\}\};$$
$$\varnothing;\ x;\ 1;\ 1$$

The attempt to apply the **Dec** rule involves checking whether the labels at nodes **1** (i.e., the adjacency list of f) and **2** (the adjacency list of the same f) have a common symbol. There are actually two: f (with the argument relation $\{(1,1),(2,2)\}$) and h (with the argument relation $\{(1,1),(1,2)\}$).

The next step is the consistency check. For the case of f, we should check whether the set of terms at nodes **3** (corresponds to Q_{11} in the **Dec** rule), **5** (Q_{12}), **4** (Q_{21}), and **6** (Q_{22}) are consistent. All these checks are successful. In the process, we can do even more: perform the consistency check concurrently for **3** and **5**, and for **4** and **6** (as these pairs come from

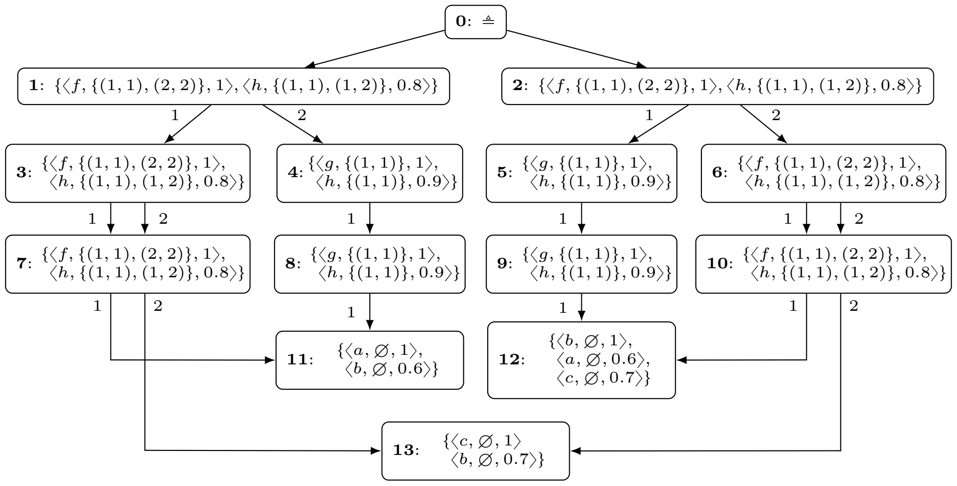

0: ≜
1: {⟨f, {(1, 1), (2, 2)}, 1⟩, ⟨h, {(1, 1), (1, 2)}, 0.8⟩}
2: {⟨f, {(1, 1), (2, 2)}, 1⟩, ⟨h, {(1, 1), (1, 2)}, 0.8⟩}
3: {⟨f, {(1, 1), (2, 2)}, 1⟩, ⟨h, {(1, 1), (1, 2)}, 0.8⟩}
4: {⟨g, {(1, 1)}, 1⟩, ⟨h, {(1, 1)}, 0.9⟩}
5: {⟨g, {(1, 1)}, 1⟩, ⟨h, {(1, 1)}, 0.9⟩}
6: {⟨f, {(1, 1), (2, 2)}, 1⟩, ⟨h, {(1, 1), (1, 2)}, 0.8⟩}
7: {⟨f, {(1, 1), (2, 2)}, 1⟩, ⟨h, {(1, 1), (1, 2)}, 0.8⟩}
8: {⟨g, {(1, 1)}, 1⟩, ⟨h, {(1, 1)}, 0.9⟩}
9: {⟨g, {(1, 1)}, 1⟩, ⟨h, {(1, 1)}, 0.9⟩}
10: {⟨f, {(1, 1), (2, 2)}, 1⟩, ⟨h, {(1, 1), (1, 2)}, 0.8⟩}
11: {⟨a, ∅, 1⟩, ⟨b, ∅, 0.6⟩}
12: {⟨b, ∅, 1⟩, ⟨a, ∅, 0.6⟩, ⟨c, ∅, 0.7⟩}
13: {⟨c, ∅, 1⟩ ⟨b, ∅, 0.7⟩}
1
2

Figure 5.1: Dag representation of the anti-unification problem between $f(f(f(a,c), f(a,c)), g(g(a)))$ and $f(g(g(b)), f(f(b,c), f(b,c)))$.

the same AUTs), and use the same new function symbol in each pair when applying the Red rule. (For instance, we can use h for **3** and **5** as it appears in both nodes.). Repeatedly apply this concurrent check to the children of the involved nodes in the process of showing consistency. Memoize common function symbols as they will be useful in the subsequent applications of Dec. After applying this process as long as possible, in the cash we will have:[5]

3 and **5** are consistent and have the common symbol h,

4 and **6** are consistent and have the common symbol h,

7 and **9** are consistent and have the common symbol h,

8 and **10** are consistent and have the common symbol h,

11 $\cup$ **13** and **12** are consistent and have the common symbol b,

11 and **12** $\cup$ **13** are consistent and have the common symbol b.

It is important to notice that if the consistency check failed for at least one node, e.g., for **9**, then the condition of Dec would fail and this rule would not be applicable for **1** and **2** using f. Then we should try h. If the same

[5]By "**3** is consistent" we actually mean "the set of terms at node **3** is consistent", etc. Consistency of **11** $\cup$ **13** means that the union of term sets at **11** and at **13** is consistent.

thing happens for h as well, then **Dec** is not applicable to **1** and **2** at all and we have to use **Sol**. Another important thing is to see what would happen if a pair of consistent nodes did not have a common symbol: for instance, if **5** and **6** are consistent but do not have a common symbol. In this case, we would cash this info and would not continue to check consistency of the successors of these nodes, i.e., we would not check whether **8** and **10**, and **11** and **12** $\cup$ **13** are consistent.

Coming back to the derivation, we have a new configuration

$$\{y_1 : \{f(f(a,c), f(a,c))\} \triangleq \{g(g(b))\}, y_2 : \{g(g(a))\} \triangleq \{f(f(b,c), f(b,c))\}\};\\ \varnothing;\ f(y_1, y_2);\ 1;\ 1.$$

We select the first AUT and apply **Dec**. It should check whether nodes **3** and **5** have a common symbol. But we already did it in the consistency check and cashed the value. It is h. (If the cashed result told us there is no such common symbol, we would use **Sol** instead of **Dec**.) Subsequently, since in **3** the h comes with $\rho = \{(1,1),(1,2)\}$, we need to check whether the set of the first and second successors of **3** is consistent (the set Q_{11} in **Dec**). As one can see from the shared representation of the graph, this set is just **7**. We already know that it is consistent, because we checked its consistency when we showed that **3** is consistent. Similarly, **9** is consistent (the set Q_{12} in **Dec**). As for Q_{21} and Q_{22}, they both are empty because the second argument of h does not appear in the ρ's at **3** and at **5**. Therefore, the new configuration is

$$\{z_1 : \{f(a,c)\} \triangleq \{g(b)\},\ z_2 : \varnothing \triangleq \varnothing,\ y_2 : \{g(g(a))\} \triangleq \{f(f(b,c), f(b,c))\}\};\\ \varnothing;\ f(h(z_1, z_2), y_2);\ 0.8;\ 0.9.$$

By the **Tri** rule, we can remove z_2:

$$\{z_1 : \{f(a,c)\} \triangleq \{g(b)\},\ y_2 : \{g(g(a))\} \triangleq \{f(f(b,c), f(b,c))\}\};\\ \varnothing;\ f(h(z_1, \_), y_2);\ 0.8;\ 0.9.$$

Now we apply **Dec** to the first AUT. It should check whether the nodes that correspond to the terms in this AUT (i.e., **7** and **9**) have a common symbol. But again, we can retrieve it from the cash. It is h. Based on the ρ's of h in these nodes, we need to check whether the set of the first and second successors of **7**, i.e., **11** $\cup$ **13**, is consistent (the set Q_{11} in **Dec**), and the successor of **9**, i.e., **12**, is consistent (the set Q_{12} in **Dec**). We again reuse the cashed info that we got when we checked the consistency of **3**. Hence, the new configuration is

$$\{u_1 : \{a, c\} \triangleq \{b\},\ u_2 : \varnothing \triangleq \varnothing,\ y_2 : \{g(g(a))\} \triangleq \{f(f(b, c), f(b, c))\}\};$$
$$\varnothing;\ f(h(h(u_1, u_2), \_), y_2);\ 0.8;\ 0.9.$$

By the **Tri** rule, we can remove u_2:

$$\{u_1 : \{a, c\} \triangleq \{b\},\ y_2 : \{g(g(a))\} \triangleq \{f(f(b, c), f(b, c))\}\};$$
$$\varnothing;\ f(h(h(u_1, \_), \_), y_2);\ 0.8;\ 0.9.$$

Applying **Dec** to the first AUT, we check what is the common symbol between the nodes that correspond the terms there: **11** ∪ **13** and **12**. The cashed result tells us that it is b. No further consistency checks are needed because of the empty ρ it has. We get

$$\{y_2 : \{g(g(a))\} \triangleq \{f(f(b, c), f(b, c))\}\};\ \varnothing;\ f(h(h(b, \_), \_), y_2);\ 0.6;\ 0.9$$

and continue in the similar manner:

$$\{y_2 : \{g(g(a))\} \triangleq \{f(f(b, c), f(b, c))\}\};\ \varnothing;\ f(h(h(b, \_), \_), y_2);\ 0.6;\ 0.9 \Longrightarrow_{\text{Dec}}$$
$$\{v_1 : \{g(a)\} \triangleq \{f(b, c)\},\ v_2 : \varnothing \triangleq \varnothing\};\ \varnothing;$$
$$f(h(h(b, \_), \_), h(v_1, v_2));\ 0.6;\ 0.8 \Longrightarrow_{\text{Tri}}$$
$$\{v_1 : \{g(a)\} \triangleq \{f(b, c)\}\};\ \varnothing;\ f(h(h(b, \_), \_), h(v_1, \_));\ 0.6;\ 0.8 \Longrightarrow_{\text{Dec}}$$
$$\{w_1 : \{a\} \triangleq \{b, c\},\ w_2 : \varnothing \triangleq \varnothing\};\ \varnothing;$$
$$f(h(h(b, \_), \_), h(h(w_1, w_2), \_));\ 0.6;\ 0.8 \Longrightarrow_{\text{Tri}}$$
$$\{w_1 : \{a\} \triangleq \{b, c\}\};\ \varnothing;\ f(h(h(b, \_), \_), h(h(w_1, \_), \_));\ 0.6;\ 0.7 \Longrightarrow_{\text{Dec}}$$
$$\varnothing;\ \varnothing;\ f(h(h(b, \_), \_), h(h(b, \_), \_));\ 0.6;\ 0.7.$$

This is the first terminal configuration. Remember that in the first decomposition step, we had an alternative in choosing h instead of f. Exploring it, we start with the step:

$$\{x : \{f(f(f(a, c), f(a, c)), g(g(a)))\} \triangleq \{f(g(g(b)), f(f(b, c), f(b, c)))\}\};$$
$$\varnothing;\ x;\ 1;\ 1 \Longrightarrow_{\text{Dec}}$$
$$\{y_1 : \{f(f(a, c), f(a, c)), g(g(a))\} \triangleq \{g(g(b)), f(f(b, c), f(b, c))\},$$
$$y_2 : \varnothing \triangleq \varnothing\}; \varnothing;\ h(y_1, y_2);\ 0.8;\ 0.8.$$

(To perform this step, we had to make sure that both $\mathbf{3} \cup \mathbf{4}$ and $\mathbf{5} \cup \mathbf{6}$ are $(\mathcal{R}, \lambda)$-consistent.) Continuing further, we reach the next terminal configuration:

$$\begin{aligned}&\{y_1 : \{f(f(a,c), f(a,c)), g(g(a))\} \triangleq \{g(g(b)), f(f(b,c), f(b,c))\},\\ &\quad y_2 : \varnothing \triangleq \varnothing\}; \varnothing;\ h(y_1, y_2);\ 0.8;\ 0.8 \Longrightarrow^*\\ &\varnothing;\ \varnothing;\ h(h(h(b,\_),\_),\_);\ 0.6;\ 0.7.\end{aligned}$$

Hence, we got two answers computed by $\mathfrak{A}^1_{\mathit{full\text{-}lin}}$:

$$f(h(h(b,\_),\_), h(h(b,\_))); 0.6; 0.7, \qquad h(h(h(b,\_),\_),\_), 0.6, 0.7.$$

$\mathfrak{A}^1_{\mathit{full}}$ gives the same answers, since the store is empty: no merging is needed.

5.6 Concluding remarks

We designed class-based unification, matching, and anti-unification algorithms for proximity relations in fully fuzzy signatures, where mismatches are permitted not only in symbol names but also in their arities, and proved their termination, soundness, and completeness.

Proximity between arguments of distinct function symbols is expressed by argument relations. The diagram below illustrates connections between versions of argument relations. These connections are also reflected in the problems that use these relations. The arrows in the diagram indicate the direction from more general cases to more specific ones.

$$\begin{tikzcd}\boxed{\text{unrestricted relations}} \arrow[r] \arrow[d] & \boxed{\text{correspondence relations}} \arrow[d] \\ \boxed{\text{unrestricted mappings}} \arrow[r] & \boxed{\text{correspondence mappings}}\end{tikzcd}$$

For unification, we require the argument relations to be correspondence relations in order not to have arguments skipped. The results are naturally valid for correspondence mappings as well, i.e., we covered the right column in the diagram. For matching, there is no restriction: we can use arbitrary argument relations (and, in particular, mappings).

To compare with related work, we note that on one hand, our unification and matching results can be seen as more general than those from [2] because we relax the similarity relation to proximity and do not require the argument relations to be necessarily mappings. On the other hand, for unification we have correspondence relations, that are left- and right-total, while in [2] the argument relations are total only on the smaller side. However, the requirement of using correspondence relations is not really a restriction since any argument relation can be turned into a correspondence by adding dummy arguments. Moreover, these results also extend those from Section 4.2 and Section 4.3, where the argument relations are the identity relations.

The anti-unification algorithms illustrate more fine-grained distinction depending on the used relations. For the unrestricted cases (left column), we compute mcsrg's. For correspondence relations and correspondence mappings (right column), we compute mcsg's. (In fact, for them, the notions of mcsrg and mcsg coincide). The algorithms for relations (upper row) are more involved than those for mappings (lower row): Those for relations deal with AUTs containing arbitrary sets of terms, while for mappings, those sets have cardinality at most one, thus simplifying the conditions in the rules. Moreover, the two cases in the lower row generalize the existing anti-unification problems:

- the unrestricted mappings case generalizes the problem from [2] by extending similarity to proximity and relaxing the smaller-side-totality restriction;
- the correspondence mappings case generalizes the problem from Section 4.4 by allowing permutations between arguments of proximal function symbols.

Moreover, the described anti-unification algorithms are modular and can be used to compute linear generalizations by just skipping the merging rule.

All our algorithms (unification, matching, anti-unification) can be easily turned into the corresponding algorithms for crisp tolerance relations by taking lambda-cuts and ignoring the computation of the approximation degrees.

We did not consider cases when two symbols can be related to each other by more than one argument relation, or whether a function symbol can be related to itself with a relation other than the identity. Our results can be extended to them, thus opening a way towards approximate unification, matching, and anti-unification modulo background theories specified by shallow collapse-free axioms. Note that the corresponding crisp unification has been studied quite intensively, see, e.g., [71, 15]. Another interesting direction of future work would be extending our results to quantitative algebras [52] that also deal with quantitative extensions of equality.

Chapter 6

Proximity Relations in Rule-Based Programming

In this chapter we describe an extension of a rule-based programming formalism ρLog with proximity relations. The extension combines the power of conditional transformation rules with approximate unranked pattern matching. The obtained calculus, called ρLog-prox, is suitable for nondeterministic computations in both crisp and fuzzy settings, controlled by strategies.

6.1 Introduction to ρLog

ρLog [55] is a calculus for conditional transformation of sequences of expressions, controlled by strategies. It originated from experiments aiming at extending the language of symbolic computation system Mathematica [80] by a rule-based programming package [54, 56]. Meanwhile there are some tools based on or influenced by ρLog, such as its implementation in Mathematica [53], an extension of Prolog, called PρLog [25], or an extension of Maple, called symbtrans [11].

The language of ρLog is richer than what we considered in the previous chapters. 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 [51]. Program meaning is characterized by logic programming semantics. Rules and strategies are formulated as clauses.

ρLog-based/inspired tools have been used in the extraction of frequent patterns from data mining workflows [60], for automatic derivation of multiscale models of arrays of micro- and nanosystems [81], modeling rewriting strategies [22], etc.

At the core of ρLog there is a powerful pattern matching algorithm [44]. 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 the 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., [30, 59, 58, 27, 50, 69, 65, 32, 33].

ρLog-prox, described in this chapter, is an attempt to address this problem by combining approximate reasoning and strategic rule-based programming. It extends ρLog with capabilities to process imprecise information represented by proximity relations. 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 chapter is organized as follows: In Section 6.2 we introduce the additional terminology and define our language. Section 6.3 is about the basics of ρLog-prox: its syntax, semantics, and an illustrative example are presented. In Section 6.4, we develop an algorithm for solving proximity matching problems and prove its properties. Section 6.5 is the conclusion.

6.2 Notions and terminology

In this section, we introduce the additional notions needed in the rest of the chapter.

Terms, hedges, contexts, substitutions

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

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

Besides, $\mathcal{A}$ also contains auxiliary symbols such as parenthesis and comma, and a special constant $\circ$, called *hole*. A *variable* is an element of the set $\mathcal{V} = \mathcal{V}_\mathsf{T} \cup \mathcal{V}_\mathsf{S} \cup \mathcal{V}_\mathsf{F} \cup \mathcal{V}_\mathsf{C}$. 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 $\mathcal{A}$ as follows:

$t ::= x \mid f(\tilde{s}) \mid X(\tilde{s}) \mid \overline{X}(t)$	Term
$\tilde{t} ::= t_1, \ldots, t_n \quad (n \geqslant 0)$	Term sequence
$s ::= t \mid \overline{x}$	Hedge element
$\tilde{s} ::= s_1, \ldots, s_n \quad (n \geqslant 0)$	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 $\mathcal{T}(\mathcal{F}, \mathcal{V})$, hedges by $\mathcal{H}(\mathcal{F}, \mathcal{V})$, and contexts by $\mathcal{C}(\mathcal{F}, \mathcal{V})$. Ground (i.e., variable-free) subsets of these sets are denoted by $\mathcal{T}(\mathcal{F})$, $\mathcal{H}(\mathcal{F})$, and $\mathcal{C}(\mathcal{F})$, 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 just _ for an anonymous term and function variable, and __ for an anonymous hedge and context variable. The set of anonymous variables is denoted by $\mathcal{V}_{\mathsf{An}}$.

