JOHANNES KEPLER
UNIVERSITÄT LINZ | JKU

Technisch-Naturwissenschaftliche
Fakultät

Anti-Unification Algorithms: Design, Analysis, and Implementation

DISSERTATION

zur Erlangung des akademischen Grades

Doktor der Technischen Wissenschaften

im Doktoratsstudium der

Technischen Wissenschaften

Eingereicht von:
Alexander Baumgartner

Angefertigt am:
Research Institute for Symbolic Computation (RISC)

Beurteilung:
Temur Kutsia (Betreuung)
Mateu Villaret

Linz, September 2015

Eidesstattliche Erklärung

Ich erkläre an Eides statt, dass ich die vorliegende Dissertation selbstständig und ohne fremde Hilfe verfasst, andere als die angegebenen Quellen und Hilfsmittel nicht benutzt bzw. die wörtlich oder sinngemäß entnommenen Stellen als solche kenntlich gemacht habe.

Die vorliegende Dissertation ist mit dem elektronisch übermittelten Textdokument identisch.

Linz, September 2015,

Alexander Baumgartner

Kurzfassung

Das Anti-Unifikations-Problem beschäftigt sich mit der Suche nach einer Generalisierungen (Verallgemeinerung) für zwei Eingabebeispiele. Interessant sind dabei jene Verallgemeinerungen welche am wenigsten generell sind. Informell kann das Problem wie folgt beschrieben werden: Finde einen Ausdruck für zwei konkrete Objekte welcher alle Gemeinsamkeiten beschreibt und deren Unterschiede einheitlich als Variablen darstellt. Solche Verallgemeinerungsprobleme kommen in vielen Gebieten der Computerwissenschaft und der Mathematik vor. Ein Algorithmus welcher die am wenigsten generellen Verallgemeinerungen von zwei Eingabebeispielen berechnet wird Anti-Unifikations-Algorithmus genannt. In dieser Arbeit studieren wir das Anti-Unifikations-Problem in verschiedenen Theorien. Wir entwickeln Algorithmen welche solche Probleme lösen, analysieren deren Eigenschaften, studieren deren Komplexität, demonstrieren deren Anwendung anhand von realitätsnahen Beispielen und präsentieren eine Programmbibliothek von Anti-Unifikations-Algorithmen welche in Java implementiert wurde.

Zur Präsentation der Anti-Unifikations-Algorithmen wird eine regelbasierten Methode mit einem Speicher welcher die Unterschiede in den Eingabebeispielen protokolliert verwendet. Unter gewissen Voraussetzungen kann ein Algorithmus der einen solchen Speicher besitzt sowohl das Matching-Problem als auch das Problem vom Auffinden einer gemeinsamen Instanz (Join) lösen. Manche der in dieser Arbeit entwickelten Algorithmen benötigen einen Minimierungsprozess in dem das Matching-Problem gelöst werden muß. Wir zeigen, dass diese Algorithmen selbst-minimierend sind, das heißt, dass das Matching-Problem vom Anti-Unifikations-Algorithmus selbst gelöst werden kann.

Wir entwickeln Anti-Unifikations-Algorithmen für untypisierte Terme, nominale Terme und für einfach typisierte Lambda-Terme. Für untypisierte Terme betrachten wir Variablen erster Ordnung und Variablen höherer Ordnung. Das Anti-Unifikations-Problem in solchen untypisierten Theorien höherer Ordnung ist endlich aber nicht deterministisch. Deswegen stellen wir einige wenige naturgemäße Bedingungen an die errechneten Generalisierungen, sodass wir eine eindeutige Zuordnung zwischen einer Generalisierungen und einem Gerüst erreichen. (Das Gerüst beinhaltet alle Gemeinsamkeiten der Eingabeterme welche in der Generalisierung erhalten werden.) Eine Generalisierung kann in quadratischer Zeit für ein gegebenes Gerüst berechnet werden. Das heißt, wenn zum Beispiel das Gerüst ein eingeschränkter größter Teilbaum der Eingabeterme ist, dann kann sowohl die Generalisierung als auch das Gerüst in quadratischer Zeit berechnet werden.

Generell ist das nominale Anti-Unifikations-Problem vom Typ Null (d.h., eine minimale komplette Menge von Generalisierungen existiert nicht immer). Wenn jedoch die Menge der erlaubten Atome in einer Generalisierung endlich ist, dann wird das Problem eindeutig lösbar. Dies ist das Anti-Unifikations-Problem welches wir für nominale Terme betrachten. Für einfach typisierte Lambda-Terme beschränken wir die General-

isierungen auf Pattern höherer Ordnung, sodass die errechnete Generalisierung für zwei beliebige Lambda-Terme ein eindeutiges Pattern ist. (In einem Pattern höherer Ordnung sind alle Argumente von freien Variablen unterschiedliche gebundene Variablen.) Wir zeigen einen Algorithmus welcher diese Pattern-Generalisierung in quadratischer Zeit berechnet.

Die implementierte Java-Bibliothek stellt Anti-Unifikations-Algorithmmen für alle obig diskutierten Theorien bereit. Solche Anti-Unifikations-Probleme kommen zum Beispiel in Beweisgeneralisierung, in analogischem Schlußfolgern, im Auffinden von Ähnlichkeiten in XML Dokumenten oder Stücken von Softwarecode etc. vor. Daher kann unsere Programmbibliothek ein wertvoller Bestandteil für Programme sein welche solche Generalisierungsprobleme lösen müssen.