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

We also introduce two notations: $\mathcal{V}(E)$ denotes the set of variables occurring in expression E, and $\mathcal{V}(E, \{p_1, ..., p_n\})$, where p_i's are positions in E, is defined as $\mathcal{V}(E, \{p_1, ..., p_n\}) = \cup_{i=1}^{n} \mathcal{V}(E|_{p_i})$, 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 $\mathcal{V}$ to $\mathcal{T}(\mathcal{F}, \mathcal{V}) \cup \mathcal{H}(\mathcal{F}, \mathcal{V}) \cup \mathcal{C}(\mathcal{F}, \mathcal{V}) \cup \mathcal{F} \cup \mathcal{V}_{\mathsf{F}}$, defined as

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

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 be extended to elements of the set $\mathcal{T}(\mathcal{F}, \mathcal{V}) \cup \mathcal{H}(\mathcal{F}, \mathcal{V}) \cup \mathcal{C}(\mathcal{F}, \mathcal{V}) \cup \mathcal{F} \cup \mathcal{V}_{\mathsf{F}}$ in the following way:

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

Proximity relations

Given a proximity relation $\mathcal{R}$ defined on $\mathcal{F}$, we extend it to $\mathcal{F} \cup \mathcal{V} \cup \mathcal{T}(\mathcal{F}, \mathcal{V}) \cup \mathcal{H}(\mathcal{F}, \mathcal{V}) \cup \mathcal{C}(\mathcal{F}, \mathcal{V})$:

- For variables, $V \in \mathcal{V}$:

  - $\mathcal{R}(V, V) = 1$.
- For terms, $t, t' \in \mathcal{T}(\mathcal{F}, \mathcal{V})$:
  - If $t = F(\tilde{s})$ and $t' = F'(\tilde{s}')$, then $\mathcal{R}(t, t') = \mathcal{R}(F, F') \wedge \mathcal{R}(\tilde{s}, \tilde{s}')$.
- For hedges, $s, s' \in \mathcal{H}(\mathcal{F}, \mathcal{V})$:
  - 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 $\mathcal{R}(\tilde{s}, \tilde{s}') = \mathcal{R}(s_1, s'_1) \wedge \cdots \wedge \mathcal{R}(s_n, s'_n)$.
- For contexts, $C, C' \in \mathcal{C}(\mathcal{F}, \mathcal{V})$:
  - $\mathcal{R}(\circ, \circ) = 1$.
  - If C and C' have the same number of arguments and their context arguments appear in the same position, e.g., $C = F(\tilde{s}_1, C_1, \tilde{s}_2)$ and $C' = F'(\tilde{s}'_1, C'_1, \tilde{s}'_2)$, then $\mathcal{R}(C, C') = \mathcal{R}(F, F') \wedge \mathcal{R}(\tilde{s}_1, \tilde{s}'_1) \wedge \mathcal{R}(C_1, C'_1) \wedge \mathcal{R}(\tilde{s}_2, \tilde{s}'_2)$.
- In all other cases, $\mathcal{R}(E, E') = 0$ for two syntactic expressions $E, E' \in \mathcal{V} \cup \mathcal{T}(\mathcal{F}, \mathcal{V}) \cup \mathcal{H}(\mathcal{F}, \mathcal{V}) \cup \mathcal{C}(\mathcal{F}, \mathcal{V})$.

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

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

6.3 The syntax of ρLog-prox

Syntactic matching and proximity matching problems

The extensions of syntactic matching and proximity matching to the expressions defined in this chapter are straightforward.

Example 6.3.1. The syntactic matching atom

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

has two solutions:

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

Example 6.3.2. Let the proximity relation $\mathcal{R}$ be given by the following:

$$\begin{aligned}
&\mathcal{R}(g_1, h_1) = \mathcal{R}(g_2, h_1) = 0.4\\
&\mathcal{R}(g_1, h_2) = \mathcal{R}(g_2, h_2) = 0.5\\
&\mathcal{R}(g_2, h_3) = \mathcal{R}(g_3, h_3) = 0.6\\
&\mathcal{R}(a, b) = 0.7
\end{aligned}$$

Let the proximity atom be

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

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.)

$$\begin{aligned}
&\lambda = 0.4 :\\
&\quad \sigma_1 = \{\overline{x} \mapsto (), x \mapsto h_1(a), \overline{Y} \mapsto \circ, \overline{z} \mapsto f(g_3(a))\}\\
&\quad \sigma_2 = \{\overline{x} \mapsto (), x \mapsto h_2(a), \overline{Y} \mapsto \circ, \overline{z} \mapsto f(g_3(a))\}\\
&\quad \sigma_3 = \{\overline{x} \mapsto (), x \mapsto h_1(b), \overline{Y} \mapsto \circ, \overline{z} \mapsto f(g_3(a))\}\\
&\quad \sigma_4 = \{\overline{x} \mapsto (), x \mapsto h_2(b), \overline{Y} \mapsto \circ, \overline{z} \mapsto f(g_3(a))\}\\
&\quad \sigma_5 = \{\overline{x} \mapsto (), x \mapsto h_1(a), \overline{Y} \mapsto \circ, \overline{z} \mapsto f(g_3(b))\}\\
&\quad \sigma_6 = \{\overline{x} \mapsto (), x \mapsto h_2(a), \overline{Y} \mapsto \circ, \overline{z} \mapsto f(g_3(b))\}\\
&\quad \sigma_7 = \{\overline{x} \mapsto (), x \mapsto h_1(b), \overline{Y} \mapsto \circ, \overline{z} \mapsto f(g_3(b))\}\\
&\quad \sigma_8 = \{\overline{x} \mapsto (), x \mapsto h_2(b), \overline{Y} \mapsto \circ, \overline{z} \mapsto f(g_3(b))\}\\
&\quad \sigma_9 = \{\overline{x} \mapsto (), x \mapsto h_1(a), \overline{Y} \mapsto \circ, \overline{z} \mapsto f(h_3(a))\}\\
&\quad \sigma_{10} = \{\overline{x} \mapsto (), x \mapsto h_2(a), \overline{Y} \mapsto \circ, \overline{z} \mapsto f(h_3(a))\}\\
&\quad \sigma_{11} = \{\overline{x} \mapsto (), x \mapsto h_1(b), \overline{Y} \mapsto \circ, \overline{z} \mapsto f(h_3(a))\}\\
&\quad \sigma_{12} = \{\overline{x} \mapsto (), x \mapsto h_2(b), \overline{Y} \mapsto \circ, \overline{z} \mapsto f(h_3(a))\}\\
&\quad \sigma_{13} = \{\overline{x} \mapsto (), x \mapsto h_1(a), \overline{Y} \mapsto \circ, \overline{z} \mapsto f(h_3(b))\}\\
&\quad \sigma_{14} = \{\overline{x} \mapsto (), x \mapsto h_2(a), \overline{Y} \mapsto \circ, \overline{z} \mapsto f(h_3(b))\}\\
&\quad \sigma_{15} = \{\overline{x} \mapsto (), x \mapsto h_1(b), \overline{Y} \mapsto \circ, \overline{z} \mapsto f(h_3(b))\}\\
&\quad \sigma_{16} = \{\overline{x} \mapsto (), x \mapsto h_2(b), \overline{Y} \mapsto \circ, \overline{z} \mapsto f(h_3(b))\}\\
&\quad \sigma_{17} = \{\overline{x} \mapsto g_1(a), x \mapsto h_3(a), \overline{Y} \mapsto f(\circ), \overline{z} \mapsto ()\}
\end{aligned}$$

$$\sigma_{18} = \{\overline{x} \mapsto g_1(a), x \mapsto h_3(b), \overline{Y} \mapsto f(\circ), \overline{z} \mapsto ()\}$$
$$\sigma_{19} = \{\overline{x} \mapsto g_1(b), x \mapsto h_3(a), \overline{Y} \mapsto f(\circ), \overline{z} \mapsto ()\}$$
$$\sigma_{20} = \{\overline{x} \mapsto g_1(b), x \mapsto h_3(b), \overline{Y} \mapsto f(\circ), \overline{z} \mapsto ()\}$$
$$\sigma_{21} = \{\overline{x} \mapsto h_1(a), x \mapsto h_3(a), \overline{Y} \mapsto f(\circ), \overline{z} \mapsto ()\}$$
$$\sigma_{22} = \{\overline{x} \mapsto h_1(a), x \mapsto h_3(b), \overline{Y} \mapsto f(\circ), \overline{z} \mapsto ()\}$$
$$\sigma_{23} = \{\overline{x} \mapsto h_1(b), x \mapsto h_3(a), \overline{Y} \mapsto f(\circ), \overline{z} \mapsto ()\}$$
$$\sigma_{24} = \{\overline{x} \mapsto h_1(b), x \mapsto h_3(b), \overline{Y} \mapsto f(\circ), \overline{z} \mapsto ()\}$$
$$\sigma_{25} = \{\overline{x} \mapsto h_2(a), x \mapsto h_3(a), \overline{Y} \mapsto f(\circ), \overline{z} \mapsto ()\}$$
$$\sigma_{26} = \{\overline{x} \mapsto h_2(a), x \mapsto h_3(b), \overline{Y} \mapsto f(\circ), \overline{z} \mapsto ()\}$$
$$\sigma_{27} = \{\overline{x} \mapsto h_2(b), x \mapsto h_3(a), \overline{Y} \mapsto f(\circ), \overline{z} \mapsto ()\}$$
$$\sigma_{28} = \{\overline{x} \mapsto h_2(b), x \mapsto h_3(b), \overline{Y} \mapsto f(\circ), \overline{z} \mapsto ()\}$$

$\lambda = 0.5$:

$$\sigma_{1} = \{\overline{x} \mapsto (), x \mapsto h_2(a), \overline{Y} \mapsto \circ, \overline{z} \mapsto f(g_3(a))\}$$
$$\sigma_{2} = \{\overline{x} \mapsto (), x \mapsto h_2(b), \overline{Y} \mapsto \circ, \overline{z} \mapsto f(g_3(a))\}$$
$$\sigma_{3} = \{\overline{x} \mapsto (), x \mapsto h_2(a), \overline{Y} \mapsto \circ, \overline{z} \mapsto f(g_3(b))\}$$
$$\sigma_{4} = \{\overline{x} \mapsto (), x \mapsto h_2(b), \overline{Y} \mapsto \circ, \overline{z} \mapsto f(g_3(b))\}$$
$$\sigma_{5} = \{\overline{x} \mapsto (), x \mapsto h_2(a), \overline{Y} \mapsto \circ, \overline{z} \mapsto f(h_3(a))\}$$
$$\sigma_{6} = \{\overline{x} \mapsto (), x \mapsto h_2(b), \overline{Y} \mapsto \circ, \overline{z} \mapsto f(h_3(a))\}$$
$$\sigma_{7} = \{\overline{x} \mapsto (), x \mapsto h_2(a), \overline{Y} \mapsto \circ, \overline{z} \mapsto f(h_3(b))\}$$
$$\sigma_{8} = \{\overline{x} \mapsto (), x \mapsto h_2(b), \overline{Y} \mapsto \circ, \overline{z} \mapsto f(h_3(b))\}$$
$$\sigma_{9} = \{\overline{x} \mapsto g_1(a), x \mapsto h_3(a), \overline{Y} \mapsto f(\circ), \overline{z} \mapsto ()\}$$
$$\sigma_{10} = \{\overline{x} \mapsto g_1(a), x \mapsto h_3(b), \overline{Y} \mapsto f(\circ), \overline{z} \mapsto ()\}$$
$$\sigma_{11} = \{\overline{x} \mapsto g_1(b), x \mapsto h_3(a), \overline{Y} \mapsto f(\circ), \overline{z} \mapsto ()\}$$
$$\sigma_{12} = \{\overline{x} \mapsto g_1(b), x \mapsto h_3(b), \overline{Y} \mapsto f(\circ), \overline{z} \mapsto ()\}$$
$$\sigma_{13} = \{\overline{x} \mapsto h_2(a), x \mapsto h_3(a), \overline{Y} \mapsto f(\circ), \overline{z} \mapsto ()\}$$
$$\sigma_{14} = \{\overline{x} \mapsto h_2(a), x \mapsto h_3(b), \overline{Y} \mapsto f(\circ), \overline{z} \mapsto ()\}$$
$$\sigma_{15} = \{\overline{x} \mapsto h_2(b), x \mapsto h_3(a), \overline{Y} \mapsto f(\circ), \overline{z} \mapsto ()\}$$
$$\sigma_{16} = \{\overline{x} \mapsto h_2(b), x \mapsto h_3(b), \overline{Y} \mapsto f(\circ), \overline{z} \mapsto ()\}$$

$$\begin{aligned}
\lambda = 0.6 : & \\
& \sigma_1 = \{\overline{x} \mapsto g_1(a), x \mapsto h_3(a), \overline{Y} \mapsto f(\circ), \overline{z} \mapsto ()\} \\
& \sigma_2 = \{\overline{x} \mapsto g_1(a), x \mapsto h_3(b), \overline{Y} \mapsto f(\circ), \overline{z} \mapsto ()\} \\
& \sigma_3 = \{\overline{x} \mapsto g_1(b), x \mapsto h_3(a), \overline{Y} \mapsto f(\circ), \overline{z} \mapsto ()\} \\
& \sigma_4 = \{\overline{x} \mapsto g_1(b), x \mapsto h_3(b), \overline{Y} \mapsto f(\circ), \overline{z} \mapsto ()\} \\
\lambda = 0.7 : & \quad \text{No solutions.}
\end{aligned}$$

Note that because of strictness, syntactic matching can be seen as special proximity matching for an arbitrary $\mathcal{R}$ with the lambda-cut equal to 1.

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 $B_1, \dots, B_n$. 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 $\mathcal{R}$, the set of (f, g) pairs with $\mathcal{R}(f, g) > 0$ is finite.

Moreover, with each query, a global cut value $\omega \in (0, 1]$ is provided. The meaning of this value is that it sets the lower bound of the approximation degrees of computed answers.

For working with proximity relations, we introduced a special predefined strategy **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 matching problem $\tilde{s}_2 \ll_{\mathcal{R},\max(\lambda,\omega)} \tilde{s}_1$ is solvable for the given $\mathcal{R}$. When $\lambda = 1$, **prox** coincides with the identity strategy **id** of the original ρLog [55] (the strictness assumption is important here). When $\lambda = 0$, matching is done by taking into account only the global cut value ω. For brevity, we may skip the argument in this case and write $\mathbf{prox} :: \tilde{s}_1 \Longrightarrow \tilde{s}_2$ instead of $\mathbf{prox}(0) :: \tilde{s}_1 \Longrightarrow \tilde{s}_2$.

Providing λ as the argument of **prox** gives a flexibility to the user to require the approximation degree for some problems higher than what ω would guarantee: note $\max(\lambda, \omega)$ in the matching problem for **prox**. On the other hand, the same $\max(\lambda, \omega)$ makes sure that a lower value of λ is overridden by a higher value of ω.

For the original version of ρLog, semantics of programs can be defined in the same way as it is done for logic programming [5, 51]. 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, a proximity relation $\mathcal{R}$, and the lower bound ω for the answer approximation are given.

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

$$\frac{\mathit{str}_{\mathrm{q}} :: \mathit{lhs}_{\mathrm{q}} \Longrightarrow \mathit{rhs}_{\mathrm{q}}, Q \qquad\qquad \mathit{str}_{\mathrm{p}} :: \mathit{lhs}_{\mathrm{p}} \Longrightarrow \mathit{rhs}_{\mathrm{p}} \leftarrow \mathit{Body}}{(\mathit{Body},\ \mathbf{prox}(1) :: \mathit{rhs}_{\mathrm{p}} \Longrightarrow \mathit{rhs}_{\mathrm{q}},\ Q)\sigma},$$

where σ is a solution of the proximity matching problem $\{\mathit{str}_{\mathrm{p}} \ll_{\mathcal{R},1} \mathit{str}_{\mathrm{q}}, \mathit{lhs}_{\mathrm{p}} \ll_{\mathcal{R},1} \mathit{lhs}_{\mathrm{q}}\}$. The strategy $\mathit{str}_{\mathrm{q}}$ 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:

$$\frac{\mathbf{prox}(\lambda) :: \mathit{lhs}_{\mathrm{q}} \Longrightarrow \mathit{rhs}_{\mathrm{q}},\ Q}{Q\sigma},$$

where σ is a solution of the $(\mathcal{R}, \max(\lambda, \omega))$-proximity matching problem $\{\mathit{rhs}_{\mathrm{q}} \ll_{\mathcal{R},\max(\lambda,\omega)} \mathit{lhs}_{\mathrm{q}}\}$.

A *derivation* of a query Q from a program P (with respect to a proximity relation $\mathcal{R}$) with the approximation degree at least ω is a sequence of triples $\langle Q_0, \vartheta_0, \delta_0\rangle$, $\langle Q_1, \vartheta_1, \delta_1\rangle, \ldots$, where

- $Q_0 = Q$, $\vartheta_0 = \mathit{Id}$, and $\delta_0 = 1$;

- Q_i is a query obtained from Q_{i-1} by resolution or proximity factoring;
- $\vartheta_i = \vartheta_{i-1}\sigma$ and $\delta_i = \delta_{i-1} \wedge \alpha$, where σ is computed at the application of resolution or proximity factoring when obtaining Q_i from Q_{i-1}, and α is the approximation degree of σ. (For resolution, $\alpha = 1$. For proximity factoring, $\alpha \geqslant \max(\lambda, \omega)$.)

A derivation is *successful* if it ends with $\langle Q_n, \vartheta_n, \delta_n\rangle$, where Q_n is the empty query. Then the *answer computed for Q via P* by this derivation is the pair (ϑ, δ), where ϑ is the restriction of ϑ_n to the variables of Q and $\delta = \delta_n$. (Sometimes we refer only to the substitution as the computed answer.) By construction, $\delta \geqslant \omega$. A derivation is *failed*, if none of the inference rules can apply to the last query. Like for the original ρLog, the inference system is sound: the computed answers are also correct with respect to 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 [5, 51, 6]. *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 [42, 21]), we work with well-moded programs and queries.

Definition 6.3.1 (Well-moded clauses, programs, queries)**.** *Let C be a (normal) clause*

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

where for each $1 \leqslant i \leqslant n$, 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 $1 \leqslant i \leqslant n+1$, we have*

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

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

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

Example 6.3.3. In this rather extended example we illustrate ρLog-prox clauses, strategies, and evaluation mechanism. We borrow the material from [25] 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 duplicates merging. The following strategy implements the idea:

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

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 $\overline{x}$ stands for the subhedge before the first occurrence of x, $\overline{y}$ takes the subhedge between two occurrences of x, and $\overline{z}$ 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 $(a,\, b,\, c,\, b,\, a)$:

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

To this query, ρLog-prox returns two answers: $\{\overline{x} \mapsto (a,\, b,\, c,\, b)\}$ and $\{\overline{x} \mapsto (a,\, b,\, c,\, a)\}$. 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:

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

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. Note that **prox** does not have an argument (equivalent to **prox**(0)), which means that the lower bound of the approximation degree will be taken from the query.

Assume now that in the proximity relation $\mathcal{R}$, we have $\mathcal{R}(a,e) = 0.6$ and $\mathcal{R}(b,d) = 0.7$. Let $\omega = 0.8$. Then the query

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

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 $\omega = 0.7$, we get a single answer: $\{\overline{x} \mapsto (a,\ b,\ c,\ e)\}$ with the computed approximation degree 0.7. Decreasing ω further to 0.6 will lead via backtracking to the following two answers: $\{\overline{x} \mapsto (a,\ b,\ c,\ d)\}$ (with degree 0.6) and $\{\overline{x} \mapsto (a,\ b,\ c,\ e)\}$ (with degree 0.7).

A duplicates-free hedge is a normal form of the single-step transformation merge_duplicates. 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 in the following clause:

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

The effect of **nf** is that it applies merge_duplicates to $\overline{x}$, 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 $\overline{y}$.

Now, for $\omega = 0.6$ and the query

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

we get a single answer $\{\overline{x} \mapsto (a,\ b,\ c)\}$ with degree 0.6. However, procedurally, this answer can be computed multiple times (via backtracking). To avoid such multiple computations, we can use another predefined strategy **first_one**:

$$\begin{aligned} &\text{merge\_all\_duplicates} :: \overline{x} \Longrightarrow \overline{y} \ \leftarrow \\ &\qquad\qquad \mathbf{first\_one}(\mathbf{nf}(\text{merge\_duplicates})) :: \overline{x} \Longrightarrow \overline{y}. \end{aligned}$$

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) and returns (by instantiating $\overline{y}$) only one result of its application to $\overline{x}$.

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 strategy it is pretty easy to define a transformation that merges neighboring branches in a tree, which are approximately the same:

$$\begin{aligned}&\text{merge\_duplicate\_branches} :: \overline{X}(Y(\overline{x})) \Longrightarrow \overline{X}(Y(\overline{y})) \;\leftarrow\\ &\qquad\qquad \text{merge\_duplicates} :: \overline{x} \Longrightarrow \overline{y}.\end{aligned}$$

Now, we can ask to merge neighboring branches in a given tree, which are at least 0.6-approximate of each other (for the same $\mathcal{R}$ as above) by setting $\omega = 0.6$ and evaluating the query

$$\begin{aligned}&\text{merge\_duplicate\_branches} ::\\ &\qquad f(g(a,b,e,h(c,c)),\ h(c),\ g(a,b,d,h(c))) \Longrightarrow x.\end{aligned}$$

ρLog-prox computes three answers together with their approximation degrees:

$$\begin{aligned}&\{x \mapsto f(g(a,b,h(c,c)),\ h(c),\ g(a,b,d,h(c)))\},\ 0.6,\\ &\{x \mapsto f(g(a,b,e,h(c)),\ h(c),\ g(a,b,d,h(c)))\},\ 1,\\ &\{x \mapsto f(g(a,b,e,h(c,c)),\ h(c),\ g(a,b,h(c)))\},\ 0.7.\end{aligned}$$

To obtain the first one, ρLog-prox matched the context variable $\overline{X}$ to the context $f(\circ,\ h(c), g(a,b,d,h(c)))$, the function variable Y to the function symbol g, and the hedge variable $\overline{x}$ to the hedge $(a,b,e,h(c,c))$. Then merge_duplicates 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 a variable. One can have an arbitrary hedge there. For instance, we may be interested in trees that contain $h(c,c)$ (with $\omega = 0.6$):

$$\begin{aligned}&\text{merge\_duplicate\_branches} ::\\ &\qquad f(g(a,b,e,h(c,c)),\ h(c),\ g(a,b,d,h(c))) \Longrightarrow \overline{X}(h(c,c)).\end{aligned}$$

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

$$\{\overline{X} \mapsto f(g(a,b,\circ),\ h(c),\ g(a,b,d,h(c)))\},\ 0.6,$$

$$\{\overline{X} \mapsto f(g(a,b,e,\circ),\ h(c),\ g(a,b,h(c)))\},\ 0.7.$$

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:

$$\begin{aligned}&\text{merge\_all\_duplicate\_branches} :: x \Longrightarrow y \leftarrow\\ &\qquad \textbf{first\_one}(\textbf{nf}(\text{merge\_duplicate\_branches})) :: x \Longrightarrow y.\end{aligned}$$

For the query (with $\omega = 0.6$)

$$\begin{aligned}&\text{merge\_all\_duplicate\_branches} ::\\ &\qquad f(g(a,b,e,h(c,c)),\ h(c),\ g(a,e,b,h(c))) \Longrightarrow \overline{x}.\end{aligned}$$

we get a single answer $\{\overline{x} \mapsto f(g(a,b,h(c)),h(c))\}$ with degree 0.6.

Example 6.3.4. Due to the non-transitivity of proximity relations and the way how matchers are computed from left to right, rearranging the sequence elements in the query for merge_all_duplicates from the previous example affects the computed answers. For instance, if $\mathcal{R}(a,b) = 0.7$, $\mathcal{R}(b,c) = 0.6$ and $\omega = 0.5$, then for the query

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

ρLog-prox computes the answer $\{\overline{x} \mapsto (a,c)\}$ with the approximation degree 0.7. If we take

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

then the answer is $\{\overline{x} \mapsto b\}$ with the approximation degree 0.6. See also [20] for similar examples.

6.4 Solving proximity matching problems

As one could see in the previous section, the inference rules of ρLog-prox heavily rely on solving proximity matching problems. Well-modedness guarantees that only proximity matching 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.

We describe in this section an algorithm $\mathfrak{M}_\rho$, based on the matching algorithm $\mathfrak{M}$ from section 4.3. $\mathfrak{M}_\rho$ computes not only solutions to proximity matching problems, but also the degrees 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 $\sigma_S := \{V_i \mapsto E_i \mid V_i \approx E_i \in S\}$.

While in $\mathfrak{M}$ we have rules that compute the solutions in a compact form, as proximity classes, the rules of $\mathfrak{M}_\rho$, as they are defined bellow, will generate through branching all individual solutions, as well as their proximity degrees. We have six success and four failure rules:

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

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

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

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

TVE: Term variable elimination

$\{x \ll_{\mathcal{R},\lambda} t\} \uplus M;\ S;\ \alpha \Longrightarrow M;\ S \cup \{x \approx t'\};\ \alpha \wedge \beta$,
where $\mathcal{R}(t', t) = \beta \geqslant \lambda$.

HVE: Hedge variable elimination

$\{(\overline{x}, \tilde{s}) \ll_{\mathcal{R},\lambda} (\tilde{t}_1, \tilde{t}_2)\} \uplus M;\ S;\ \alpha \Longrightarrow M \cup \{\tilde{s} \ll_{\mathcal{R},\lambda} \tilde{t}_2\};\ S \cup \{\overline{x} \approx \tilde{t}'_1\};\ \alpha \wedge \beta$,
where $\mathcal{R}(\tilde{t}'_1, \tilde{t}_1) = \beta \geqslant \lambda$.

Cla1: Clash 1

$\{f(\tilde{s}) \ll_{\mathcal{R},\lambda} g(\tilde{t})\} \uplus M;\ S;\ \alpha \Longrightarrow \bot, \qquad \text{if } \mathcal{R}(f, g) < \lambda.$

Cla2: Clash 2

$\{(t, \tilde{s}) \ll_{\mathcal{R},\lambda} ()\} \uplus M;\ S;\ \alpha \Longrightarrow \bot.$

Cla3: Clash 3

$\{() \ll_{\mathcal{R},\lambda} (t, \tilde{t})\} \uplus M;\ S;\ \alpha \Longrightarrow \bot.$

Inc: Inconsistency

$M;\ S;\ \alpha \Longrightarrow \bot$,
if S contains two equations with the same variable in the left hand side.

For solving a proximity matching problem M, we create the initial configuration $M; \varnothing; 1$ and start applying the rules exhaustively. If the same configuration can be transformed by multiple rules, they are applied concurrently, except when 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 $\mathfrak{M}_\rho$.

Theorem 6.4.1 (Termination)**.** *The proximity matching algorithm $\mathfrak{M}_\rho$ terminates. Each final configuration has the form either $\bot$ or $\varnothing; S; \alpha$, where S is in matching solved form.*

Proof. Let $size(E)$ denote the number of symbols in an expression E. By $Msize(M)$ we denote the multiset $\{size(E_2) \mid E_1 \ll_{\mathcal{R},\lambda} E_2 \in M\}$. 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 [18] and the standard ordering on natural numbers. The **RFC** and **Dec** rules decrease the first component of the measure. (Note that for **Dec** the decrease 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 $\perp$ is not transformed further. Hence, the algorithm terminates.

Since for each possible shape of a proximity matching problem there is a corresponding rule, the process stops either with $\perp$ 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 $\perp$. □

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 $\{V_1 \approx E_1, \dots, V_n \approx E_n\}$ iff $V_i\sigma = E_i$ for each $1 \leqslant i \leqslant n$. A solution of a pair $M; S$, where M is a proximity matching problem and S a set of equations in pre-solved form, is a substitution σ that solves both M and S. The configuration $\perp$ has no solutions.

Theorem 6.4.2 (Soundness)**.** *Let M be a proximity matching 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 $M_1; S_1; \alpha_1 \Longrightarrow_{\mathsf{R}} M_2; S_2; \alpha_2$ 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 $\mathcal{R}(f, g) = \beta \geqslant \lambda$. Obviously, if $\mathcal{R}(\tilde{s}\sigma, \tilde{t}) \geqslant \alpha_1 \wedge \beta$, then $\mathcal{R}(f(\tilde{s})\sigma, g(\tilde{t})) \geqslant \alpha_1 \wedge \beta$. 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 $\mathcal{R}(g', g) = \beta$. Besides, $g' = X\sigma$. Therefore, if $\mathcal{R}(\tilde{s}\sigma, \tilde{t}) \geqslant \alpha_1 \wedge \beta$, then $\mathcal{R}(X(\tilde{s})\sigma, g(\tilde{t})) \geqslant \alpha_1 \wedge \beta$, 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 have just established. □

Lemma 6.4.1. *If $M; S; \alpha \Longrightarrow \bot$, then $M; S$ has no solution.*

Proof. Assume M is an $(\mathcal{R}, \lambda)$-matching problem and analyze the rules that lead to $\bot$. For the **Cla1** rule, M is unsolvable, because $\mathcal{R}((f\tilde{s}))\sigma, g(\tilde{t})) = \mathcal{R}(f(\tilde{s}\sigma), g(\tilde{t})) = \mathcal{R}(f, g) \wedge \mathcal{R}(\tilde{s}\sigma, \tilde{t}) \leqslant \mathcal{R}(f, g) < \lambda$. In **Cla2** and **Cla3** rules, unsolvability of M follows from the fact that a nonempty hedge cannot 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 6.4.3 (Completeness)**.** *Let M be a proximity matching problem and σ be its solution with the proximity degree α. Then there exists a derivation in $\mathfrak{M}_\rho$ ending with a configuration $M; \varnothing; 1 \Longrightarrow^* \varnothing; S; \alpha$, 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. Applying **RFS** and **Dec** steps will not lower the proximity degree under α, because σ is a solution. No clashing and inconsistency step will be performed, because by Lemma 6.4.1 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 6.4.1. We use the proximity relation and problem from Example 6.3.2. The relation $\mathcal{R}$ is

$$\begin{aligned}
&\mathcal{R}(g_1, h_1) = \mathcal{R}(g_2, h_1) = 0.4\\
&\mathcal{R}(g_1, h_2) = \mathcal{R}(g_2, h_2) = 0.5\\
&\mathcal{R}(g_2, h_3) = \mathcal{R}(g_3, h_3) = 0.6\\
&\mathcal{R}(a, b) = 0.7
\end{aligned}$$

The proximity matching problem is

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

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

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

6.5 Conclusion

We extended the ρLog calculus with the capabilities of working 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.

Chapter 7

Multiple Similarity Relations

7.1 Introduction

While the previous chapters were all about symbolic techniques for proximity relations, here we move the focus onto similarity. Reasoning with similarity relations requires solving similarity-based constraints, which is the central computational mechanism for such inferences. Several approaches to unification modulo similarity have been proposed, see, e.g., [40, 59, 28, 1, 2, 29, 69, 27, 78, 79, 34, 26]. The techniques studied in these papers usually assume a single fuzzy similarity relation. However, in many practical situations, one needs to deal with several similarities between the objects from the same set, see, e.g. [76, 77], where examples about building online fashion compatibility representation and understanding visual similarities are considered in the context of learning image embeddings.

Multiple similarities pose challenges to constraint solving, since we can not rely on the transitivity property anymore. Note that proximity relations are not transitive either, but their unification methods have some limitations in dealing with multiple similarities simultaneously.

In this chapter, we address this problem, proposing an algorithm for solving constraints over multiple similarity relations. A simple example below illustrates the problem together with the results of different approaches, and motivates the development of a dedicated technique for it.

Example 7.1.1. Let *white-circle*, *white-ellipse*, *gray-circle* and *gray-ellipse* be four symbols and $\mathcal{R}_1$ and $\mathcal{R}_2$ be two similarity relations, where $\mathcal{R}_1$ stands for "similar color, same shape" and $\mathcal{R}_2$ denotes "same color, similar shape". They are defined as

- $\mathcal{R}_1(\textit{white-circle}, \textit{gray-circle}) = \mathcal{R}_1(\textit{white-ellipse}, \textit{gray-ellipse}) = 0.5$,
- $\mathcal{R}_2(\textit{white-circle}, \textit{white-ellipse}) = \mathcal{R}_2(\textit{gray-circle}, \textit{gray-ellipse}) = 0.7$.

Assume we want to find an object X such that from the color point of view, it is at least 0.4-similar to *white-circle* and from the shape point of view, it is at least 0.5-similar to *gray-ellipse*. The corresponding constraint is $X \simeq_{\mathcal{R}_1,0.4}$ *white-circle* and $X \simeq_{\mathcal{R}_2,0.5}$ *gray-ellipse*. The expected answer is $X =$ *gray-circle*. But it is problematic to compute it by the existing fuzzy unification techniques.

The direct approach, trying to solve each equation separately by the weak unification algorithm from [69] leads to no solution in this case, because *white-circle* and *gray-ellipse* are not similar to each other by any of the given relations.

An alternative way could be to consider the constraint over the relation $\mathcal{R}_1 \cup \mathcal{R}_2$, which is a proximity, not a similarity, since transitivity is not satisfied. However, the proximity unification algorithm from [33] gives no solution. We can try to use the algorithm for solving proximity constraints from Section 4.2, but it would give two answers instead of one: $X =$ *gray-circle* and $X =$ *white-ellipse*. On the other hand, the algorithm proposed in this chapter computes the right solution $X =$ *gray-circle*. Its similarity degrees are 0.5 for the relation $\mathcal{R}_1$ and 0.7 for $\mathcal{R}_2$.

It should be mentioned that there exists a so called multi-adjoint framework [40, 59] that is flexible enough to accommodate multiple similarities. It is a logic programming-based approach, where one needs to extend programs by fuzzy similarity axioms for each alphabet symbol and use classical unification. The authors show how to encode Sessa's algorithm [69] in this framework.

We take a different approach. We develop the solving algorithm directly, without being dependent on the implementation or application preferences. It can be incorporated in a modular way in the constraint logic programming schema, can be used for constrained rewriting, querying, or similar purposes. It combines three parts: solving syntactic equations, solving similarity problems for one relation, and solving mixed problems. We permit not only variables for terms, but also variables for function symbols, since they are necessary in the process of finding an "intermediate object" between terms in different similarity relations.

Before going into the details of our algorithm we will digress by looking at how Sessa's algorithm [69] for a single similarity relation works. It is an elegant generalization of the syntactic unification algorithm [57, 67] to similarity relations.

7.1.1 Related work: Sessa's weak unification

Sessa's weak unification algorithm [69] originally contained five rules, putting variable elimination and occurrence check into one. We bring here a slight reformulation of the algorithm that fits better to the framework we follow in this thesis. The rules work on triples $P; \sigma; \alpha$, where P is the similarity unification problems to be solved, σ is the substitution computed so far, and α is the proximity degree, also computed so far. Remember that terms in this algorithm are the standard first-order terms, with the similarity relation extended to them in the usual way.

We give the rules the names, prefixed with weak-: tri for trivial, dec for decomposition, ori for orient, elim for variable elimination, cla for clash, and occ for occurrence check.

weak-tri: $\{x \simeq^?_{\mathcal{R},\lambda} x\} \uplus P; \sigma; \alpha \Longrightarrow P; \sigma; \alpha.$

weak-dec: $\{f(t_1, \dots, t_n) \simeq^?_{\mathcal{R},\lambda} g(s_1, \dots, s_n)\} \uplus P; \sigma; \alpha \Longrightarrow$
$\{t_1 \simeq^?_{\mathcal{R},\lambda} s_1, \dots, t_n \simeq^?_{\mathcal{R},\lambda} s_n\} \cup P; \sigma; \min(\alpha, \beta),$ [1]
where $n > 0$ and $\mathcal{R}(f, g) = \beta \geqslant \lambda$.

weak-ori: $\{t \simeq^?_{\mathcal{R},\lambda} x\} \uplus P; \sigma; \alpha \Longrightarrow \{x \simeq^?_{\mathcal{R},\lambda} t\} \uplus P; \sigma; \alpha,$
where t is not a variable.

weak-elim: $\{x \simeq^?_{\mathcal{R},\lambda} t\} \uplus P; \sigma; \alpha \Longrightarrow P\{x \mapsto t\}; \sigma\{x \mapsto t\}; \alpha,$ if $x \notin \mathcal{V}(t)$.

weak-cla: $\{f(t_1, \dots, t_n) \simeq^?_{\mathcal{R},\lambda} g(s_1, \dots, s_m)\} \uplus P; \sigma; \alpha \Longrightarrow \mathsf{false},$ [2]
if $\mathcal{R}(f, g) < \lambda$.

weak-occ: $\{x \simeq^?_{\mathcal{R},\lambda} t\} \uplus P; \sigma; \alpha \Longrightarrow \mathsf{false},$ if $x \in \mathcal{V}(t)$ and $x \neq t$.

The main difference between this formulation and Sessa's original algorithm is that we consider the λ-cut and proceed with decomposition if the similarity between the functions symbols is not smaller than λ (the condition $\mathcal{R}(f, g) \geqslant \lambda$ in weak-dec), while in the original algorithm, the decomposition is performed when $\mathcal{R}(f, g) > 0$. Similarly, in weak-cla we have $\mathcal{R}(f, g) < \lambda$, while in Sessa's paper it is $\mathcal{R}(f, g) = 0$.

[1] In general, one would need a T-norm here, but since we consider only the minimum T-norm in this thesis, we wrote min in the rule explicitly.

[2] Note that here function symbols with different arities have the proximity degree 0. That means that in the weak-cla rule, if $m \neq n$, then $\mathcal{R}(f, g) = 0$.

To solve a weak unification problem between t and s with respect to a similarity relation $\mathcal{R}$ and a given λ, we create the initial triple $\{t \simeq_{\mathcal{R},\lambda} s\}; Id; 1$ and apply the rules as long as possible. When the process ends with $\emptyset; \sigma; \alpha$, then σ is a weak unifier of t and s with the similarity degree $\alpha \geqslant \lambda$. If false is reached, then t and s are not weakly unifiable.