Abstract

The anti-unification problem is concerned with finding a generalization for two input examples. The interesting generalizations are the least general ones. Informally, the problem can be described as: Given two concrete objects, find an expression that describes all their common features and uniformly represents their differences by variables. Such generalization problems arise in many areas of computer science and mathematics. An algorithm that computes least general generalizations for two input examples is called an anti-unification algorithm. In this work, we study the anti-unification problem for various theories. We develop algorithms that solve such problems, analyze their properties, study their computational complexity, demonstrate their usage on real world examples, and present a library of anti-unification algorithms which is implemented in Java.

Anti-unification algorithms are presented in a rule-based manner featuring a store that keeps track of all the differences at the input examples. Under certain circumstances, an algorithm that maintains such a store can be used to solve the matching problem as well as the problem of finding a join. Some of the algorithms developed in this work need a minimization step that requires to solve the matching problem. We show that those algorithms are self-minimizing in the sense that the matching problem can be solved by the anti-unification algorithm itself.

We develop anti-unification algorithms for unranked terms, nominal terms, and simply-typed lambda terms. For unranked terms, we consider first-order variables and higher-order variables. The anti-unification problem in such higher-order unranked theories is finitary but highly nondeterministic. Therefore, we impose a few natural restrictions on the computed generalizations so that we can establish a one-to-one correspondence between a generalization and a skeleton. (The skeleton consists of all those common parts of the input terms that are to be retained in the generalization.) A generalization can be computed in quadratic time for a given skeleton. This means that, for instance, if the skeleton is a constrained longest common subtree of the input terms, then both skeleton and generalization computation can be done in quadratic time.

In general, the problem of nominal anti-unification is of type zero, (i.e., a minimal complete set of generalizations does not always exist) but if the set of atoms permitted in generalizations is finite, then it becomes unitary. This is the anti-unification problem we consider for nominal terms. For simply-typed lambda terms, we restrict generalizations to be higher-order patterns so that a generalization computed for two arbitrary lambda terms is a unique pattern. (In higher-order patterns all the arguments of free variables are distinct bound variables.) We show an algorithm that computes it in quadratic time.

The implemented Java library provides anti-unification algorithms for all the theories discussed above. Such anti-unification problems arise, for instance, in proof generalization, in analogical reasoning, in detection of similarities in XML documents or in pieces of software code, etc. Therefore, our library can be a valuable ingredient for tools that need to solve such generalization problems.

Acknowledgments

I would like to thank Temur Kutsia for supervising this thesis and his great support throughout the last three years. I could not have imagined having a better advisor for my doctoral studies. Furthermore, I would like to thank Jordi Levy and Mateu Villaret. Our collaboration was a very pleasant experience for me and I hope that it will continue for many years.

This research has been partially supported by the Austrian Science Fund (FWF) with the project SToUT (P 24087-N18) and by the strategic program "Innovatives OÖ 2010plus" by the Upper Austrian Government.

Parts of this work have been published in [12, 13, 14, 15].

Contents

Chapter 1

Introduction

In this work, we study the anti-unification problem for different term languages. The anti-unification problem is concerned with finding an expression that generalizes two input examples. Interesting generalizations are the least general ones. They inherit as many common features of the input examples as possible and uniformly represent their differences by variables. Such generalization problems arise in many areas of computer science and mathematics. An algorithm for constructing a least general generalization of two given input examples is called an anti-unification algorithm. We develop such algorithms for three different term languages, namely unranked terms, nominal terms, and simply-typed lambda terms. We discuss the presented algorithms, study their properties, give examples for possible applications, and demonstrate our library of anti-unification algorithms that has been implemented in Java.

We now introduce and discuss the underlying concepts in general. All the notions which are discussed here also hold for more complex objects, like hedges or terms-in-context, which are discussed in the following chapters. For the sake of readability, we will talk about terms.

Let $\mathcal{F}$ be a set of function symbols and $\mathcal{V}$ be a set of variables such that $\mathcal{F} \cap \mathcal{V} = \varnothing$. We denote by $\mathcal{T}(\mathcal{F}, \mathcal{V})$ the set of terms from a given language that is constructed over $\mathcal{F}$ and $\mathcal{V}$. A $\mathcal{T}(\mathcal{F}, \mathcal{V})$-*substitution*, or simply substitution, if the set of terms is irrelevant or clear from the context, is a function $\sigma : \mathcal{V} \to \mathcal{T}(\mathcal{F}, \mathcal{V})$ such that $\{x \in \mathcal{V} \mid \sigma(x) \neq x\}$ is finite. We write $\{x_1 \mapsto t_1, x_2 \mapsto t_2, \ldots, x_n \mapsto t_n\}$ for the substitution that maps the variable x_1 to the term t_1, x_2 to t_2, etc., and all the other variables are mapped to themselves. The *domain* of a substitution σ is the finite set of variables $\mathrm{Dom}(\sigma) ::= \{x \in \mathcal{V} \mid \sigma(x) \neq x\}$ and the *range* is the set of terms $\mathrm{Ran}(\sigma) ::= \{\sigma(x) \mid x \in \mathrm{Dom}(\sigma)\}$.