Example 7.1.2. From [69, p.414], slightly reformulated. Consider a similarity $\mathcal{R}$ such that $\mathcal{R}(p,r) = 0.5$, $\mathcal{R}(f,g) = 0.7$, $\mathcal{R}(a,c) = 0.3$ and perform the steps of the algorithm for $p(f(x),a,y)$ and $r(g(b),c,f(x))$ to be unified. Let $\lambda = 0.2$.

$$\begin{aligned}
&\{p(f(x),a,y) \simeq^?_{\mathcal{R},\lambda} r(g(b),c,f(x))\}; Id; 1 \Longrightarrow_{\text{weak-dec}}\\
&\{f(x) \simeq^?_{\mathcal{R},\lambda} g(b),\ a \simeq^?_{\mathcal{R},\lambda} c,\ y \simeq^?_{\mathcal{R},\lambda} f(x)\}; Id; 0.5 \Longrightarrow_{\text{weak-dec}}\\
&\{x \simeq^?_{\mathcal{R},\lambda} b,\ a \simeq^?_{\mathcal{R},\lambda} c,\ y \simeq^?_{\mathcal{R},\lambda} f(x)\}; Id; 0.5 \Longrightarrow_{\text{weak-elim}}\\
&\{a \simeq^?_{\mathcal{R},\lambda} c,\ y \simeq^?_{\mathcal{R},\lambda} f(b)\}; \{x \mapsto b\}; 0.5 \Longrightarrow_{\text{weak-dec}}\\
&\{y \simeq^?_{\mathcal{R},\lambda} f(b)\}; \{x \mapsto b\}; 0.3 \Longrightarrow_{\text{weak-elim}}\\
&\emptyset; \{x \mapsto b, y \mapsto f(b)\}; 0.3.
\end{aligned}$$

The computed substitution $\{x \mapsto b, y \mapsto f(b)\}$ is a weak most general unifier of $p(f(x),a,y)$ and $r(g(b),c,f(x))$ with the degree 0.3.

7.2 Notions and terminology

Before presenting our algorithm we define the additional notions needed in this chapter. They are direct adaptations of the corresponding notions from the previous chapters.

Terms, atoms, substitutions.

Our alphabet $\mathcal{A}$ consists of the following pairwise disjoint sets of symbols:

- $\mathcal{V}_{\mathsf{T}}$: term variables, denoted by x, y, z,
- $\mathcal{V}_{\mathsf{F}}$: function variables, denoted by F, G, H,
- $\mathcal{F}$: function constants, denoted by f, g, h.

By $\mathbf{V}$ we denote the set of variables $\mathbf{V} = \mathcal{V}_{\mathsf{T}} \cup \mathcal{V}_{\mathsf{F}}$, and V is used for its elements.

A *function symbol* is a function variable or a function constant, i.e., an element of the set $\mathbf{F} = \mathcal{F} \cup \mathcal{V}_{\mathsf{F}}$. We use the letters $\mathsf{f}, \mathsf{g}, \mathsf{h}$ to denote function symbols. Each function symbol has a fixed arity.

Terms over $\mathcal{A}$ are defined by the grammar $t := x \mid \mathsf{f}(t_1, \dots, t_n)$, where f is an n-ary function symbol. For terms we use the letters t, s, r. The set of terms over $\mathcal{A}$ is denoted by $\mathcal{T}(\mathcal{A})$.

For a term $\mathsf{f}(t_1, \dots, t_n)$, if $n = 0$, we write just f instead of $\mathsf{f}()$. Usually, from the context it is clear whether we are talking about a symbol or about a term.

Some of the standard notions related to substitution need to be adjusted in this chapter, to apply to our alphabet (see below), while others, like domain, range, restriction, etc., remain unchanged.

A *substitution* σ is a mapping from $\mathbf{V}$ to $\mathbf{F} \cup \mathcal{T}(\mathcal{A})$ such that

- $\sigma(x) \in \mathcal{T}(\mathcal{A})$ for all $x \in \mathcal{V}_{\mathsf{T}}$,
- $\sigma(F) \in \mathbf{F}$ for all $F \in \mathcal{V}_{\mathsf{F}}$,
- $\sigma(V) = V$ for all but finitely many variables $V \in \mathbf{V}$.

Substitution application to variables, constants, and terms is defined as follows: $f\sigma = f$ for all $f \in \mathcal{F}$, $V\sigma = \sigma(V)$ for all $V \in \mathbf{V}$, and $\mathsf{f}(t_1, \dots, t_n)\sigma = (\mathsf{f}\sigma)(t_1\sigma, \dots, t_n\sigma)$.

Similarity relations on syntactic domains

Our similarity relations are defined on the set of constants $\mathcal{F}$. Any such relation $\mathcal{R}$ should satisfy the restriction: $\mathcal{R}(f, g) = 0$, if f and g have different arity.

Given an $\mathcal{R}$ defined on $\mathcal{F}$, we extend it to $\mathbf{F} \cup \mathcal{T}(\mathcal{A})$:

- For variables: $\mathcal{R}(V, V) = 1$.
- For nonvariable terms: $\mathcal{R}(\mathsf{f}(t_1, \dots, t_n), \mathsf{g}(s_1, \dots, s_n)) = \min(\mathcal{R}(\mathsf{f}, \mathsf{g}), \mathcal{R}(t_1, s_1), \dots, \mathcal{R}(t_n, s_n))$, when f and g are both n-ary.
- In all other cases, $\mathcal{R}(\mathsf{t}_1, \mathsf{t}_2) = 0$ for $\mathsf{t}_1, \mathsf{t}_2 \in \mathbf{F} \cup \mathcal{T}(\mathcal{A})$.

Given a similarity relation $\mathcal{R}$ and the cut value $\lambda \in (0,1]$, we extend the definition of the $(\mathcal{R},\lambda)$-*neighborhood* of t as $\mathbf{nb}(\mathsf{t},\mathcal{R},\lambda) := \{\mathsf{t}' \mid \mathcal{R}(\mathsf{t},\mathsf{t}') \geqslant \lambda\}$, where $\mathsf{t},\mathsf{t}' \in \mathbf{F} \cup \mathcal{T}(\mathcal{A})$. Based on the definition of similarity relations above, it is obvious that neighborhoods of function constants (resp. variables) contain only function constants (resp. variables) of the same arity. Neighborhoods of terms contain only terms. All terms in the same neighborhood have the same structure (same set of positions). We require for each $f \in \mathcal{F}$, $\mathcal{R}$, and λ, the set $\mathbf{nb}(f,\mathcal{R},\lambda)$ to be finite. It implies that term neighborhoods are finite as well.

Constraints

In our constraint language, the elements of $\mathbf{F} \cup \mathcal{T}(\mathcal{A})$ are the basic objects. In the rest of the chapter, the letter t is used to denote its elements. Besides, we have the equality predicate constant $\doteq$ (interpreted as syntactic equality), one or more similarity predicate constants $\simeq_1$, $\simeq_2$, …, (interpreted as similarity relations on $\mathbf{F} \cup \mathcal{T}(\mathcal{A})$), propositional constants true and false, connectives $\wedge$, $\vee$, and the quantifier $\exists$.

Primitive constraints $\mathcal{P}$ are defined by the grammar

$$\mathcal{P} ::= \ \mathsf{true} \mid \mathsf{false} \mid t \doteq s \mid t \simeq s \mid \mathsf{f} \doteq \mathsf{g} \mid \mathsf{f} \simeq \mathsf{g},$$

where $\simeq \, \in \{\simeq_1, \simeq_2 \ldots\}$. Primitive $\doteq$- and $\simeq$-constraints are called *primitive equality constraints* and *primitive similarity constraints*, respectively. A *literal* L is an atom or a primitive constraint. A (positive) *constraint* $\mathcal{C}$ over $\mathcal{A}$ is defined as $\mathcal{C} ::= \mathcal{P} \mid \mathcal{C} \wedge \mathcal{C} \mid \mathcal{C} \vee \mathcal{C} \mid \exists x.\mathcal{C}$. In this chapter we consider only positive constraints.

The domain of the *intended interpretation* of our constraint language is its Herbrand universe (the set of ground terms). The predicate constant $\doteq$ is interpreted as syntactic equality. Each similarity predicate constant $\simeq$ is interpreted as a similarity relation on the domain as defined in the previous section. When a predicate constant $\simeq$ is to be interpreted by a relation $\mathcal{R}$ with the cut value $\lambda \in (0,1]$, we write $\simeq_{\mathcal{R},\lambda}$ instead of $\simeq$.

A *variable-predicate pair (VP-pair)* is either $\langle V, \simeq_{\mathcal{R},\lambda}\rangle$ or $\langle V, \doteq\rangle$. We say that a substitution σ is *more general* than ϑ on a set of VP-pairs $\mathcal{W}$ iff there exists a substitution φ such that $\mathcal{R}(V\sigma\varphi, V\vartheta) \geqslant \lambda$ for all $\langle V, \simeq_{\mathcal{R},\lambda}\rangle \in \mathcal{W}$ and $V\sigma\varphi = V\vartheta$ for all $\langle V, \doteq\rangle \in \mathcal{W}$. In this case we write $\sigma \preceq_{\mathcal{W}} \vartheta$.

Example 7.2.1. Let $\mathcal{R}_1(a,b) = 0.7$, $\mathcal{R}_1(b,c) = 0.7$, $\mathcal{R}_1(a,c) = 0.8$, $\mathcal{R}_2(b,c) = 0.9$, and $\mathcal{W} = \{\langle x, \simeq_{\mathcal{R}_1,0.5}\rangle, \langle y, \simeq_{\mathcal{R}_1,0.6}\rangle, \langle y, \simeq_{\mathcal{R}_2,0.7}\rangle\}$.

- Let $\sigma = \{x \mapsto y\}$ and $\vartheta = \{x \mapsto a, y \mapsto b\}$. Then $\sigma \preceq_{\mathcal{W}} \vartheta$, because for $\varphi = \{x \mapsto b, y \mapsto b\}$ we have $x\sigma\varphi = b \simeq_{\mathcal{R}_1,0.5} a = x\vartheta$, $y\sigma\varphi = b \simeq_{\mathcal{R}_1,0.6} b = y\vartheta$, and $y\sigma\varphi = b \simeq_{\mathcal{R}_2,0.7} b = y\vartheta$.

- Let $\sigma = \{x \mapsto y\}$ and $\vartheta = \{x \mapsto a, y \mapsto c\}$. Then $\sigma \preceq_{\mathcal{W}} \vartheta$, because for $\varphi = \{x \mapsto b, y \mapsto b\}$ we have $x\sigma\varphi = b \simeq_{\mathcal{R}_1,0.5} a = x\vartheta$, $y\sigma\varphi = b \simeq_{\mathcal{R}_1,0.6} c = y\vartheta$, and $y\sigma\varphi = b \simeq_{\mathcal{R}_2,0.7} c = y\vartheta$.

- Let $\sigma = \{x \mapsto f(y), y \mapsto z\}$ and $\vartheta = \{x \mapsto f(z), y \mapsto a, z \mapsto x\}$. Then $\sigma \preceq_{\mathcal{W}} \vartheta$, because for $\varphi = \{y \mapsto z, z \mapsto a\}$ we have $x\sigma\varphi = f(z) \simeq_{\mathcal{R}_1,0.5} f(z) = x\vartheta$, $y\sigma\varphi = a \simeq_{\mathcal{R}_1,0.6} a = y\vartheta$, and $y\sigma\varphi = a \simeq_{\mathcal{R}_2,0.7} a = y\vartheta$.

Theorem 7.2.1. *$\preceq_{\mathcal{W}}$ is a quasi-ordering for all $\mathcal{W}$.*

Proof. Reflexivity is obvious. For transitivity, assume $\sigma_1 \preceq_{\mathcal{W}} \sigma_2$ and $\sigma_2 \preceq_{\mathcal{W}} \sigma_3$. We will show $\sigma_1 \preceq_{\mathcal{W}} \sigma_3$. Take $\langle V, \simeq_{\mathcal{R},\lambda}\rangle \in \mathcal{W}$. Then for some φ_1 and φ_2 we have $\mathcal{R}(V\sigma_1\varphi_1, V\sigma_2) \geqslant \lambda$ and $\mathcal{R}(V\sigma_2\varphi_2, V\sigma_3) \geqslant \lambda$. Since similarity is stable for substitutions [69, Proposition 3.1], we have $\mathcal{R}(V\sigma_1\varphi_1\varphi_2, V\sigma_2\varphi_2) \geqslant \lambda$. By transitivity of similarity, we get $\mathcal{R}(V\sigma_1\varphi_1\varphi_2, V\sigma_3) \geqslant \min(\mathcal{R}(V\sigma_1\varphi_1\varphi_2, V\sigma_2\varphi_2), \mathcal{R}(V\sigma_2\varphi_2, V\sigma_3)) \geqslant \lambda$, which implies that $\sigma_1 \preceq_{\mathcal{W}} \sigma_3$. □

We denote the equivalence relation induced by $\preceq_{\mathcal{W}}$ by $\cong$.

The notation $\mathcal{K}_{\doteq}$ denotes a conjunction of primitive equality constraints. By $\mathcal{K}_{\mathcal{R},\lambda}$ we denote a conjunction of primitive similarity constraints, all with the same relation $\mathcal{R}$ and the same λ-cut: $\mathcal{K}_{\mathcal{R},\lambda} = \mathsf{t}_1 \simeq_{\mathcal{R},\lambda} \mathsf{t}'_1 \wedge \cdots \wedge \mathsf{t}_n \simeq_{\mathcal{R},\lambda} \mathsf{t}'_n$.

Given a constraint $\mathcal{K} = \mathcal{K}_{\doteq} \wedge \mathcal{K}_{\mathcal{R}_1,\lambda_1} \wedge \cdots \wedge \mathcal{K}_{\mathcal{R}_m,\lambda_m}$, we denote by $\mathcal{W}(\mathcal{K})$ the set of VP-pairs $\mathcal{W}(\mathcal{K}) := \{\langle V, \doteq\rangle \mid V \in \mathcal{V}(\mathcal{K}_{\doteq})\} \cup \{\langle V, \simeq_{\mathcal{R}_i,\lambda_i}\rangle \mid V \in \mathcal{V}(\mathcal{K}_{\mathcal{R}_i,\lambda_i}), 1 \leqslant i \leqslant m\}$.

Definition 7.2.1 (Solution). *A substitution σ is called a* solution *of a primitive constraint $\mathcal{P}$, if*

- *$\mathcal{P} = \mathsf{t}_1 \doteq \mathsf{t}_2$ and $\mathsf{t}_1\sigma = \mathsf{t}_2\sigma$, or*

- *$\mathcal{P} = \mathsf{t}_1 \simeq_{\mathcal{R},\lambda} \mathsf{t}_2$ and $\mathcal{R}(\mathsf{t}_1\sigma, \mathsf{t}_2\sigma) \geqslant \lambda$.*

Any substitution is a solution of true*, while* false *has no solution.*

A substitution σ is a solution of a conjunction of primitive constraints $\mathcal{K}$ *iff it solves each primitive constraint in $\mathcal{K}$. We denote the* set of all solutions *of $\mathcal{K}$ by $Sol(\mathcal{K})$. For a constraint $\mathcal{C} = \mathcal{K}_1 \vee \cdots \vee \mathcal{K}_n$ in disjunctive normal form (DNF), we define $Sol(\mathcal{C}) = \cup_{i=1}^n Sol(\mathcal{K}_i)$.*

Given similarity relations $\mathcal{R}_1, \dots, \mathcal{R}_n$, a conjunction of primitive constraints $\mathcal{K}$, and its solution σ, we say that σ solves $\mathcal{K}$ with approximation degrees $\mathfrak{D} = \{\langle \mathcal{R}_1, \mathfrak{d}_1 \rangle, \dots, \langle \mathcal{R}_n, \mathfrak{d}_n \rangle\}$ *if*

- $\mathcal{K} = \mathsf{true}$ *or* $\mathcal{K} = \mathcal{K}_{\doteq}$ *and* $\mathfrak{d}_1 = \cdots = \mathfrak{d}_n = 1$,
- $\mathcal{K} = \mathsf{t}_1 \simeq_{\mathcal{R}_j, \lambda_j} \mathsf{t}_2$ *for some* $1 \leqslant j \leqslant n$, $\mathfrak{d}_j = \mathcal{R}_j(\mathsf{t}_1\sigma, \mathsf{t}_2\sigma) \geqslant \lambda_j$, *and* $\mathfrak{d}_i = 1$ *for all* $1 \leqslant i \leqslant n$, $i \neq j$,
- $\mathcal{K} = \mathcal{P} \wedge \mathcal{K}'$ *for a primitive constraint* $\mathcal{P}$, σ *solves* $\mathcal{P}$ *and* $\mathcal{K}'$ *with approximation degrees* $\{\langle \mathcal{R}_1, \mathfrak{d}_1^{\mathcal{P}} \rangle, \dots, \langle \mathcal{R}_n, \mathfrak{d}_n^{\mathcal{P}} \rangle\}$ *and* $\{\langle \mathcal{R}_1, \mathfrak{d}'_1 \rangle, \dots, \langle \mathcal{R}_n, \mathfrak{d}'_n \rangle\}$, *respectively, and* $\mathfrak{d}_i = \min\{\mathfrak{d}_i^{\mathcal{P}}, \mathfrak{d}'_i\}$ *for all* $1 \leqslant i \leqslant n$.

Such a definition of approximation degrees gives the flexibility to characterize approximations with respect to each involved relation independently from each other.

Theorem 7.2.2. *Let* $\mathcal{K} = \mathcal{K}_{\doteq} \wedge \mathcal{K}_{\mathcal{R}_1, \lambda_1} \wedge \cdots \wedge \mathcal{K}_{\mathcal{R}_m, \lambda_m}$ *be a constraint. If* σ *is a solution of* $\mathcal{K}$ *and* $\sigma \preceq_{\mathcal{W}(\mathcal{K})} \vartheta$, *then* ϑ *is a solution of* $\mathcal{K}$.

Proof. Let $s_1 \simeq_{\mathcal{R}_i, \lambda_i} s_2 \in \mathcal{K}_{\mathcal{R}_i, \lambda_i}$. From $\sigma \preceq_{\mathcal{W}(\mathcal{K})} \vartheta$, by definition of $\preceq_{\mathcal{W}(\mathcal{K})}$, there exists a φ such that $\mathcal{R}(V\sigma\varphi, V\vartheta) \geqslant \lambda_i$ for each $V \in \mathcal{V}(\mathcal{K}_{\mathcal{R}_i, \lambda_i})$. It implies that

$$\mathcal{R}(s_j\sigma\varphi, s_j\vartheta) \geqslant \lambda_i, \quad j = 1, 2. \tag{7.1}$$

On the other hand, for similarity relations $\mathcal{R}(s_1\sigma\varphi, s_2\sigma\varphi) = \mathcal{R}(s_1\sigma, s_2\sigma)$ (see [69]). Since σ is a solution of $\mathcal{K}$, $\mathcal{R}(s_1\sigma, s_2\sigma) \geqslant \lambda_i$. Hence, we have $\mathcal{R}(s_1\sigma\varphi, s_2\sigma\varphi) \geqslant \lambda_i$. From this inequality and (7.1), by symmetry and transitivity of $\mathcal{R}$, we get $\mathcal{R}(s_1\vartheta, s_2\vartheta) \geqslant \lambda_i$. Hence, ϑ is a solution of $s_1 \simeq_{\mathcal{R}_i, \lambda_i} s_2$.

It is straightforward that ϑ is a solution of any equation from $\mathcal{K}_{\doteq}$. Hence, ϑ is a solution of $\mathcal{K}$. □

Definition 7.2.2 (Solved form, approximately solved form)**.** *A conjunction of primitive constraints* $\mathcal{K}$ *is in* solved form, *if* $\mathcal{K}$ *is either* true *or each primitive constraint in* $\mathcal{K}$ *has a form* $V \doteq \mathsf{t}$ *or* $V \simeq_{\mathcal{R}, \lambda} \mathsf{t}$, *where* V *appears only once in* $\mathcal{K}$. *A constraint in DNF* $\mathcal{K}_1 \vee \cdots \vee \mathcal{K}_n$ *is in solved form, if each* $\mathcal{K}_i$ *is in solved form.*

A conjunction of primitive constraints $\mathcal{K}_{\mathrm{sol}} \wedge \mathcal{K}_{\mathrm{var}}$ *is in* approximately solved form (appr-solved form) *if* $\mathcal{K}_{\mathrm{sol}}$ *is in solved form, and* $\mathcal{K}_{\mathrm{var}}$ *is a conjunction of primitive similarity constraints between variables* $V_1 \simeq_{\mathcal{R}, \lambda} V_2$ *such* *that neither V_1 nor V_2 appear in the left hand side of any primitive constraint in $\mathcal{K}_{\mathrm{sol}}$. A constraint in DNF $\mathcal{K}_1 \vee \cdots \vee \mathcal{K}_n$ is in appr-solved form, if each $\mathcal{K}_i$ is in appr-solved form.*

Solved forms are also appr-solved forms, but not vice versa. Each solved form $\mathcal{K}$ *induces* a substitution, denoted by $\sigma_{\mathcal{K}}$: if $\mathcal{K} = \mathsf{true}$, then $\sigma_{\mathcal{K}} = Id$, otherwise $\sigma_{\mathcal{K}} = \{V \mapsto \mathsf{t} \mid V \doteq \mathsf{t} \in \mathcal{K} \text{ or } V \simeq_{\mathcal{R},\lambda} \mathsf{t} \in \mathcal{K}\}$. Obviously, $\sigma_{\mathcal{K}}$ is a solution of $\mathcal{K}$. A constraint $\mathcal{K}_{\mathrm{sol}} \wedge \mathcal{K}_{\mathrm{var}}$ in appr-solved form is also solvable, because $\sigma_{\mathcal{K}_{\mathrm{sol}}}$ solves $\mathcal{K}_{\mathrm{sol}}$ and $\mathcal{K}_{\mathrm{var}}$ always has at least a trivial solution mapping all terms variables to the same term variable and all function variables to the same function variable.

Example 7.2.2. Let $\mathcal{R}_1$ and $\mathcal{R}_2$ be defined as in Example 7.1.1 and $\mathcal{K} = \mathcal{K}_{\mathcal{R}_1,0.4} \wedge \mathcal{K}_{\mathcal{R}_2,0.5}$ be a constraint, where $\mathcal{K}_{\mathcal{R}_1,0.4} = x \simeq_{\mathcal{R}_1,0.4} \textit{white-circle} \wedge x \simeq_{\mathcal{R}_1,0.4} y$ and $\mathcal{K}_{\mathcal{R}_2,0.5} = x \simeq_{\mathcal{R}_2,0.5} \textit{gray-ellipse} \wedge y \simeq_{\mathcal{R}_2,0.5} \textit{white-ellipse}$.

One can bring $\mathcal{K}_{\mathcal{R}_1,0.4}$ to its equivalent solved form (e.g., by an algorithm along the lines of the weak unification algorithm in [69]). $\mathcal{K}_{\mathcal{R}_2,0.5}$ is already in the solved form. Hence, $\mathcal{K}$ is equivalent to the constraint

$$x \simeq_{\mathcal{R}_1,0.4} \textit{white-circle} \wedge y \simeq_{\mathcal{R}_1,0.4} \textit{white-circle} \wedge$$
$$x \simeq_{\mathcal{R}_2,0.5} \textit{gray-ellipse} \wedge y \simeq_{\mathcal{R}_2,0.5} \textit{white-ellipse},$$

which is not yet in a solved form. A solved form, equivalent to $\mathcal{K}$, would be $x \doteq \textit{gray-circle} \wedge y \doteq \textit{white-circle}$. It induces the substitution $\sigma = \{x \mapsto \textit{gray-circle}, y \mapsto \textit{white-circle}\}$.

Example 7.2.3. Let $\mathcal{R}_1$ and $\mathcal{R}_2$ be two similarity relations defined as

$$\begin{aligned}
\mathcal{R}_1 :\quad & \mathcal{R}_1(a_1, c_1) = \mathcal{R}_1(b_1, c_1) = 0.7, \quad \mathcal{R}_1(a_1, b_1) = 0.8,\\
& \mathcal{R}_1(a_2, c_2) = \mathcal{R}_1(b_2, c_2) = 0.6, \quad \mathcal{R}_1(a_2, b_2) = 0.7,\\
\mathcal{R}_2 :\quad & \mathcal{R}_2(b_1, b_3) = \mathcal{R}_2(b_2, b_3) = 0.6, \quad \mathcal{R}_2(b_1, b_2) = 0.7, \quad \mathcal{R}_2(c_1, c_2) = 0.8.
\end{aligned}$$

Visually:

$$\begin{tikzcd}
 & & b_3 \arrow[dl, dash, dotted, "0.6"'] \arrow[dr, dash, dotted, "0.6"] & & \\
 & b_1 \arrow[rr, dash, dotted, "0.7"] \arrow[dl, dash, "0.8"'] \arrow[d, dash, "0.7"] & & b_2 \arrow[d, dash, "0.6"'] \arrow[dr, dash, "0.7"] & \\
a_1 \arrow[r, dash, "0.7"] & c_1 \arrow[rr, dash, dotted, "0.8"] & & c_2 \arrow[r, dash, "0.6"] & a_2
\end{tikzcd}$$

$\mathcal{R}_1$: the solid lines, $\mathcal{R}_2$: the dotted lines.

Let $\mathcal{K} = x \simeq_{\mathcal{R}_1,0.5} f(a_1, a_2) \wedge x \simeq_{\mathcal{R}_2,0.6} f(y, y)$. It is equivalent to the disjunction of two solved forms, e.g., $(x \doteq f(b_1, b_2) \wedge y \simeq_{\mathcal{R}_2,0.6} b_1) \vee (x \doteq f(c_1, c_2) \wedge y \simeq_{\mathcal{R}_2,0.6} c_1)$. The solved forms induce two substitutions: $\sigma_1 = \{x \mapsto f(b_1, b_2),\, y \mapsto b_1\}$ and $\sigma_2 = \{x \mapsto f(c_1, c_2),\, y \mapsto c_1\}$. They are solutions of $\mathcal{K}$. There are other solutions of $\mathcal{K}$ that are $\cong$-equivalent to σ_1 or σ_2: $\vartheta_1 = \{x \mapsto f(b_1, b_2),\, y \mapsto b_2\} \cong \sigma_1$, $\vartheta_2 = \{x \mapsto f(b_1, b_2),\, y \mapsto b_3\} \cong \sigma_1$, and $\vartheta_3 = \{x \mapsto f(c_1, c_2),\, y \mapsto c_2\} \cong \sigma_2$.

Now let $\mathcal{K} = x \simeq_{\mathcal{R}_1,0.8} g(y) \wedge x \simeq_{\mathcal{R}_2,0.6} g(z)$. A solved form $\mathcal{K}_{\mathrm{s}} = x \doteq g(z) \wedge y \doteq z$ implies $\mathcal{K}$, but is not equivalent to it, because $\mathcal{K}$ has solutions $\{x \mapsto g(b_1), y \mapsto a_1, z \mapsto b_2\}$ and $\{x \mapsto g(b_1), y \mapsto a_1, z \mapsto b_3\}$, which do not solve $\mathcal{K}_{\mathrm{s}}$. On the other hand, if we take the approximate solved form $\mathcal{K}_{\mathrm{as}} = x \doteq g(x_1) \wedge x_1 \simeq_{\mathcal{R}_1,0.8} y \wedge x_1 \simeq_{\mathcal{R}_2,0.6} z$, then every solution of $\mathcal{K}_{\mathrm{as}}$ solves $\mathcal{K}$, and $(\exists x_1.\mathcal{K}_{\mathrm{as}})\sigma$ holds for any solution σ of $\mathcal{K}$. (Substitution application to a quantified constraint avoids variable capture.) Alternatively, we could have taken another solved form $\mathcal{K}'_{\mathrm{s}} = x \doteq g(x_1) \wedge y \simeq_{\mathcal{R}_1,0.8} x_1 \wedge z \simeq_{\mathcal{R}_2,0.6} x_1$ which has the same properties as $\mathcal{K}_{\mathrm{as}}$.

Example 7.2.4. Let $\mathcal{R}_1(a, b_1) = \mathcal{R}_1(b_1, b_2) = \mathcal{R}_1(a, b_2) = 0.8$, $\mathcal{R}_2(c, b_1) = \mathcal{R}_2(b_1, b_2) = \mathcal{R}_2(b_2, c) = 0.7$ and consider a constraint $\mathcal{K} = x \simeq_{\mathcal{R}_1,0.6} f(y, y) \wedge x \simeq_{\mathcal{R}_2,0.5} f(z, z)$. The straightforward solved form $x \doteq f(z, z) \wedge y \doteq z$, as in the previous example, has fewer solutions than $\mathcal{K}$, e.g., $\{x \mapsto f(b_1, b_2), y \mapsto a, z \mapsto c\}$ would be lost. If we take an appr-solved form $\mathcal{K}_{\mathrm{as}} = x \doteq f(x_1, x_2) \wedge x_1 \simeq_{\mathcal{R}_1,0.6} y \wedge x_1 \simeq_{\mathcal{R}_2,0.5} z \wedge x_2 \simeq_{\mathcal{R}_1,0.6} y \wedge x_2 \simeq_{\mathcal{R}_2,0.5} z$, then all solutions of $\mathcal{K}_{\mathrm{as}}$ solve $\mathcal{K}$ and for each solution σ of $\mathcal{K}$, we have $(\exists x_1, x_2.\mathcal{K}_{\mathrm{as}})\sigma$. Unlike the previous example, we can not turn this appr-solved form into a solved form by swapping sides of variables-only equations.

7.3 Constraint solving

The constraint solving algorithm Solve presented in this section works on constraints in DNF. Its rules are divided into three groups: for equalities, for similarities, and for mixed problems. They are applied modulo associativity, commutativity, and idempotence of $\wedge$ and $\vee$, treating false as the unit element of $\vee$. We first introduce the rules and then define Solve by using them.

In the rules, the superscript ? indicates that the constraints are supposed to be solved. The sides of an equation $V \doteq^? \mathsf{t}$ belong to the same syntactic category, i.e., it stands either for $x \doteq^? t$ or for $F \doteq^? \mathsf{f}$. The same holds for $V \simeq^?_{\mathcal{R},\lambda} \mathsf{t}$.

7.3.1 Equality rules

In this subsection we describe the rules that solve equality constraints. Essentially, these are first-order unification rules with a slight modification, which concerns dealing with function and predicate variables. The rules have the form $\mathcal{K} \rightsquigarrow \mathcal{K}'$, which defines the transformation $\mathcal{K} \vee \mathcal{C} \rightsquigarrow \mathcal{K}' \vee \mathcal{C}$. Note that $\mathcal{C}$ does not change.

The rules are Del-eq (deletion), Dec-eq (decomposition), Ori-eq (orientation), Elim-eq (variable elimination), Confl-eq (conflict), Mism-eq (arity mismatch), Occ-eq (occurrence check), all formulated for the equality relation $\doteq$.

$$\mathsf{Del\text{-}eq}: \quad \mathsf{t} \doteq^? \mathsf{t} \wedge \mathcal{K} \rightsquigarrow \mathcal{K}, \quad \text{where } \mathsf{t} \in \mathcal{F} \cup \mathcal{V}_\mathsf{F} \cup \mathcal{V}_\mathsf{T}.$$

$$\mathsf{Dec\text{-}eq}: \quad \mathsf{f}(t_1, \dots, t_n) \doteq^? \mathsf{g}(s_1, \dots, s_n) \wedge \mathcal{K} \rightsquigarrow \\ \mathsf{f} \doteq^? \mathsf{g} \wedge t_1 \doteq^? s_1 \wedge \cdots \wedge t_n \doteq^? s_n \wedge \mathcal{K}, \text{ where } n > 0.$$

$$\mathsf{Ori\text{-}eq}: \quad \mathsf{t} \doteq^? V \wedge \mathcal{K} \rightsquigarrow V \doteq^? \mathsf{t} \wedge \mathcal{K}, \quad \text{if } \mathsf{t} \notin \mathbf{V}.$$

$$\mathsf{Elim\text{-}eq}: \quad V \doteq^? \mathsf{t} \wedge \mathcal{K} \rightsquigarrow V \doteq^? \mathsf{t} \wedge \mathcal{K}\{V \mapsto \mathsf{t}\}, \quad \text{if } V \notin \mathcal{V}(\mathsf{t}) \text{ and } V \in \mathcal{V}(\mathcal{K}).$$

$$\mathsf{Confl\text{-}eq}: \quad f \doteq^? g \wedge \mathcal{K} \rightsquigarrow \mathsf{false}, \quad \text{if } f \neq g.$$

$$\mathsf{Mism\text{-}eq}: \quad \mathsf{f}(t_1, \dots, t_n) \doteq^? \mathsf{g}(s_1, \dots, s_m) \wedge \mathcal{K} \rightsquigarrow \mathsf{false}, \text{ if } n \neq m.$$

$$\mathsf{Occ\text{-}eq}: \quad x \doteq^? t \wedge \mathcal{K} \rightsquigarrow \mathsf{false}, \text{ if } x \in \mathcal{V}(t) \text{ and } x \neq t.$$

Note that the Elim-eq rule replaces occurrences of a variable in the whole $\mathcal{K}$, i.e., the variable gets replaced both in equality and similarity constraints.

We define the algorithm $\mathfrak{Unif}$, which applies the equality rules as long as possible. When there are more than one applicable rule, the algorithm may choose one arbitrarily.

Theorem 7.3.1. $\mathfrak{Unif}$ *is terminating.*

Proof. Similar to the proof of termination of the unification algorithm in [7]. □

Lemma 7.3.1 (Soundness lemma for $\mathfrak{Unif}$)**.** *If $\mathcal{K} \rightsquigarrow \mathcal{K}'$ is a step performed by a rule in $\mathfrak{Unif}$, then $Sol(\mathcal{K}) = Sol(\mathcal{K}')$.*

Proof. When $\mathcal{K}$ consists of equational constraints only, then so is $\mathcal{K}'$ and the lemma can be proved as the analogous property of the unification algorithm in [7]. If $\mathcal{K}$ contains similarity constraints as well, the only nontrivial case to consider is the Elim-eq rule. We will need the fact that for any σ and ϑ, $\vartheta\sigma \in Sol(\mathcal{K})$ iff $\sigma \in Sol(\mathcal{K}\vartheta)$ (which is straightforward to show).

Let $\mathcal{K} = \{V \doteq^? \mathsf{t}\} \wedge \mathcal{K}_0$ and $\vartheta = \{V \mapsto \mathsf{t}\}$. Then $\mathcal{K}' = \{V \doteq^? \mathsf{t}\} \wedge \mathcal{K}_0\vartheta$ and we have

$$\begin{aligned}
&\sigma \in Sol(\mathcal{K}) \text{ iff } \sigma \in Sol(\{V \doteq^? \mathsf{t}\} \wedge \mathcal{K}_0) \text{ iff}\\
&\qquad V\sigma = \mathsf{t}\sigma \wedge \sigma \in Sol(\mathcal{K}_0) \text{ iff (because } V\sigma = \mathsf{t}\sigma \text{ implies } \sigma = \vartheta\sigma)\\
&\qquad V\sigma = \mathsf{t}\sigma \wedge \vartheta\sigma \in Sol(\mathcal{K}_0) \text{ iff}\\
&\qquad V\sigma = \mathsf{t}\sigma \wedge \sigma \in Sol(\mathcal{K}_0\vartheta) \text{ iff}\\
&\qquad \sigma \in Sol(\{V \doteq^? \mathsf{t}\} \wedge \mathcal{K}_0\vartheta) \text{ iff}\\
&\sigma \in Sol(\mathcal{K}').
\end{aligned}$$

□

7.3.2 Similarity rules

The rules in this section are designed for similarity relations. They resemble weak unification rules [34, 69], with the difference that function variables and multiple similarity relations are permitted.

In the Elim-sim rule, the variable V is replaced by t only in the constraints for the same similarity relation. This is justified by the fact that although a λ-cut of each similarity relation is transitive, from $t \simeq_{\mathcal{R}_1,\lambda_1} s$, $s \simeq_{\mathcal{R}_2,\lambda_2} r$ we can not conclude anything about similarity between t and r.

The similarity rules have the same form as the equality rules: $\mathcal{K} \rightsquigarrow \mathcal{K}'$, which defines the transformation $\mathcal{K} \vee \mathcal{C} \rightsquigarrow \mathcal{K}' \vee \mathcal{C}$. The names are also similar to those for equalities, using sim instead of eq.

Del-sim : $\mathsf{t}_1 \simeq^?_{\mathcal{R},\lambda} \mathsf{t}_2 \wedge \mathcal{K} \rightsquigarrow \mathcal{K}$,
where $\mathsf{t}_1, \mathsf{t}_2 \in \mathcal{F} \cup \mathcal{V}_\mathsf{F} \cup \mathcal{V}_\mathsf{T}$ and $\mathcal{R}(\mathsf{t}_1, \mathsf{t}_2) \geqslant \lambda$.

Dec-sim : $\mathsf{f}(t_1, \ldots, t_n) \simeq^?_{\mathcal{R},\lambda} \mathsf{g}(s_1, \ldots, s_n) \wedge \mathcal{K} \rightsquigarrow$
$\mathsf{f} \simeq^?_{\mathcal{R},\lambda} \mathsf{g} \wedge t_1 \simeq^?_{\mathcal{R},\lambda} s_1 \wedge \cdots \wedge t_n \simeq^?_{\mathcal{R},\lambda} s_n \wedge \mathcal{K}$, where $n > 0$.

Ori-sim : $\mathsf{t} \simeq^?_{\mathcal{R},\lambda} V \wedge \mathcal{K} \rightsquigarrow V \simeq^?_{\mathcal{R},\lambda} \mathsf{t} \wedge \mathcal{K}$, where $\mathsf{t} \notin \mathbf{V}$.

Elim-sim : $V \simeq^?_{\mathcal{R},\lambda} \mathsf{t} \wedge \mathcal{K}_{\mathcal{R},\lambda} \wedge \mathcal{K} \rightsquigarrow V \simeq^?_{\mathcal{R},\lambda} \mathsf{t} \wedge \mathcal{K}_{\mathcal{R},\lambda}\{V \mapsto \mathsf{t}\} \wedge \mathcal{K}$ where $\mathcal{K}$ does not contain primitive $\simeq^?_{\mathcal{R},\lambda}$-constraints, $V \notin \mathcal{V}(\mathsf{t})$, and $V \in \mathcal{K}_{\mathcal{R},\lambda}$.