Any substitution σ can be extended to a mapping $\hat{\sigma} : \mathcal{T}(\mathcal{F}, \mathcal{V}) \to \mathcal{T}(\mathcal{F}, \mathcal{V})$ by induction on the structure of terms. The set $\mathcal{T}(\mathcal{F}, \mathcal{V})$ is assumed to be closed under $\hat{\sigma}$ (i.e. under variable instantiation). For the sake of readability we do not distinguish between a substitution σ and its extension $\hat{\sigma}$. We use postfix notation for substitution application, for instance, $t\sigma$ denotes the application of a substitution σ to a term t. The *composition* of two substitutions σ_1 and σ_2 is written as $\sigma_1\sigma_2$ and defined by $t\sigma_1\sigma_2 = (t\sigma_1)\sigma_2$ for any $t \in \mathcal{T}(\mathcal{F}, \mathcal{V})$. The *identity* substitution is denoted by *Id*. Since the concrete instances of $\mathcal{F}$ and $\mathcal{V}$ do not matter in our general discussion, we simply write $\mathcal{T}$ instead of $\mathcal{T}(\mathcal{F}, \mathcal{V})$.

We define a binary relation $\preceq_{\equiv}$ on $\mathcal{T}$ such that $(t_1, t_2) \in \preceq_{\equiv}$ if there exists a substitution σ such that $t_1\sigma$ and t_2 are equivalent with respect to a given equivalence relation $\equiv$ (e.g., term equality, alpha equivalence, equality modulo some equational theory, etc.). As usual, we write $t_1 \preceq_{\equiv} t_2$ instead of $(t_1, t_2) \in \preceq_{\equiv}$ and we say that t_1 is *more general*

(with respect to $\equiv$) than t_2. Conversely, t_2 is called *more specific* (with respect to $\equiv$) than t_1, or an ($\equiv$-)instance of t_1. If $t_1 \preceq_\equiv t_2$ and $t_2 \preceq_\equiv t_1$, then we say that t_1 and t_2 are *equigeneral* (with respect to $\equiv$) and we write $t_1 \simeq_\equiv t_2$. Furthermore, if $t_1 \preceq_\equiv t_2$ and $t_1 \not\simeq_\equiv t_2$, then t_1 is *strictly more general* (with respect to $\equiv$) than t_2, written $t_1 \prec_\equiv t_2$.

For example, the first-order term $f(x)$ is syntactically (strictly) more general than $f(f(a))$ because the variable x can be instantiated by a term $f(a)$. The two terms $f(x)$ and $g(a)$ are incomparable with respect to $\preceq_=$. Note that $\simeq_=$ is an instance of $\simeq_\equiv$.

Theorem 1.1. *Any relation $\preceq_\equiv$ is reflexive and transitive.*

Proof. Let $t \in \mathcal{T}$ and $\sigma = Id$, then $t\sigma \equiv t$ holds. It follows that $t \preceq_\equiv t$. Furthermore, if $t_1, t_2, t_3 \in \mathcal{T}$ and $t_1 \preceq_\equiv t_2$ and $t_2 \preceq_\equiv t_3$ holds, then there are substitutions σ_1, σ_2 such that $t_1\sigma_1 \equiv t_2$ and $t_2\sigma_2 \equiv t_3$. From transitivity of $\equiv$ it follows that $t_1\sigma_1\sigma_2 \equiv t_3$. □

Hence, the relation $\preceq_\equiv$ is a quasiorder. It is called the *instantiation quasiorder*. It is straightforward to obtain a partial order on $\mathcal{T}$ from the quasiorder $\preceq_\equiv$ by considering term equality modulo $\simeq_\equiv$. We call that partial order the *instantiation order*. The set $\mathcal{T}$ is a meet-semilattice with respect to the instantiation order if for all elements t_1 and t_2 of $\mathcal{T}$, the greatest lower bound (meet) of t_1 and t_2 exists. In many cases the set of terms $\mathcal{T}$ is such a meet-semilattice. For instance, if $\mathcal{T}$ denotes the set of all simply-typed lambda terms of a certain type, or the set of all the unranked terms which we consider in this work. Figure 1.1 illustrates this lattice structure for some simple first-order terms. For instance, the two terms $f(a, x_1)$ and $f(x_2, b)$ have a meet $f(y_1, y_2)$, and also a least upper bound (join) which is $f(a, b)$, while the terms $f(a, x_1)$ and $f(b, b)$ do not have a join. We decided to put $f(y_1, y_2)$ on top of Figure 1.1 because one can see it as the root of its instances. In particular, a single variable x would be the root of all first-order terms. In this work we only consider languages where a meet exists (for terms of compatible types), while a join might not exist. In general, a meet is not unique. Consider, for instance, the terms $f(a)$ and $f(f(a))$ from a language with higher-order variables. There are two incomparable meets, namely $f(X(a))$ and $X(f(a))$, where X is such a higher-order variable.

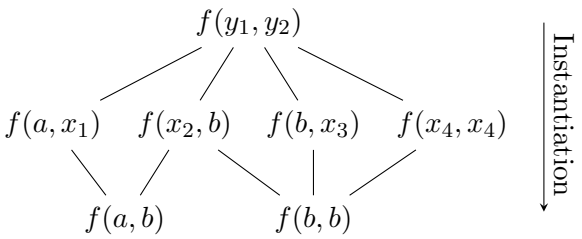

Figure 1.1: Meet and join of some terms. All the terms have pairwise a meet.

A meet of two given terms is usually called their *least general generalization* (lgg) and the process of finding a minimal complete set of lggs for two input terms is called *anti-unification*. Intuitively, the anti-unification problem can be described as follows:

Given: Two input examples (i.e. terms).

Find: A term (i.e. lgg) which inherits all the common features from the input terms, and uniformly represents their differences by variables.

Example 1.1. *For instance, consider the first-order terms* $t_1 = f(f(a,a),a)$ *and* $t_2 = f(f(b,a),b)$*. The term* $f(x,y)$ *generalizes* t_1 *and* t_2*, hence it is a generalization of them, but it is not least general, neither is* $f(f(x,a),y)$ *an lgg. The unique lgg of* t_1 *and* t_2 *modulo* $\simeq_=$ *is the term* $f(f(x,a),x)$*. Figure 1.2 illustrates the two terms and their lgg. (Again,* $\simeq_=$ *is the concrete instance of* $\simeq_\equiv$*.)*

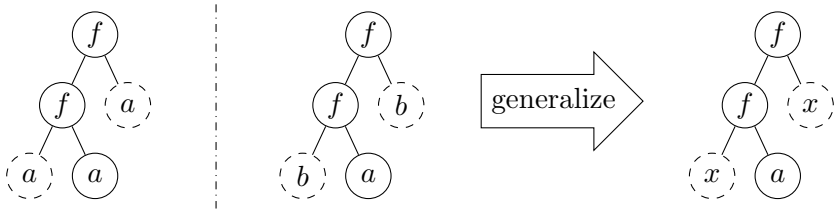

Figure 1.2: The terms from Example 1.1 and their lgg.

The anti-unification problem requires finding a least general term t for two input terms t_1 and t_2 such that $t_1 \equiv t\sigma_1$ and $t_2 \equiv t\sigma_2$ for some substitutions σ_1 and σ_2. The problem of finding a join corresponds to the *weak unification problem* which is formulated in the following way: Given two term t_1 and t_2, find two substitutions σ_1 and σ_2 such that $t_1\sigma_1 \equiv t_2\sigma_2$. In contrast to the anti-unification problem, its *dual* problem, the *unification problem*, aims at finding a substitution σ so that $t_1\sigma \equiv t_2\sigma$. If σ exists, then we say that t_1 and t_2 are *unifiable*, $t_1\sigma$ is a *common instance* of t_1 and t_2 modulo $\equiv$, and σ is called a *unifier* of t_1 and t_2. As already mentioned, the interesting generalizations are the least general ones. Conversely, a unification algorithm aims at finding the *most general common instance* of two terms. The most general common instance coincides with the join if it exists but the existence of a join does not imply unifiability. For instance, $f(a,x)$ and $f(x,b)$ are both smaller than $f(a,b)$, hence joinable, but they are not unifiable. In contrast to joinability, the variable names matter for unifiability of two terms. The *matching problem* aims at answering the question if a term t_1 is more general than t_2. A substitution σ so that $t_1\sigma \simeq_\equiv t_2$ is called a *matcher* of t_1 towards t_2. Notice that matching can be used to check whether t_1 and t_2 are equigeneral.

Theorem 1.2. *Let* $\mathcal{T}$ *be a meet-semilattice of terms w.r.t. the instantiation ordering and* $t_1, t_2 \in \mathcal{T}$*. If* t_1 *and* t_2 *do not share variables, then unifiability and the existence of a join are equivalent.*

Proof. Let t_1 and t_2 be variable disjoint terms. ($\Rightarrow$) Trivial. ($\Leftarrow$) Let t be a join of t_1 and t_2 such that it neither shares variables with t_1 nor with t_2. Since t, t_1 and t_2 are pairwise variable disjoint, there are substitutions σ_1 and σ_2 such that $t_1\sigma_1 \equiv t \equiv t_2\sigma_2$ and σ_1 maps all variables from t_2 to themselves (i.e., $t_2\sigma_1 = t_2$) and σ_2 maps all variables from t to themselves (i.e., $t\sigma_2 = t$). Therefore $t_2\sigma_1\sigma_2 \equiv t$ and $t_1\sigma_1\sigma_2 \equiv t$. □

Theorem 1.3. *Let* $t_1, t_2 \in \mathcal{T}(\mathcal{F},\mathcal{V})$*,* $f, g \in \mathcal{F}$ *and* $x \in \mathcal{V}$ *be a variable which occurs neither in* t_1 *nor in* t_2*. The terms* t_1 *and* t_2 *are unifiable iff* $f(g(t_1), g(t_2))$ *and* $f(x,x)$ *are unifiable.*

Proof. Let $t_1, t_2 \in \mathcal{T}(\mathcal{F}, \mathcal{V})$, $f, g \in \mathcal{F}$ and $x \in \mathcal{V}$ be a variable which occurs neither in t_1 nor in t_2. ($\Rightarrow$) There is σ so that $t_1\sigma \equiv t \equiv t_2\sigma$ and since x does not occur in t_1, t_2 we can choose $x\sigma = g(t)$. ($\Leftarrow$) From $g(t_1) \equiv x$ and $g(t_2) \equiv x$ follows that $g(t_1) \equiv g(t_2)$. □

From Theorem 1.3 it follows that the unification problem can be reduced to the search of the join. This corresponds to the notion of defining the anti-unification problem as the search of the meet.

We use the so called *generalization type* to classify the anti-unification problem for different theories. The types are defined by means of a minimal complete set of generalizations.

Definition 1.1. *A* complete set *of generalizations of two terms t_1 and t_2 is a set G of terms that satisfies the properties:*

Soundness: Each $t \in G$ is a generalization of both t_1 and t_2.

Completeness: For each generalization t of t_1 and t_2, there is $t' \in G$ such that $t \preceq_{\equiv} t'$.

Definition 1.2. *A* minimal complete set *of generalizations (mcg) of two terms t_1 and t_2 is a complete set G of terms that satisfies also the property:*

Minimality: For each $t, t' \in G$, if $t \preceq_{\equiv} t'$ then $t = t'$.