Confl-sim : $f \simeq^?_{\mathcal{R},\lambda} g \wedge \mathcal{K} \rightsquigarrow \mathsf{false}$, if $\mathcal{R}(f,g) < \lambda$.

Mism-sim : $\mathsf{f}(t_1,\ldots,t_n) \simeq^?_{\mathcal{R},\lambda} \mathsf{g}(s_1,\ldots,s_m) \wedge \mathcal{K} \rightsquigarrow \mathsf{false}$, if $n \neq m$.

Occ-sim : $x \simeq^?_{\mathcal{R},\lambda} t \wedge \mathcal{K} \rightsquigarrow \mathsf{false}$, if $x \in \mathcal{V}(t)$ and $x \neq t$.

The algorithm $\mathfrak{Sim}$ applies the similarity rules as long as possible. When there are more than one applicable rule, the algorithm may choose one non-deterministically.

Termination of $\mathfrak{Sim}$ can be proved as termination of $\mathfrak{Unif}$:

Theorem 7.3.2. $\mathfrak{Sim}$ *is terminating.*

Lemma 7.3.2 (Soundness lemma for $\mathfrak{Sim}$)**.** *If $\mathcal{K} \rightsquigarrow \mathcal{K}'$ is a step performed by a rule in $\mathfrak{Sim}$, then $Sol(\mathcal{K}) = Sol(\mathcal{K}')$.*

Proof. When we have only one similarity relation, soundness follows from soundness of weak unification algorithm [69]. For the extension to multiple similarity relations, the only nontrivial rule is Elim-sim. (For the others, $Sol(\mathcal{K}) = Sol(\mathcal{K}')$ holds directly.) It is important to notice that in this rule, $\{V \mapsto \mathsf{t}\}$ applies only to $\mathcal{K}_{\mathcal{R},\lambda}$. Then for $V \simeq^?_{\mathcal{R},\lambda} \mathsf{t} \wedge \mathcal{K}_{\mathcal{R},\lambda}$ we have $Sol(V \simeq^?_{\mathcal{R},\lambda} \mathsf{t} \wedge \mathcal{K}_{\mathcal{R},\lambda}) = Sol(V \simeq^?_{\mathcal{R},\lambda} \mathsf{t} \wedge \mathcal{K}_{\mathcal{R},\lambda}\{V \mapsto \mathsf{t}\})$. (It follows from soundness of weak unification algorithm [69], since the constraint is over a single similarity relation.) Constraints for all other relations remain unchanged. It implies that the solution sets for constraints in both sides of the Elim-sim rule are the same. □

Example 7.3.1. let $\mathcal{K}_{\mathcal{R}_1,0.4} = x \simeq_{\mathcal{R}_1,0.4}$ *white-circle* $\wedge\ x \simeq_{\mathcal{R}_1,0.4} y$ and $\mathcal{K}_{\mathcal{R}_2,0.5} = x \simeq_{\mathcal{R}_2,0.5}$ *gray-ellipse* $\wedge\ y \simeq_{\mathcal{R}_2,0.5}$ *white-ellipse* be the constraints from Example 7.2.2. The reduction mentioned in that example is modeled by performing the Elim-sim step and replacing x by *white-circle* in $\mathcal{K}_{\mathcal{R}_1,0.4}$:

$$\begin{aligned}
&x \simeq_{\mathcal{R}_1,0.4} \textit{white-circle} \wedge x \simeq_{\mathcal{R}_1,0.4} y \wedge \\
&\quad x \simeq_{\mathcal{R}_2,0.5} \textit{gray-ellipse} \wedge y \simeq_{\mathcal{R}_2,0.5} \textit{white-ellipse} \rightsquigarrow_{\mathsf{Elim\text{-}sim}} \\
&x \simeq_{\mathcal{R}_1,0.4} \textit{white-circle} \wedge \textit{white-circle} \simeq_{\mathcal{R}_1,0.4} y \wedge \\
&\quad x \simeq_{\mathcal{R}_2,0.5} \textit{gray-ellipse} \wedge y \simeq_{\mathcal{R}_2,0.5} \textit{white-ellipse}.
\end{aligned}$$

If Elim-sim permitted to replace x not only in $\mathcal{K}_{\mathcal{R}_1,0.4}$, but also in $\mathcal{K}_{\mathcal{R}_2,0.5}$, we would get

$$x \simeq_{\mathcal{R}_1,0.4} \textit{white-circle} \wedge \textit{white-circle} \simeq_{\mathcal{R}_1,0.4} y \wedge \\ \textit{white-circle} \simeq_{\mathcal{R}_2,0.5} \textit{gray-ellipse} \wedge y \simeq_{\mathcal{R}_2,0.5} \textit{white-ellipse},$$

but *white-circle* $\simeq_{\mathcal{R}_2,0.5}$ *gray-ellipse* is unsolvable. Hence, we would lose a solution.

Mixed rules

The rules in this section apply when there are at least two primitive constraints over different similarity relations. The notation $t[x]$ below means that the variable x occurs in the term t.

Definition 7.3.1 (Occurrence cycle)**.** *An* occurrence cycle *for a variable x_1 is called the conjunction of primitive constraints $x_1 \simeq^?_{\mathcal{R}_1,\lambda_1} t_1[x_2] \wedge x_2 \simeq^?_{\mathcal{R}_2,\lambda_2} t_2[x_3] \wedge \cdots \wedge x_n \simeq^?_{\mathcal{R}_n,\lambda_n} t_n[x_1]$, where $n > 1$, $\mathcal{R}_i \neq \mathcal{R}_{i+1}$ for all $1 \leqslant i \leqslant n-1$, $\mathcal{R}_n \neq \mathcal{R}_1$, and at least one t is not a variable.*

Remark 7.3.1. Note that in the definition of occurrence cycle, if two neighboring primitive similarity constraints use the same relation, they can be contracted into one constraint by transitivity, i.e., instead of $x_i \simeq^?_{\mathcal{R}_i,\lambda_i} t_i[x_{i+1}] \wedge x_{i+1} \simeq^?_{\mathcal{R}_i,\lambda_i} t_{i+1}[x_{i+2}]$ we can have $x_i \simeq^?_{\mathcal{R}_i,\lambda_i} t_i[t_{i+1}[x_{i+2}]]$, getting rid of consecutive identical similarity relations. The same is true for the last and the first constraints.

Theorem 7.3.3. *If a conjunction of primitive constraints contains an occurrence cycle modulo symmetry of $\simeq^?_{\mathcal{R},\lambda}$, then it has no solution.*

Proof. In similarity relations, symbols of different arities can not be similar. Therefore, similar terms have the same set of positions, i.e., as trees they are the same up to renaming of nodes.

We prove by contradiction. Assume that the given occurrence cycle has a solution ϑ. It means that the following term pairs have the same structure: $x_1\vartheta$ and $t_1[x_2]\vartheta$, $x_2\vartheta$ and $t_2[x_3]\vartheta$, …, $x_n\vartheta$ and $t_n[x_1]\vartheta$. Then $x_1\vartheta$ and $t_1[t_2[\cdots[t_n[x_1]]\cdots]]\vartheta$ have the same structure. Since at least one of t_i's is not a variable, $x_1\vartheta$ is a proper subterm of $t_1[t_2[\cdots[t_n[x_1]]\cdots]]\vartheta$. But a term and its proper subterm can not have the same structure. We reached thus a contradiction.

The phrase "modulo symmetry of $\simeq^?_{\mathcal{R},\lambda}$" in the theorem means that the sides of primitive constraints can be swapped, in order to detect an occurrence cycle. Since side swapping does not affect solvability of constraints, the theorem remains true if an occurrence cycle is not in the explicit form in the constraint. $\square$

Below, when we talk about existence of an occurrence cycle in a constraint, we mean existence modulo symmetry of the similarity predicate.

The rules in the mixed group are rules for occurrence check (**Occ-mix**), mismatch (**Mism-mix**), and elimination of term variables (**TVE-mix**) and of function variables (**FVE-mix**). All of them except **FVE-mix** have the form $\mathcal{K} \rightsquigarrow \mathcal{K}'$. As usual, they define the constraint transformation $\mathcal{K} \vee \mathcal{C} \rightsquigarrow \mathcal{K}' \vee \mathcal{C}$. As for **FVE-mix**, its form is $\mathcal{K} \rightsquigarrow \mathcal{K}'_1 \vee \cdots \vee \mathcal{K}'_n$, defining a transformation $\mathcal{K} \vee \mathcal{C} \rightsquigarrow \mathcal{K}'_1 \vee \cdots \vee \mathcal{K}'_n \vee \mathcal{C}$. Note that $\mathcal{C}$ does not change in any of these rules.

In all the rules it is assumed that the constraint to be transformed (i.e., the constraint in the left side of $\rightsquigarrow$) has the form $\mathcal{K}_{\doteq} \wedge \mathcal{K}_{\mathcal{R}_1,\lambda_1} \wedge \cdots \wedge \mathcal{K}_{\mathcal{R}_m,\lambda_m}$, where $\mathcal{K}_{\doteq}$ and each $\mathcal{K}_{\mathcal{R}_i,\lambda_i}$, $1 \leqslant i \leqslant m$, are in *solved form*.

The **TVE-mix** rule uses the *renaming function* ρ. Applied to a term, ρ gives its fresh copy, obtained by replacing each occurrence of a constant from $\mathcal{F}$ by a new function variable, each occurrence of a term variable by a fresh term variable, and each occurrence of a function variable by a fresh function variable. For instance, if the term is $f(F(a, x, x, f(a)))$, we have $\rho(f(F(a, x, x, f(a)))) = G_1(G_2(G_3(), y_1, y_2, G_4(G_5())))$, where $G_1, G_2, G_3, G_4, G_5 \in \mathcal{V}_{\mathsf{F}}$ are new function variables and $y_1, y_2 \in \mathcal{V}_{\mathsf{T}}$ are new term variables.

Occ-mix : $x \simeq^?_{\mathcal{R},\lambda} t \wedge \mathcal{K} \rightsquigarrow \mathsf{false}$,
if $x \simeq^?_{\mathcal{R},\lambda} t \wedge \mathcal{K}$ contains an occurrence cycle for x.

Mism-mix : $x \simeq^?_{\mathcal{R}_1,\lambda_1} \mathsf{f}(t_1, \ldots, t_n) \wedge x \simeq^?_{\mathcal{R}_2,\lambda_2} \mathsf{g}(s_1, \ldots, s_m) \wedge \mathcal{K} \rightsquigarrow \mathsf{false}$,
if $\mathcal{R}_1 \neq \mathcal{R}_2$ and $m \neq n$.

TVE-mix : $x \simeq^?_{\mathcal{R},\lambda} \mathsf{f}(t_1, \ldots, t_n) \wedge \mathcal{K} \rightsquigarrow x \doteq F(t'_1, \ldots, t'_n) \wedge F \simeq^?_{\mathcal{R},\lambda} \mathsf{f} \wedge$
$t'_1 \simeq^?_{\mathcal{R},\lambda} t_1 \wedge \cdots \wedge t'_n \simeq^?_{\mathcal{R},\lambda} t_n \wedge \mathcal{K}\vartheta$,
where $x \in \mathcal{V}(\mathcal{K})$, $x \simeq^?_{\mathcal{R}_1,\lambda_1} \mathsf{f}(t_1, \ldots, t_n) \wedge \mathcal{K}$ does not contain an occurrence cycle for x, $F(t'_1, \ldots, t'_n) = \rho(\mathsf{f}(t_1, \ldots, t_n))$, and $\vartheta = \{x \mapsto F(t'_1, \ldots, t'_n)\}$.

FVE-mix : $F \simeq^?_{\mathcal{R},\lambda} f \wedge \mathcal{K} \rightsquigarrow \vee_{g \in \mathbf{nb}(f,\mathcal{R},\lambda)} \big(F \doteq g \wedge \mathcal{K}\{F \mapsto g\}\big)$, where $F \in \mathcal{V}(\mathcal{K})$.

By $\mathfrak{Mix}$ we denote one application of any of the mixed rules.

Lemma 7.3.3 (Soundness lemma for $\mathfrak{Mix}$)**.** *If $\mathcal{K} \rightsquigarrow \mathcal{C}$ is a step performed by a rule in $\mathfrak{Mix}$, and $\sigma \in Sol(\mathcal{C})$, then $\sigma \in Sol(\mathcal{K})$.*

Proof. For failing rules it is trivial as false has no solution. For **FVE-mix**, the definition of neighborhood implies it. For **TVE-mix** we reason as follows: Let $\mathcal{K} = \{x \simeq^?_{\mathcal{R},\lambda} \mathsf{f}(t_1, \ldots, t_n)\} \wedge \mathcal{K}_1$ and σ be a solution of the right-hand side of this rule. Then $x\sigma = F(t'_1, \ldots, t'_n)\sigma \simeq_{\mathcal{R},\lambda} \mathsf{f}(t_1, \ldots, t_n)\sigma$ and σ solves $x \simeq^?_{\mathcal{R},\lambda} \mathsf{f}(t_1, \ldots, t_n)$. For any other equation $eq \in \mathcal{K}_1$, we have $eq\vartheta$ in the right-hand side, where $\vartheta = \{x \mapsto F(t'_1, \ldots, t'_n)\}$. On the other hand, σ is a solution of $eq\vartheta$ iff $\vartheta\sigma$ is a solution of eq. The equality $x\sigma = F(t'_1, \ldots, t'_n)\sigma$ implies $\vartheta\sigma = \sigma$. Hence, σ is a solution of eq. □

Our constraint solving algorithm Solve is designed as a strategy of applying $\mathfrak{Unif}$, $\mathfrak{Sim}$, and $\mathfrak{Mix}$. To solve a conjunction of primitive equality and similarity constraints $\mathcal{K} = \mathcal{K}_{\doteq} \wedge \mathcal{K}_{\mathcal{R}_1,\lambda_1} \wedge \cdots \wedge \mathcal{K}_{\mathcal{R}_m,\lambda_m}$, it performs the following steps:

```
$\mathcal{C} := \mathcal{K}$
while $\mathcal{C}$ is not in the appr-solved form do
    $\mathcal{C} := \mathfrak{Unif}(\mathcal{C})$, if $\mathcal{C} = \mathsf{false}$, return $\mathsf{false}$
    $\mathcal{C} := \mathfrak{Sim}(\mathcal{C})$, if $\mathcal{C} = \mathsf{false}$, return $\mathsf{false}$
    $\mathcal{C} := \mathfrak{Mix}(\mathcal{C})$, if $\mathcal{C} = \mathsf{false}$, return $\mathsf{false}$
return $\mathcal{C}$
```

We write $\mathsf{Solve}(\mathcal{K}) = \mathcal{C}$, if the algorithm returns $\mathcal{C}$ for the input $\mathcal{K}$. Respectively, $\mathsf{Solve}(\mathcal{K}) = \mathsf{false}$ if false is returned.

Example 7.3.2. Let $\mathcal{R}_1$ and $\mathcal{R}_2$ be the relations defined in Example 7.2.3 and illustrate the steps Solve would make to solve $x \simeq^?_{\mathcal{R}_1,0.5} f(a_1, a_2) \wedge x \simeq^?_{\mathcal{R}_2,0.6} f(y, y)$. We will explicitly distinguish between function variables and terms made of a function variable only, i.e., between F and $F()$. For the same reason, we write constant-terms a_1 and a_2 in their full form $a_1()$ and $a_2()$. Primitive constraints selected to perform a particular step are underlined.

$$\underline{x \simeq^?_{\mathcal{R}_1,0.5} f(a_1(), a_2())} \wedge x \simeq^?_{\mathcal{R}_2,0.6} f(y, y) \rightsquigarrow_{\mathsf{TVE\text{-}mix}}$$

$$
\begin{aligned}
& x \doteq F(G_1(), G_2()) \wedge F \simeq^?_{\mathcal{R}_1,0.5} f \wedge \underline{G_1() \simeq^?_{\mathcal{R}_1,0.5} a_1()} \wedge \underline{G_2() \simeq^?_{\mathcal{R}_1,0.5} a_2()} \wedge \\
& \quad F(G_1(), G_2()) \simeq^?_{\mathcal{R}_2,0.6} f(y,y) \rightsquigarrow_{\mathsf{Dec\text{-}sim}\times 2} \\
& x \doteq F(G_1(), G_2()) \wedge F \simeq^?_{\mathcal{R}_1,0.5} f \wedge G_1 \simeq^?_{\mathcal{R}_1,0.5} a_1 \wedge G_2 \simeq^?_{\mathcal{R}_1,0.5} a_2 \wedge \\
& \quad \underline{F(G_1(), G_2()) \simeq^?_{\mathcal{R}_2,0.6} f(y,y)} \rightsquigarrow_{\mathsf{Dec\text{-}sim,Ori\text{-}sim}} \\
& x \doteq F(G_1(), G_2()) \wedge F \simeq^?_{\mathcal{R}_1,0.5} f \wedge G_1 \simeq^?_{\mathcal{R}_1,0.5} a_1 \wedge G_2 \simeq^?_{\mathcal{R}_1,0.5} a_2 \wedge \\
& \quad F \simeq^?_{\mathcal{R}_2,0.6} f \wedge \underline{y \simeq^?_{\mathcal{R}_2,0.6} G_1()} \wedge G_2() \simeq^?_{\mathcal{R}_2,0.6} y \rightsquigarrow_{\mathsf{Elim\text{-}sim}} \\
& x \doteq F(G_1(), G_2()) \wedge F \simeq^?_{\mathcal{R}_1,0.5} f \wedge G_1 \simeq^?_{\mathcal{R}_1,0.5} a_1 \wedge G_2 \simeq^?_{\mathcal{R}_1,0.5} a_2 \wedge \\
& \quad F \simeq^?_{\mathcal{R}_2,0.6} f \wedge y \simeq^?_{\mathcal{R}_2,0.6} G_1() \wedge \underline{G_2() \simeq^?_{\mathcal{R}_2,0.6} G_1()} \rightsquigarrow_{\mathsf{Dec\text{-}sim}} \\
& x \doteq F(G_1(), G_2()) \wedge \underline{F \simeq^?_{\mathcal{R}_1,0.5} f} \wedge G_1 \simeq^?_{\mathcal{R}_1,0.5} a_1 \wedge G_2 \simeq^?_{\mathcal{R}_1,0.5} a_2 \wedge \\
& \quad F \simeq^?_{\mathcal{R}_2,0.6} f \wedge y \simeq^?_{\mathcal{R}_2,0.6} G_1() \wedge G_2 \simeq^?_{\mathcal{R}_2,0.6} G_1 \rightsquigarrow_{\mathsf{FVE\text{-}mix}} \\
& x \doteq f(G_1(), G_2()) \wedge F \doteq f \wedge G_1 \simeq^?_{\mathcal{R}_1,0.5} a_1 \wedge G_2 \simeq^?_{\mathcal{R}_1,0.5} a_2 \wedge \\
& \quad \underline{f \simeq^?_{\mathcal{R}_2,0.6} f} \wedge y \simeq^?_{\mathcal{R}_2,0.6} G_1() \wedge G_2 \simeq^?_{\mathcal{R}_2,0.6} G_1 \rightsquigarrow_{\mathsf{Del\text{-}sim}} \\
& x \doteq f(G_1(), G_2()) \wedge F \doteq f \wedge \underline{G_1 \simeq^?_{\mathcal{R}_1,0.5} a_1} \wedge \underline{G_2 \simeq^?_{\mathcal{R}_1,0.5} a_2} \wedge \\
& \quad y \simeq^?_{\mathcal{R}_2,0.6} G_1() \wedge G_2 \simeq^?_{\mathcal{R}_2,0.6} G_1 \rightsquigarrow_{\mathsf{FVE\text{-}mix}\times 2} \\
& \big(x \doteq f(b_1(), b_2()) \wedge F \doteq f \wedge G_1 \doteq b_1 \wedge G_2 \doteq b_2 \wedge \\
& \quad y \simeq^?_{\mathcal{R}_2,0.6} b_1() \wedge \underline{b_2 \simeq^?_{\mathcal{R}_2,0.6} b_1}\big) \vee \\
& \big(x \doteq f(c_1(), c_2()) \wedge F \doteq f \wedge G_1 \doteq c_1 \wedge G_2 \doteq c_2 \wedge \\
& \quad y \simeq^?_{\mathcal{R}_2,0.6} c_1() \wedge \underline{c_2 \simeq^?_{\mathcal{R}_2,0.6} c_1}\big) \rightsquigarrow_{\mathsf{Del\text{-}sim}\times 2} \\
& \big(x \doteq f(b_1(), b_2()) \wedge F \doteq f \wedge G_1 \doteq b_1 \wedge G_2 \doteq b_2 \wedge y \simeq^?_{\mathcal{R}_2,0.6} b_1()\big) \vee \\
& \big(x \doteq f(c_1(), c_2()) \wedge F \doteq f \wedge G_1 \doteq c_1 \wedge G_2 \doteq c_2 \wedge y \simeq^?_{\mathcal{R}_2,0.6} c_1()\big).
\end{aligned}
$$