Definition 1.3. *A theory is of* generalization type

unitary *iff an mcg exists for all pairs of terms from the considered theory, and it has cardinality $\leqslant 1$,*

finitary *iff an mcg exists for all pairs of terms from the considered theory, and it has finite cardinality,*

infinitary *iff an mcg exists for all pairs of terms from the considered theory, and there exists a pair of terms within that theory for which the set is infinite,*

zero *iff there exists a pair of terms within the considered theory that does not have an mcg.*

Similarly to the generalization types, one can define unification types and matching types by means of a minimal complete set of unifiers and matchers, respectively.

To simplify the notation we write $\preceq$ instead of $\preceq_{\equiv}$ if the concrete instance of $\equiv$ is either unimportant or unmistakable by the context. For instance, if we talk about syntactic first-order term, then $\preceq$ denotes the instance $\preceq_{=}$. Similarly, when talking about an equational theory E then $\preceq$ denotes the instance $\preceq_E$. In the context of simply-typed lambda terms, the instance $\preceq_{=_\alpha}$ is denoted by $\preceq$, etc.

1.1 Related Work

The original motivation of introducing anti-unification was its application in automating induction. In an article, named "An Experiment in Automatic Induction" [72], Popplestone originated the idea that generalizations and least general generalizations of terms (literals in his work) exist. He mentioned that it would be useful when looking

for methods of induction. In 1970, Plotkin [71] and Reynolds [74] independently came up with an anti-unification algorithm that computes a unique least general generalization (modulo variable renaming) for two syntactic first-order terms. Both algorithms were formulated in an imperative style. In his work, Reynolds coined the term anti-unification. While Reynolds used the form $t_1 \succeq t_2$, Plotkin uses $t_1 \preceq t_2$ to indicate that the term (clause in his work) t_1 is more general than t_2. To justify his choice, he states that in the case of clauses, $\preceq$ is almost the same as $\subseteq$.

In his PhD thesis 1976, Huet [45] formulated an algorithm in terms of recursive equations. He uses a bijection ϕ from pairs of terms to variables. This bijection ensures that the generalizations he computes are least general. The algorithm is the following:

> Let ϕ be a bijection from pairs of terms to variables.
>
> Define a function λ, which maps pairs of terms to terms:

1. $\lambda(f(t_1, \ldots, t_n), f(s_1, \ldots, s_n)) = f(\lambda(t_1, s_1), \ldots, \lambda(t_n, s_n))$, for any f.
2. $\lambda(t, s) = \phi(t, s)$ otherwise.

Like Plotkin's and Reynolds' algorithm it is designed for first-order ranked terms (i.e., where function symbols have a fixed arity) in the syntactic case. Since then, a number of algorithms and their modifications have been developed, addressing the problem in various theories (e.g., [4, 9, 23, 45, 68]) and from the point of view of different applications (e.g., [6, 20, 30, 49, 57, 56, 77]).

Equational Anti-Unification. The idea that operations such as the addition or the multiplication of numbers are associative, commutative, and that they possess a unit element was implicitly assumed until the 19th century (see, e.g., [41]). Nowadays, classes of algebras (like groups, rings, etc.) are often defined by equations. For instance, the following equations stand for the axioms of associativity, commutativity, and unity:

$$\begin{array}{lll} f(x, f(y, z)) \approx f(f(x, y), z) & & \text{(associativity)} \\ f(x, y) \approx f(y, x) & & \text{(commutativity)} \\ f(x, e) \approx x \quad \text{and} & f(e, x) \approx x & \text{(unity)} \end{array}$$

where f is a binary function symbol and e is a constant, the unit element with respect to f. Generalizations that consider such axioms are called equational generalizations. For instance, under consideration of the commutativity axiom, the term $f(g(x, x), c)$ is an lgg of the two terms $f(g(a, a), c)$ and $f(c, g(b, b))$ while in the syntactic case $f(x, y)$ would be their lgg, where x and y are variables. Up until now, there are only a few theories where term equality modulo algebraic axioms has been considered for the anti-unification problem [2, 4, 9, 22, 23, 44, 73], and all of them are flavors of first-order settings.

Since we do not address anti-unification problems modulo equational theories in the present work, those results are only loosely related to our work.

The first work on equational generalization was presented by Pottier [73] and it appeared in 1989. He considers term equality modulo associativity and commutativity. In 1991, Baader [9] showed that commutative anti-unification is unitary. Burghardt [23] (2005) uses regular tree grammars to compute a finite representation of generalizations modulo some equational theory that represents a deductive closure of finitely

many ground equations. In Alpuente et al. [2] (2008) an anti-unification algorithm is presented that considers combinations of the axioms associativity, commutativity, and unity. Some years later, in 2014, the algorithm was adapted to the order-sorted theory [4]. It has been implemented in the Maude system (see, e.g., [29] as a Maude reference) and the source code is freely available for download.

Unranked Anti-Unification. Unranked terms are constructed from function symbols that do not have a fixed arity. For instance, $f(f(a), b, f)$ is an unranked term where the function f is applied to different numbers of arguments. Notice that, like common in the ranked case, we written a, b, f instead of $a(), b(), f()$ for zero argument applications. In the unranked setting, variables that can be instantiated by sequences of terms are usually considered. For example, in a term $f(\tilde{x})$ where $\tilde{x}$ is a sequence variable, $\tilde{x}$ might be instantiated by the sequence (a, b) leading to the term $f(a, b)$. For that reason, the term $f(\tilde{x}, b, \tilde{x})$ is a generalization[†] of the two terms $f(a, b, a)$ and $f(c, d, b, c, d)$. A hedge is a sequence of unranked terms, e.g., $(a, f(a, b), f)$ is a hedge.

In the year 2001, Yamamoto et al. came up with an algorithm that solves a special class of anti-unification problems for hedges, with the intention of using it for inductive reasoning of semi-structured documents such as HTML and XML [85]. During their work, it turned out that modeling semi-structured documents with first-order terms is inadequate. Therefore, they use a translation between semi-structured documents and hedges that has been proven useful in other contexts, e.g., [17, 61, 62]. The special class of hedges that is considered in their work forbids multiple occurrences of the same sequence variable. Such terms are called linear. Furthermore, for each subterm $f(s_1, \ldots, s_n)$ occurring in the hedge at any depth, there is at most one sequence variable among $(s_1, \ldots, s_n)$. Galitsky et al. [40] (2011) essentially describe a specialized variant of hedge anti-unification that finds commonalities between portions of text that is represented as a linguistic parse tree.

Word anti-unification is a special case of hedge anti-unification. A word can be considered as a hedge of constants, i.e., a hedge where all the applications have the empty argument hedge. Since word anti-unification is already highly nondeterministic, special classes of generalizations have been considered, e.g., in [18, 28].

Function symbols of feature terms do not have fixed arity, like in unranked terms. Nevertheless, they are fundamentally different to the concept of unranked terms and hedges. In feature terms each argument is denoted by a symbol identifier instead of position. The ordering of arguments does not matter, in contrast to unranked terms. Furthermore, sequence variables are not considered in the theory of feature terms. For this reasons, feature term anti-unification, as discussed in [1, 6, 70], is not comparable to unranked anti-unification.

Motivated by the fact that an universal algorithm that solves the unranked anti-unification problem would be useful in many different applications, such as detecting clones of software code, clustering of XML documents, detecting similarities and differences in them, etc., Kutsia et al. (2014) studied the first-order case of unranked anti-unification in [54]. Two different kinds of variables are used: Sequence variables

[†]Notice that, depending on the term algebra, it might not be an lgg because $f(\tilde{x}, b, \tilde{x}) \prec f(\tilde{x}, x, b, \tilde{x}, x)$ if x is a term variable and $\tilde{x}$ a sequence variable that can be instantiated by the empty sequence.

to fill in gaps in generalizations and term variables to abstract single subterms with different top function symbols. An algorithm is presented that computes a minimal complete set of unranked generalizations. Since the word anti-unification problem is already highly nondeterministic, the minimal complete hedge anti-unification algorithm is only of theoretical interest. To tackle this problem, they introduce the notion of a rigid generalization where consecutive sequence variables are forbidden and design an algorithm that is parametric by a skeleton computation function.

We will discuss that algorithm in more detail in chapter 2. Based on their idea of a skeleton computation function, we design an algorithm to solve tractable classes of the higher-order hedge anti-unification problem. To the best of our knowledge, up until now, there exists no work that addresses higher-order anti-unification for hedges or unranked terms. In the present work we also study higher-order anti-unification for simply-typed lambda terms. Since unranked terms are fundamentally different to typed calculus and all the existing work on higher-order anti-unification is rather related to our work on simply-typed lambda terms than the one on unranked terms, we discuss the work on higher-order anti-unification below.

Parts of that work have been developed in cooperation between Baumgartner and Kutsia and were presented in [12].

Nominal Anti-Unification. In 1999, Pitts and Gabbay [39] introduced nominal techniques to study first-order systems with bindings. In contrast to lambda calculus, nominal logic distinguishes between atoms that can be bound and variables which can be instantiated. The unification problem for nominal terms has been studied, for instance, in [55, 83]. To the best of our knowledge, anti-unification has not yet been studied for nominal terms. We address that shortcoming in our work and formulate an algorithm that solves first-order anti-unification for nominal terms.

Parts of that work have been developed in cooperation between Baumgartner, Kutsia, Levy, and Villaret and were presented in [15].

Higher-Order Anti-Unification. First-order generalization techniques are not suitable to detect similarities in terms that appear under different function applications. For instance, the terms $f(f(a,b),c)$ and $g(g(a,b),c)$ have a variable x as first-order lgg, while a higher-order variable, say F, would lead to a higher-order lgg $F(F(a,b),c)$. Some applications of anti-unification need to detect such similarities at the input examples. Higher-order features are requested, for instance, to reuse proofs in program verification [57]. A restricted variant of higher-order anti-unification was used for analogy making with Heuristic-Driven Theory Projection [49], and anti-unification with combinator terms tuned out to be useful in replaying program derivations [43], just to name a few.

Motivated by the problem of generalizing a given program, such that the class of problems it solves is enlarged, Hagiya [42] (1989) describes a procedure that computes a generalized program by extending the parametrization of an input program, so that it applies to a wider problem specification. It can be seen as a method to compute a higher-order generalization of a given program which serves as a skeleton of the computed output program. However, this approach is quite different from what we

consider the anti-unification problem because it only takes one input example and the generalization process is guided by the problem specification.

Two years later, in 1991, Pfenning presented an algorithm to solve the anti-unification problem for higher-order pattern in the Calculus of Constructions [68], with the intention of using it for proof generalization. Higher-order pattern are λ-terms where the arguments of free variables are distinct bound variables. Introduced by Miller [59] they gained popularity because they show nice computational behavior.

Since then, anti-unification algorithms have been developed to address the generalization problem within various restricted higher-order settings. Motivated by applications in inductive learning, Feng and Muggleton [34] (1992) presented an anti-unification algorithm in $M\lambda$ that can be seen as an extension of higher-order pattern.