Restricting the obtained result to the original variables (and writing constant-terms in the conventional way), we get the solved form

$$(x \doteq f(b_1, b_2) \wedge y \simeq_{\mathcal{R}_2,0.6} b_1) \vee (x \doteq f(c_1, c_2) \wedge y \simeq_{\mathcal{R}_2,0.6} c_1).$$

Example 7.3.3. Now we show how Solve computes an appr-solved form for the constraint from Example 7.2.4:

$$
\begin{aligned}
& \underline{x \simeq^?_{\mathcal{R}_1,0.6} f(y,y)} \wedge x \simeq^?_{\mathcal{R}_2,0.5} f(z,z) \rightsquigarrow_{\mathsf{TVE\text{-}mix}} \\
& x \doteq F(x_1, x_2) \wedge \underline{F \simeq^?_{\mathcal{R}_1,0.6} f} \wedge x_1 \simeq^?_{\mathcal{R}_1,0.6} y \wedge x_2 \simeq^?_{\mathcal{R}_1,0.6} y \wedge
\end{aligned}
$$

$$
\begin{aligned}
& F(x_1,x_2) \simeq^?_{\mathcal{R}_2,0.5} f(z,z) \rightsquigarrow_{\mathsf{FVE\text{-}mix}} \\
x \doteq f(x_1,x_2) \wedge F \doteq f \wedge x_1 \simeq^?_{\mathcal{R}_1,0.6} y \wedge x_2 \simeq^?_{\mathcal{R}_1,0.6} y \wedge & \\
& \underline{f(x_1,x_2) \simeq^?_{\mathcal{R}_2,0.5} f(z,z)} \rightsquigarrow_{\mathsf{Dec\text{-}sim,\ Del\text{-}sim}} \\
x \doteq f(x_1,x_2) \wedge F \doteq f \wedge x_1 \simeq^?_{\mathcal{R}_1,0.6} y \wedge x_2 \simeq^?_{\mathcal{R}_1,0.6} y \wedge & \\
& x_1 \simeq^?_{\mathcal{R}_2,0.5} z \wedge x_2 \simeq^?_{\mathcal{R}_2,0.5} z.
\end{aligned}
$$

The result gives an appr-solved form. If we did not generate new copies for each variable occurrence in the **TVE-mix** rule, we would end up with $x \doteq f(z,z) \wedge F \doteq f \wedge y \doteq z$. As we saw in Example 7.2.4, some solutions would be lost in this case.

To prove termination of Solve, we will need an ordering on directed acyclic graphs (dags).

Theorem 7.3.4 (Termination of Solve)**.** *The algorithm* Solve *terminates and gives either* false *or a constraint in appr-solved form.*

Proof. Let $\mathcal{K}_0$ be a given conjunction of primitive constraints. We build a term variable dependency graph $G = (\mathit{Vert}, E)$ from $\mathcal{K}_0$ and maintain it during the process of solving. The construction is similar to the one in the proof of Theorem 4.2.1 with the difference that the edges are also labeled by the involved relation names, including the syntactic equality. Hence, two nodes can be connected by multiple edges, i.e., it is actually a multigraph.

The initial version of the graph is denoted by $G_{\mathcal{K}_0}$.

In the process of the application of Solve, the graph gets modified as follows:

a) *The applied rule is of the form $\mathcal{K} \rightsquigarrow \mathcal{K}'$.* Then from the graph $G_{\mathcal{K}}$ we obtain the graph $G_{\mathcal{K}'}$ depending on the rule:

- **Elim-eq** with $x \doteq^? t$ and **Elim-sim** with $x \simeq^?_{\mathcal{R},\lambda} t$ modify the mark of x: $\mathit{mark}(\mathit{vrt}(x), G_{\mathcal{K}'}) := \mathit{mark}(\mathit{vrt}(x), G_{\mathcal{K}})\backslash\{x\}$. Additionally, an edge labeled by $\doteq$ (in case of **Elim-eq**) or by $\mathcal{R}$ (in case of **Elim-sim**) is created from $\mathit{vrt}(x)$ to $\mathit{vrt}(y)$ for all $y \in \mathcal{V}(t)$.
- **TVE-mix** with $x \simeq^?_{\mathcal{R},\lambda} t$ changes the marking for $\mathit{vrt}(x)$ as it is done for **Elim-eq** and **Elim-sim** above and modifies the markings for $\mathit{vrt}(y)$ for each $y \in \mathcal{V}(t)$ as follows: Let $y_1, \ldots, y_m$, $m \geqslant 0$, be all the copies of y created by the renaming function ρ in the **TVE-mix**

step. Then $mark(vrt(y), G_{\mathcal{K}'}) := mark(vrt(y), G_{\mathcal{K}}) \cup \{y_1, \dots, y_m\}$. Moreover, $vrt(y_i)$ is defined as to be $vrt(y)$ for all $1 \leqslant i \leqslant m$. Besides, an edge (labeled by $\mathcal{R}$) is created from $vrt(x)$ to $vrt(y)$ for each $vrt(y) \in \mathcal{V}(t)$, if it does not exist already.

- No other rule of the form $\mathcal{K} \rightsquigarrow \mathcal{K}'$ modifies the graph.

b) *The applied rule is of the form* $\mathcal{K} \rightsquigarrow \mathcal{K}'_1 \vee \cdots \vee \mathcal{K}'_n$, $n > 1$. Then $G_{\mathcal{K}} = G_{\mathcal{K}'_1} = \cdots = G_{\mathcal{K}'_n}$.

One can see that the set *Vert* remains unchanged during the process.

The failing rules stop the algorithm immediately. For the nonfailing ones, as the termination criterion, we consider the lexicographic combination of five measures: (1) the number of missing edges until the variable dependency graph is completed, (2) marked serializations of variable dependency graphs, (3) the number of function variables, (4) the multisets of sizes of equations in each conjunction of primitive constraints, and (5) the number of equations with a non-variable term in the left and a variable in the right. Then

- Del-eq, Dec-eq, Del-sim, and Dec-sim decrease (4) without affecting (1)–(3);
- Ori-eq and Ori-sim decrease (5) without affecting (1)–(4);
- Elim-eq and Elim-sim decrease (1);
- TVE-mix either decreases (1) or leaves it unchanged and decreases (2) with respect to $>_{mark}$.
- FVE-mix decreases (3) without affecting (1)–(2).

If Solve does not stop with false, the only possible non-solved primitive constraints are those between variables, whose left hand side has occurrences in at least two different kind of constraints. For any other case, there is an applicable rule. Hence, the obtained constraint is in appr-solved form. □

Theorem 7.3.5 (Soundness of Solve). *Let $\mathcal{K}$ be a conjunction of primitive constraints. Then every solution of the constraint* Solve($\mathcal{K}$) *is a solution of* $\mathcal{K}$.

Proof. By induction on the length of a rule application sequence leading from $\mathcal{K}$ to Solve($\mathcal{K}$), using the soundness lemmas for equality, similarity, and mixed rules (Lemmas 7.3.1, 7.3.2, 7.3.3). □

Theorem 7.3.6 (Completeness of Solve)**.** *Let $\mathcal{K}$ be a conjunction of primitive constraints, and ϑ be its solution. Then $\mathsf{Solve}(\mathcal{K})$ is a constraint $(\mathcal{K}_{\text{sol}} \wedge \mathcal{K}_{\text{var}}) \vee \mathcal{C}$, where $\mathcal{K}_{\text{sol}} \wedge \mathcal{K}_{\text{var}}$ is in appr-solved form, and $\sigma_{\mathcal{K}_{\text{sol}}}\sigma_{\mathcal{K}_{\text{var}}} \preceq_{\mathcal{W}(\mathcal{K})} \vartheta$, where $\sigma_{\mathcal{K}_{\text{sol}}}$ is the substitution induced by $\mathcal{K}_{\text{sol}}$, and $\sigma_{\mathcal{K}_{\text{var}}}$ is a solution of $\mathcal{K}_{\text{var}}$.*

Proof. In the proof we use completeness of unification and weak unification algorithms [8, 34, 69]. First, note that if one of the failure rules is applicable to a constraint, then it has no solution. For $\mathsf{Occ\text{-}mix}$ it follows from Theorem 7.3.3. For $\mathsf{Mism\text{-}mix}$, it is guaranteed by the fact that symbols with different arities are not similar. For failure rules in $\mathfrak{Unif}$ and $\mathfrak{Sim}$ it is known from their completeness results.

Application of $\mathfrak{Unif}$ to $\mathcal{K}$ leads to a new constraint $\mathcal{C}_{\text{un}}$, which contains a solved form $\mathcal{K}_{\text{un-sol}}$ such that $\sigma_{\mathcal{K}_{\text{un-sol}}} \preceq_{\mathcal{W}(\mathcal{K})} \vartheta$. Application of $\mathfrak{Sim}$ to $\mathcal{C}_{\text{un}}$ gives $\mathcal{C}_{\text{sim}}$, which contains a solved form $\mathcal{K}_{\text{sim-sol}}$ (an extension of $\mathcal{K}_{\text{un-sol}}$) such that $\sigma_{\mathcal{K}_{\text{sim-sol}}} \preceq_{\mathcal{W}(\mathcal{K})} \vartheta$. After that, if $\mathsf{TVE\text{-}mix}$ is applicable, we have an equation $x \simeq^{?}_{\mathcal{R},\lambda} \mathsf{f}(t_1, \dots, t_n)$. $\mathsf{TVE\text{-}mix}$ extends the solved form by a new equation $x \doteq^{?} \mathsf{\rho}(\mathsf{f}(t_1, \dots, t_n))$, obtaining $\mathcal{K}_{\text{tve-sol}}$. By definition of $\mathsf{\rho}$, the term $\mathsf{\rho}(\mathsf{f}(t_1, \dots, t_n))$ contains fresh variables for each symbol in $\mathsf{f}(t_1, \dots, t_n)$ and, hence, $\sigma_{\mathcal{K}_{\text{tve-sol}}} \preceq_{\mathcal{W}(\mathcal{K})} \vartheta$. It is important at this step to record which fresh variable is a copy of which original variable, maintaining an function *original-of*$(V') = V$, where $V \in \mathcal{V}(\mathcal{K})$ and V' is zero or more applications of $\mathsf{\rho}$ to it (i.e., V' is V, or its copy, or a copy of its copy etc.). If the rule $\mathsf{FVE\text{-}mix}$ is applicable, we have an equation $F \simeq^{?}_{\mathcal{R},\lambda} f$. We make a step by this rule, adding a new equation $F \doteq^{?}$ *original-of*$(F)\vartheta$ and obtaining a new solved form $\mathcal{K}_{\text{fve-sol}}$. Let φ be the substitution $\{$*original-of*$(F) \mapsto$ *original-of*$(F)\vartheta\}$. Then we have $\sigma_{\mathcal{K}_{\text{fve-sol}}}\varphi \preceq_{\mathcal{W}(\mathcal{K})} \vartheta$. Iterating this process, we do not get false, since $\mathcal{K}$ was solvable. By Theorem 7.3.4, the process terminates with an appr-solved form $\mathcal{K}_{\text{sol}} \wedge \mathcal{K}_{\text{var}}$ such that $\sigma_{\mathcal{K}_{\text{sol}}}\varphi_1 \cdots \varphi_k \preceq_{\mathcal{W}(\mathcal{K})} \vartheta$, where the φ's are substitutions of the form $\{$*original-of*$(V) \mapsto$ *original-of*$(V)\vartheta\}$. Let $\sigma_{\mathcal{K}_{\text{var}}}$ be the restriction of ϑ to the variables of $\mathcal{K}_{\text{var}}$. Then $\sigma_{\mathcal{K}_{\text{var}}}$ is a solution of $\mathcal{K}_{\text{var}}$. And we have $\sigma_{\mathcal{K}_{\text{sol}}}\varphi_1 \cdots \varphi_k\sigma_{\mathcal{K}_{\text{var}}} \preceq_{\mathcal{W}(\mathcal{K})} \vartheta$.

For the φ's, composition is commutative, because they are ground substitutions with disjoint domains. For some of them we have $\sigma_{\mathcal{K}_{\text{sol}}}\varphi_i = \sigma_{\mathcal{K}_{\text{sol}}}$, because at some step we might have solved an equation with *original-of*(V) variable in its left for *original-of*$(V) \in$ *dom*(φ_i). We assume that the φ's in the composition are rearranged so that $\sigma_{\mathcal{K}_{\text{sol}}}\varphi_1 \cdots \varphi_i = \sigma_{\mathcal{K}_{\text{sol}}}\varphi_{i+1} \cdots \varphi_k$. These remaining φ's are those for which the algorithm reached a variables-only equation containing *original-of*(V), which occurs in the domain of one of the φ's. But then $\varphi_{i+1}\cdots\varphi_k$ is a part of $\sigma_{\mathcal{K}_{\mathrm{var}}}$. Hence, we can get rid of them, obtaining $\sigma_{\mathcal{K}_{\mathrm{sol}}}\sigma_{\mathcal{K}_{\mathrm{var}}}$. Hence, we get $\sigma_{\mathcal{K}_{\mathrm{sol}}}\sigma_{\mathcal{K}_{\mathrm{var}}} \preceq_{\mathcal{W}(\mathcal{K})} \vartheta$. □

7.4 Computing approximation degrees

In the algorithm, we have not included the computation of approximation degrees, but it can be done easily. Instead of constraints in DNF of the form $\mathcal{K}_1 \vee \cdots \vee \mathcal{K}_n$, we will be working with expressions (we call them extended constraints) $(K_1, \mathfrak{D}_1) \vee \cdots \vee (K_n, \mathfrak{D}_n)$, where $\mathfrak{D}_1, \ldots, \mathfrak{D}_n$ are approximation degrees. The rules will carry the degree ("computed so far") as an additional parameter, but only two rules would change them: **Del-sim** and **FVE-mix**. Their variants with degree modification would work on constraint-degree pairs ($\uplus$ stands for disjoint union):

$$\begin{aligned}
\mathsf{Del\text{-}sim\text{-}deg}: \quad & \big(\mathsf{t}_1 \simeq^?_{\mathcal{R},\lambda} \mathsf{t}_2 \wedge \mathcal{K},\ \{\langle \mathcal{R}, \mathfrak{d}\rangle\} \uplus \mathfrak{D}\big) \rightsquigarrow \\
& \quad (\mathcal{K},\ \{\langle \mathcal{R}, \min\{\mathfrak{d}, \mathcal{R}(\mathsf{t}_1,\mathsf{t}_2)\}\rangle\} \cup \mathfrak{D}) \\
& \text{where } \mathsf{t}_1,\mathsf{t}_2 \in \mathcal{F} \cup \mathcal{V}_{\mathsf{F}} \cup \mathcal{V}_{\mathsf{T}} \text{ and } \mathcal{R}(\mathsf{t}_1,\mathsf{t}_2) \geqslant \lambda. \\
\mathsf{FVE\text{-}mix\text{-}deg}: \quad & \big(F \simeq^?_{\mathcal{R},\lambda} f \wedge \mathcal{K},\ \{\langle \mathcal{R}, \mathfrak{d}\rangle\} \uplus \mathfrak{D}\big) \rightsquigarrow \\
& \quad \vee_{g \in \mathbf{nb}(f,\mathcal{R},\lambda)} \big(F \doteq g \wedge \mathcal{K}\{F \mapsto g\}, \\
& \quad \{\langle \mathcal{R}, \min\{\mathfrak{d}, \mathcal{R}(f,g)\}\rangle\} \cup \mathfrak{D}\big), \\
& \text{where } F \in \mathcal{V}(\mathcal{K}).
\end{aligned}$$

For any other rule **R** of the form $\mathcal{K} \rightsquigarrow \mathcal{K}_1 \vee \cdots \vee \mathcal{K}_n$, $n \geqslant 1$, its degree variant **R-deg** will have the form $(\mathcal{K}, \mathfrak{D}) \rightsquigarrow (\mathcal{K}_1, \mathfrak{D}) \vee \cdots \vee (\mathcal{K}_n, \mathfrak{D})$, i.e., $\mathfrak{D}$ will not change. Let us denote the corresponding versions of $\mathfrak{Unif}$, $\mathfrak{Sim}$, and $\mathfrak{Mix}$ by **Unif-deg**, **Sim-deg**, and **Mix-deg**. The notions of solved and approx-solved forms generalize directly to extended constraints. Then we can define **Solve-deg** along the lines of **Solve**: To solve a conjunction of primitive equality and similarity constraints $\mathcal{K}$ with respect to similarity relations $\mathcal{R}_1, \ldots, \mathcal{R}_m$, it performs the following steps:

```
$\mathcal{C} := (\mathcal{K}, \{\langle \mathcal{R}_1, 1\rangle, \ldots, \langle \mathcal{R}_m, 1\rangle\})$
while $\mathcal{C}$ is not in the appr-solved form do
    $\mathcal{C}$ := Unif-deg($\mathcal{C}$), if $\mathcal{C}$ = false, return false
    $\mathcal{C}$ := Sim-deg($\mathcal{C}$), if $\mathcal{C}$ = false, return false
    $\mathcal{C}$ := Mix-deg($\mathcal{C}$), if $\mathcal{C}$ = false, return false
return $\mathcal{C}$
```

7.5 Summary

Multiple similarities cause the transitivity property to be lost, which makes the problem particularly challenging. In this respect, the problem of solving multiple similarity constraints resembles the problem of solving proximity constraints, but as we have shown in the introduction, there are good reasons why proximity-based algorithms are not quite adequate for it. It justifies the design of a dedicated method for multiple constraints.