While in [34, 68] higher-order anti-unification has been introduced on higher-order formulas with lambda abstraction, Furukawa et al. [36] (1996) developed an algorithm that essentially introduces function variables for first-order formulas by recursively applying first-order anti-unification. Later on, in the same year, they present an extension of that work in [66] which is based on Currying.

In contrast to Feng and Muggleton who consider a subset of the language $M\lambda$, Lu et al. [57] (2000) studied anti-unification in the language $\lambda 2$ (a second-order λ-calculus with type variables [11]), where they forbid the occurrence of abstractions inside arguments. This restriction guarantees uniqueness of the computed lgg. They give an algorithm and demonstrate its application in program verification to reuse proofs. Together with Pfenning's algorithm, it also influenced the generalization algorithm used in the program transformation technique called supercompilation [58].

Krumnack et al. [49] (2007) formulate another restricted variant of a higher-order anti-unification algorithm that they use in the context of analogy making.

In 2009, Pientka uses linear higher-order pattern anti-unification to develop a higher-order term indexing strategy based on substitution trees [69]. Linear higher-order patterns refine the notion of higher-order patterns from [59]. First, linear higher-order patterns require that every free variable occurs only once and in addition every free variable is applied to all distinct bound variables in its context. She presents an algorithm to insert terms into the index based on computing a linear higher-order pattern lgg of two linear higher-order pattern terms.

A more recent work related to higher-order anti-unification is that of Schmidt et al. from 2011 [78]. They present a restricted version of higher-order anti-unification which can be used to find structural commonalties and generate mappings between domains in the symbolic analogy model Heuristic-Driven Theory Projection.

In the present work we study higher-order anti-unification for unranked terms and for simply-typed lambda terms. We give proofs of correctness for both algorithms and prove upper bounds for their computational complexity. Work related to unranked anti-unification has already been discussed above, since unranked terms are fundamentally different to lambda terms. In contrast to Pfenning [68], who uses a higher-order typed calculus, we do not allow higher-order types. Due to the inherent complexity of the higher-order typed calculus, Pfenning was not able to formulate the algorithm in Huet's rule-based style [45]. The complexity has not been studied and the proofs of the algorithm's properties have been just sketched. Another difference is that, in contrast to Pfenning, we do not restrict the input terms to be higher-order patterns. Feng

and Muggleton [34] restrict their input to a certain extension of higher-order patterns in $M\lambda$. They do not give complexity analysis of their anti-unification algorithm nor the proofs of its properties. Pientka [69] uses linear higher-order pattern which is an additional restriction to higher-order pattern.

All the discussed approaches are quite different from ours, for similar reasons: First, we do not restrict the input terms, while all the other approaches do. Second, we prove all the properties and complexity bounds for the algorithm. Furthermore, our algorithm has been implemented and is freely available online.

Parts of that work have been developed in cooperation between Baumgartner, Kutsia, Levy, and Villaret and were presented in [14].

Application of Anti-Unification in Software Clone Detection. Generalization problems arise in many different research areas such as analogy making [20, 49, 57, 77], machine learning [6, 80], clone detection [20, 54, 56], data clustering [7, 35], etc. Therefore the problem has been addressed from the point of view of various applications.

The present work concentrates on studying the anti-unification problem in different term languages from a theoretical point of view and on the development of universally usable algorithms. To illustrate the developed algorithms we show how they can be used to detect software clones. Clone detection by anti-unification is not the main topic of this work and the algorithms should be useful ingredients to solve problems in various research areas. We only briefly discuss the topic of clone detection here and point the reader to more comprehensive works, like [75, 76].

Software clone detection became an active research topic in the past decades since clones are considered a serious problem for software maintainability. In the survey paper from Roy et al. a classification of software clones has been suggested. They define four different types of clones, namely Type-1, Type-2, Type-3, and Type-4. The hardness of detecting a clone increases by the number of the type. The suggested classification of the different clone types is the following:

Type-1: Identical code fragments except for variations in whitespace, layout and comments.

Type-2: Syntactically identical fragments except for variations in identifiers, literals, types, whitespace, layout and comments.

Type-3: Copied fragments with further modifications such as changed, added or removed statements, in addition to variations in identifiers, literals, types, whitespace, layout and comments.

Type-4: Two or more code fragments that perform the same computation but are implemented by different syntactic variants.

The types 1, 2, and 3 describe textual similarities while Type-4 describes the class of semantic clones. Note that the latter one does not subsume the other clone types. Some clone detection tools are based on translating the source code into an abstract syntax tree or into other hierarchically organized data structures. See, for instance, Baxter et al. [16], Evans et al. [33], Koschke et al. [48], Wahler et al. [84], and Yang [86]. Syntactic first-order anti-unification has been used for software clone detection by Bulychev et al. [20], Bulychev and Minea [21], and Li and Thompson [56]. They

essentially detect clones of Type-1 and Type-2. One disadvantage of syntactic first-order anti-unification is that it uses function symbols with fixed arity in the terms that represent the software code. For that reason it cannot detect clones where some statements have been removed from the source code or where some statements have been inserted after copying the code.

Kutsia et al. [54] and Yamamoto et al. [85] suggest to use unranked first-order anti-unification. It helps to detect some clones of Type-3, namely the ones that have been obtained by omitting/inserting some pieces of code. The transformation between abstract syntax trees and unranked terms is straightforward. Since structured documents, like XML of HTML, can also be represented by unranked trees, the method of unranked anti-unification can be used to compare structured documents, too.