The algorithm Solve presented in the chapter solves positive equational and similarity constraints, where multiple similarity relations are permitted. Given such a constraint in DNF, it computes a disjunction of approximately solved forms, from which solution substitutions can be read off. It can be easily extended to include the computation of approximation degrees of the solutions. The algorithm is terminating, sound, and complete.

Chapter 8

Conclusion

In this thesis we developed fundamental symbolic algorithms for automating approximate quantitative reasoning, where approximate information is expressed by fuzzy proximity and (multiple) similarity relations. These relations replace the exact equality by its quantitative extension that expresses proximity or similarity between objects up to a certain degree.

The techniques that we studied address the problems of solving approximate equational and generalization constraints. Different versions of unification and matching algorithms have been designed for equational constraint solving, and anti-unification algorithms have been proposed for generalization constraints.

Proximity relations are the fuzzy counterpart of tolerance (reflexive and symmetric) relations, while similarities are seen as fuzzy equivalences. In this sense, similarity is a special case of proximity (restricted by transitivity), but when dealing with multiple similarities over the same domain, one can not rely on the transitivity anymore. Hence, in our problems, we had to deal with the lack of transitivity in both proximity and multiple similarity settings. It posed a significant challenge, since the standard techniques from the unification theory do not apply, and our methods had to address it.

Proximity relations (and their crisp counterpart, tolerance relations) can be represented by weighted or non-weighted undirected graphs, and the existing approaches to proximity-based constraint solving can be characterized by the way how proximal nodes are treated in such a graph. In the block-based approach, two symbols are considered proximal if they belong to the same maximal clique partition in this graph. In the class-based approach, a symbol is proximal to any of its neighbors in the graph. Both approaches can be used to extend the constraint solving methods from a single similarity relation to proximity or multiple similarity relations and have their pros and cons. We characterized these approaches in this thesis, briefly summarized the existing block-based algorithms for proximity unification, proposed a block-based anti-unification algorithm for proximities, and developed class-based unification, matching, and anti-unification methods in several theories with proximity and multiple similarity relations. These novel results can be summarized as follows:

- For the fuzzy signatures where mismatches are allowed only in symbol names:
  - A block-based anti-unification algorithm for proximity relations (Section 3.3) and its subalgorithm for computing maximal clique partition in undirected graphs (Section 3.3.2);
  - Pre-unification and neighborhood constraint solving algorithms, forming a class-based unification algorithm (Section 4.2). The latter can be seen as a generalization of Sessa's weak unification algorithm from similarity to proximity relations in the class-based setting;
  - A class-based matching algorithm for proximity relations that works on a compact set-based representation of terms and substitutions (Section 4.3);
  - A class-based anti-unification algorithm for proximity relations that works on a compact set-based representation of terms and substitutions (Section 4.4);
  - A class-based equational constraint solving algorithm for multiple similarity relations (Chapter 7), which generalizes Sessa's algorithm from a single similarity relation to multiple similarity relations over the same domain.
- For the fuzzy signatures where mismatches are allowed both in symbol names and arities (fully fuzzy signatures):
  - A class-based unification algorithm (Section 5.3), which generalizes from similarity to proximity relations the unification algorithm proposed by Aït-Kaci and Paci;
  - A class-based matching algorithm for proximity relations (Section 5.4);
  - A generic framework for class-based anti-unification algorithms (Section 5.5) whose special cases form several concrete ones, including the one that generalizes the anti-unification algorithm by Aït-Kaci and Paci from similarity to proximity relations.

- Application: a calculus of rule-based programming with conditional transformation rules, where the core computational mechanism is an extension of matching with proximity relations (Chapter 6).

For all the algorithms described in this thesis, we proved termination, soundness, and completeness theorems. The complexity issues have been addressed as well.

Our results open a way for future work in several directions. An interesting problem would be to study proximity-based unification in equational theories. Argument relations that we introduced in Chapter 5 can be represented as a version of regular, collapse-free, shallow theories, which have been studied quite intensively in first-order equational unification. A first step towards equational proximity-based unification would be investigating it modulo such theories. In general, one may have several versions of proximity-based unification modulo equational theories depending whether the background knowledge is crisp, fuzzy, or mixed. It is quite a large area to explore where, to the best of our knowledge, not much research has bee done so far.

Another interesting and important direction is to bring the theoretical advances to practical applications. Proximity- and similarity-based unification has been used successfully in fuzzy logic programming and flexible query answering systems, proximity-based matching has been incorporated in a rule-based programming tool, but the unexplored potential of these techniques is still big. In general, quantitative theories have many useful applications, some most recent ones being related to artificial intelligence, program verification, probabilistic programming, or natural language processing. Many tasks arising in these areas require reasoning methods and computational tools that deal with quantitative information. For instance, constraint logic programming, knowledge engineering, approximate automated reasoning, program analysis and transformation methods could find approximate unification and matching techniques useful. Approximate inductive reasoning, reasoning and programming by analogy, similarity detection in programming language statements or in natural language texts could benefit from solving approximate generalization constraints. These and many related problems can form the area of many interesting practical applications for fundamental symbolic proximity- and similarity-based techniques.

Bibliography

[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, Namur, Belgium, October 10-12, 2017, Revised Selected Papers*, volume 10855 of *Lecture Notes in Computer Science*, pages 218–234. Springer, 2017.

[2] H. Aït-Kaci and G. Pasi. Fuzzy lattice operations on first-order terms over signatures with similar constructors: A constraint-based approach. *Fuzzy Sets Syst.*, 391:1–46, 2020.

[3] H. Aït-Kaci and Y. Sasaki. An axiomatic approach to feature term generalization. In L. D. Raedt and P. A. Flach, editors, *ECML*, volume 2167 of *Lecture Notes in Computer Science*, pages 1–12. Springer, 2001.

[4] M. Alpuente, S. Escobar, J. Espert, and J. Meseguer. A modular order-sorted equational generalization algorithm. *Inf. Comput.*, 235:98–136, 2014.

[5] K. R. Apt. Logic programming. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science, Volume B: Formal Models and Semantics*, pages 493–574. Elsevier and MIT Press, 1990.

[6] K. R. Apt and R. N. Bol. Logic programming and negation: A survey. *J. Log. Program.*, 19/20:9–71, 1994.

[7] F. Baader and T. Nipkow. *Term rewriting and all that.* Cambridge University Press, 1998.

[8] 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.

[9] 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.

[10] A. Baumgartner, T. Kutsia, J. Levy, and M. Villaret. Higher-order pattern anti-unification in linear time. *J. Autom. Reason.*, 58(2):293–310, 2017.

[11] 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.

[12] J. Bhasker and T. Samad. The clique-partitioning problem. *Computers and Mathematics with Applications*, 22(6):1–11, 1991.

[13] C. Bron and J. Kerbosch. Finding all cliques of an undirected graph (algorithm 457). *Commun. ACM*, 16(9):575–576, 1973.

[14] F. Cazals and C. Karande. A note on the problem of reporting maximal cliques. *Theor. Comput. Sci.*, 407(1-3):564–568, 2008.

[15] H. Comon, M. Haberstrau, and J. Jouannaud. Syntacticness, cycle-syntacticness, and shallow theories. *Inf. Comput.*, 111(1):154–191, 1994.

[16] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms, 3rd Edition.* MIT Press, 2009.

[17] M. E. Cornejo, J. Medina-Moreno, and C. Rubio-Manzano. Towards a full fuzzy unification in the Bousi∼Prolog system. In *2018 IEEE International Conference on Fuzzy Systems, FUZZ-IEEE 2018, Rio de Janeiro, Brazil, July 8-13, 2018*, pages 1–7. IEEE, 2018.

[18] N. Dershowitz and Z. Manna. Proving termination with multiset orderings. *Commun. ACM*, 22(8):465–476, 1979.

[19] D. Dubois and H. Prade. *Fuzzy sets and systems: theory and applications*, volume 144 of *Mathematics in science and engineering.* Acad. Press, 1980.

[20] B. Dundua. Pρlog: a system for rule-based programming. *CoRR*, abs/2108.11699, 2021.

[21] B. Dundua, M. Florido, T. Kutsia, and M. Marin. *CLP(H):* constraint logic programming for hedges. *TPLP*, 16(2):141–162, 2016.

[22] B. Dundua, T. Kutsia, and M. Marin. Strategies in PρLog. 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.

[23] B. Dundua, T. Kutsia, M. Marin, and C. Pau. Constraint solving over multiple similarity relations. In Z. M. Ariola, editor, *5th International Conference on Formal Structures for Computation and Deduction, FSCD 2020, June 29-July 6, 2020, Paris, France (Virtual Conference)*, volume 167 of *LIPIcs*, pages 30:1–30:19. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2020.

[24] B. Dundua, T. Kutsia, M. Marin, and C. Pau. Extending the ρLog calculus with proximity relations. In G. Jaiani and D. Natroshvili, editors, *Applications of Mathematics and Informatics in Natural Sciences and Engineering*, pages 83–100, Cham, 2020. Springer.

[25] B. Dundua, T. Kutsia, and K. Reisenberger-Hagmayer. An overview of PρLog. 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.

[26] F. A. Fontana and F. Formato. Likelog: A logic programming language for flexible data retrieval. In B. R. Bryant, G. B. Lamont, H. Haddad, and J. H. Carroll, editors, *Proceedings of the 1999 ACM Symposium on Applied Computing, SAC'99, San Antonio, Texas, USA, February 28 - March 2, 1999*, pages 260–267. ACM, 1999.

[27] F. A. Fontana and F. Formato. A similarity-based resolution rule. *Int. J. Intell. Syst.*, 17(9):853–872, 2002.

[28] F. Formato, G. Gerla, and M. I. Sessa. Extension of logic programming by similarity. In M. C. Meo and M. V. Ferro, editors, *1999 Joint Conference on Declarative Programming, AGP'99*, pages 397–410, 1999.

[29] F. Formato, G. Gerla, and M. I. Sessa. Similarity-based unification. *Fundam. Inform.*, 41(4):393–414, 2000.

[30] 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.

[31] T. R. Jensen and B. Toft. *Graph coloring problems*, volume 39. John Wiley & Sons, 2011.

[32] 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.

[33] P. Julián-Iranzo and C. Rubio-Manzano. Proximity-based unification theory. *Fuzzy Sets and Systems*, 262:21–43, 2015.

[34] P. Julián-Iranzo and C. Rubio-Manzano. A sound and complete semantics for a similarity-based logic programming language. *Fuzzy Sets and Systems*, 317:1–26, 2017.

[35] P. Julián-Iranzo and F. Sáenz-Pérez. An efficient proximity-based unification algorithm. In *2018 IEEE International Conference on Fuzzy Systems, FUZZ-IEEE 2018*, pages 1–8. IEEE, 2018.

[36] P. Julián-Iranzo and F. Sáenz-Pérez. Proximity-based unification: an efficient implementation method. *IEEE Transactions on Fuzzy Systems*, 2020.

[37] D. Kapur and P. Narendran. Matching, unification and complexity. *SIGSAM Bull.*, 21(4):6–9, 1987.

[38] R. M. Karp. Reducibility among combinatorial problems. In R. E. Miller and J. W. Thatcher, editors, *Proceedings of a symposium on the Complexity of Computer Computations, March 20-22, 1972, IBM Thomas J. Watson Research Center, Yorktown Heights, New York, USA*, The IBM Research Symposia Series, pages 85–103. Plenum Press, New York, 1972.

[39] E. Klement, R. Mesiar, and E. Pap. *Triangular Norms*, volume 8 of *Trends in Logic*. Springer, 2000.

[40] S. Krajci, R. Lencses, J. Medina, M. Ojeda-Aciego, and P. Vojtás. A similarity-based unification model for flexible querying. In T. Andreasen,

A. Motro, H. Christiansen, and H. L. Larsen, editors, *Flexible Query Answering Systems, 5th International Conference, FQAS 2002, Copenhagen, Denmark, October 27-29, 2002, Proceedings*, volume 2522 of *Lecture Notes in Computer Science*, pages 263–273. Springer, 2002.

[41] J. Krajícek and P. Pudlák. The number of proof lines and the size of proofs in first order logic. *Arch. Math. Log.*, 27(1):69–84, 1988.

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

[43] T. Kutsia, J. Levy, and M. Villaret. Anti-unification for unranked terms and hedges. *J. Autom. Reasoning*, 52(2):155–190, 2014.

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

[45] 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.

[46] T. Kutsia and C. Pau. Computing all maximal clique partitions in a graph. Report 19-04, RISC, Johannes Kepler University Linz, 2019.

[47] T. Kutsia and C. Pau. Matching and generalization modulo proximity and tolerance relations. In A. Özgün and Y. Zinova, editors, *Language, Logic, and Computation - 13th International Tbilisi Symposium, TbiLLC 2019, Batumi, Georgia, September 16-20, 2019, Revised Selected Papers*, volume 13206 of *Lecture Notes in Computer Science*, pages 323–342. Springer, 2019.

[48] T. Kutsia and C. Pau. Solving proximity constraints. In M. Gabbrielli, editor, *Logic-Based Program Synthesis and Transformation - 29th International Symposium, LOPSTR 2019, Porto, Portugal, October 8-10, 2019, Revised Selected Papers*, volume 12042 of *Lecture Notes in Computer Science*, pages 107–122. Springer, 2019.

[49] T. Kutsia and C. Pau. A framework for approximate generalization in quantitative theories. In J. Blanchette, L. Kovacs, and D. Pattinson, editors, *Automated Reasoning - 11th International Joint Conference,*

IJCAR 2022, Haifa, Israel, August 8-10, 2022, Proceedings, Lecture Notes in Computer Science. Springer, 2022. To appear.

[50] R. C. T. Lee. Fuzzy logic and the resolution principle. *J. ACM*, 19(1):109–119, 1972.

[51] J. W. Lloyd. *Foundations of Logic Programming, 2nd Edition.* Springer, 1987.

[52] R. Mardare, P. Panangaden, and G. D. Plotkin. Quantitative algebraic reasoning. In M. Grohe, E. Koskinen, and N. Shankar, editors, *Proceedings of the 31st Annual ACM/IEEE Symposium on Logic in Computer Science, LICS '16, New York, NY, USA, July 5-8, 2016*, pages 700–709. ACM, 2016.

[53] M. Marin. A system for rule-based programming in Mathematica. Available from http://staff.fmi.uvt.ro/˜mircea.marin/rholog/, 2019.

[54] 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, *Proc. 4th International Workshop on the Implementation of Logics (WIL'03)*, pages 55–68, Almaty, Kazakhstan, 2003.

[55] M. Marin and T. Kutsia. Foundations of the rule-based system ρLog. *Journal of Applied Non-Classical Logics*, 16(1-2):151–168, 2006.

[56] M. Marin and F. Piroi. Rule-based programming with Mathematica. In *Proc. 6th Int. Mathematica Symposium, Alberta, Canada*, 2004.

[57] A. Martelli and U. Montanari. An efficient unification algorithm. *ACM Trans. Program. Lang. Syst.*, 4(2):258–282, 1982.

[58] 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.

[59] J. Medina, M. Ojeda-Aciego, and P. Vojtás. Similarity-based unification: a multi-adjoint approach. *Fuzzy Sets and Systems*, 146(1):43–62, 2004.

[60] P. Nguyen. *Meta-mining: a meta-learning framework to support the recommendation, planning and optimization of data mining workflows.* PhD thesis, Dept. of Computer Science, University of Geneva, 2015.

[61] T. Nipkow. Combining matching algorithms: The regular case. *J. Symb. Comput.*, 12(6):633–654, 1991.

[62] C. Pau and T. Kutsia. Proximity-based unification and matching for fully fuzzy signatures. In *30th IEEE International Conference on Fuzzy Systems, FUZZ-IEEE 2021, Luxembourg, July 11-14, 2021*, pages 1–6. IEEE, 2021.

[63] G. D. Plotkin. A note on inductive generalization. *Machine Intel.*, 5(1):153–163, 1970.

[64] J. H. Poincaré. L'espace et la géomètrie. *Revue de métaphysique et de morale*, 3:631–646, 1895.

[65] L. D. Raedt and A. Kimmig. Probabilistic (logic) programming concepts. *Machine Learning*, 100(1):5–47, 2015.

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

[67] J. A. Robinson. A machine-oriented logic based on the resolution principle. *J. ACM*, 12(1):23–41, 1965.

[68] M. Rodríguez-Artalejo and C. A. Romero-Díaz. A declarative semantics for CLP with qualification and proximity. *TPLP*, 10(4-6):627–642, 2010.

[69] M. I. Sessa. Approximate reasoning by similarity-based SLD resolution. *Theor. Comput. Sci.*, 275(1-2):389–426, 2002.

[70] J. H. Siekmann. *Unification and Matching Problems.* Memo CSA-4-78, Essex University, 1978.

[71] J. H. Siekmann. Unification theory. *J. Symb. Comput.*, 7(3/4):207–274, 1989.

[72] E. Sperner. Ein satz über untermengen einer endlichen menge. *Mathematische Zeitschrift*, 27(1):544–548, 1928. In German.

[73] E. Tomita. Efficient algorithms for finding maximum and maximal cliques and their applications. In S. Poon, M. S. Rahman, and H. Yen, editors, *WALCOM: Algorithms and Computation, 11th International Conference and Workshops, WALCOM 2017, Hsinchu, Taiwan, March 29-31, 2017, Proceedings*, volume 10167 of *Lecture Notes in Computer Science*, pages 3–15. Springer, 2017.

[74] E. Tomita, A. Tanaka, and H. Takahashi. The worst-case time complexity for generating all maximal cliques and computational experiments. *Theor. Comput. Sci.*, 363(1):28–42, 2006.

[75] C.-J. Tseng. *Automated synthesis of data paths in digital systems*. PhD thesis, Dept. of Electrical and Computer Engineering, Carnegie-Mellon University, Pittsburg, PA, 1984.

[76] M. I. Vasileva, B. A. Plummer, K. Dusad, S. Rajpal, R. Kumar, and D. A. Forsyth. Learning type-aware embeddings for fashion compatibility. In V. Ferrari, M. Hebert, C. Sminchisescu, and Y. Weiss, editors, *Computer Vision - ECCV 2018 - 15th European Conference, Munich, Germany, September 8-14, 2018, Proceedings, Part XVI*, volume 11220 of *Lecture Notes in Computer Science*, pages 405–421. Springer, 2018.

[77] A. Veit, S. J. Belongie, and T. Karaletsos. Conditional similarity networks. In *2017 IEEE Conference on Computer Vision and Pattern Recognition, CVPR 2017, Honolulu, HI, USA, July 21-26, 2017*, pages 1781–1789. IEEE Computer Society, 2017.

[78] H. Virtanen. Vague domains, s-unification, logic programming. *Electr. Notes Theor. Comput. Sci.*, 66(5):86–103, 2002.

[79] P. Vojtás. Declarative and procedural semantics of fuzzy similarity based unification. *Kybernetika*, 36(6):707–720, 2000.

[80] S. Wolfram. *The Mathematica book, 5th Edition*. Wolfram-Media, 2003.

[81] 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, 2011.