However, since first-order anti-unification does not permit higher-order variables, it cannot locate similar code pieces which are located under distinct parent nodes or at different depths of the syntax tree. For instance, $f(a, b)$ and $g(h(a, b))$ are generalized by a single variable, although both terms contain a and b and a more natural generalization could be, e.g. $X(a, b)$, where X is a higher-order variable.

In the present work we address this shortcoming. We start by discussing the work from Kutsia et al. [54], illustrate the restrictions that naturally arise by first-order techniques on some examples from [75], and develop higher-order techniques that overcome those restrictions.

1.2 Coherence of Joinability, Matching, and Anti-Unification

We will present anti-unification algorithms as rule-based systems in the spirit of Alpuente et al. [3], using a store to keep track of differences in the input terms. More precisely, they will be described by transformation rules on triples of the form P; S; $\sigma \Longrightarrow P'$; S'; σ', where P is called the problem set, S is the store, and σ is a substitution which holds the generalization computed by successive rule application. The elements of the sets P and S are called the anti-unification equations (AUEs). P contains AUEs that have not been solved yet and S contains the already solved AUEs. For illustration, we reformulate Huet's algorithm into a rule-based system. We call the algorithm $\mathfrak{G}_1$, were the $\mathfrak{G}$ stands for generalization and the subscript 1 for first-order.

Definition 1.4. *An* anti-unification equation *is a triple* $x\colon t_1 \triangleq t_2$*, where* x *occurs neither in* t_1 *nor in* t_2*. The variable* x *is called the* generalization variable*. It stands for the most general generalization of* t_1 *and* t_2*.*

Definition 1.5. *The* anti-unification algorithm $\mathfrak{G}_1$ *works on tuples* P; S; σ*, where*

- *the problem set* P *is a set of AUEs to be solved,*
- *the store* S *is a set of already solved AUEs,*
- σ *is a substitution which "holds the generalization" computed so far*[†]*,*

[†]Let $\{x\colon t_1 \triangleq t_2\}$; $\varnothing$; $Id \Longrightarrow^* P$; S; σ be some derivation in $\mathfrak{G}_1$. Then $x\sigma \preceq t_1$ and $x\sigma \preceq t_2$. See Theorem 1.5, soundness of $\mathfrak{G}_1$.

- *for all pairs of AUEs* $\{x\colon t_1 \triangleq t_2,\ y\colon t_1' \triangleq t_2'\} \subseteq P \cup S$ *holds* $x \neq y$.

We call such a tuple a state. *The three transformation rules of the algorithm are defined below. They operate on states. The symbol* $\cup$ *stands for disjoint union.*

Dec: **Decomposition**

$\{x\colon f(t_1,\ldots,t_n) \triangleq f(t_1',\ldots,t_n')\} \cup P;\ S;\ \sigma \Longrightarrow$
$\{y_1\colon t_1 \triangleq t_1',\ldots,y_n : t_n \triangleq t_n'\} \cup P;\ S;\ \sigma\{x \mapsto f(y_1,\ldots,y_n)\}$,
where $y_1,\ldots,y_n$ are fresh variables, and $n \geqslant 0$.

Sol: **Solve**

$\{x\colon t \triangleq t'\} \cup P;\ S;\ \sigma \Longrightarrow P;\ \{x\colon t \triangleq t'\} \cup S;\ \sigma$,
if Dec is not applicable to $x\colon t \triangleq t'$.

Mer: **Merge**

$P;\ \{x\colon t \triangleq t', y\colon t \triangleq t'\} \cup S;\ \sigma \Longrightarrow P;\ \{x\colon t \triangleq t'\} \cup S;\ \sigma\{y \mapsto x\}$.

To compute a generalization of two input terms t_1 and t_2, the triple $P;\ S;\ \sigma$ has to be initialized. The procedure starts with the *initial state* $\{x\colon t_1 \triangleq t_2\};\ \varnothing;\ Id$ and applies the above rules as long as possible. The state where no more rule is applicable is called the *final state* and it is of the form $\varnothing;\ S';\ \sigma'$. One can show that $\mathfrak{G}_1$ is confluent and yields a unique final state (modulo $\simeq$) after an exhaustive rule application. We denote this unique final state by $\mathfrak{G}_1(x\colon t_1 \triangleq t_2)$, i.e., $\varnothing;\ S';\ \sigma' = \mathfrak{G}_1(x\colon t_1 \triangleq t_2)$. The lgg can be obtained from that final state by $x\sigma'$.

The idea of the store is to keep track of already solved AUEs in order to generalize the same AUEs in the same way, as it is illustrated in the Mer rule. The store contains all the differences of the two input terms.

Definition 1.6. *We define two substitutions obtained by a set* S *of AUEs:*

$$\sigma_{L}(S) ::= \{x \mapsto t_1 \mid x\colon t_1 \triangleq t_2 \in S\}$$
$$\sigma_{R}(S) ::= \{x \mapsto t_2 \mid x\colon t_1 \triangleq t_2 \in S\}$$

Given two terms t_1 and t_2, one can prove (see Theorem 1.5) that the store S' of a final state $\varnothing;\ S';\ \sigma' = \mathfrak{G}_1(x\colon t_1 \triangleq t_2)$ is sound in the sense that

$$x\sigma'\sigma_{\mathrm{L}}(S') = t_1 \text{ and } x\sigma'\sigma_{\mathrm{R}}(S') = t_2.$$

A *coherent* anti-unification algorithm computes a generalization term and two substitutions so that, when applied to the generalization term, one can obtain the two input terms, respectively. $\mathfrak{G}_1$ is an example for a coherent anti-unification algorithm.

Definition 1.7. *An anti-unification algorithm is* coherent *if it computes for two terms* t_1 *and* t_2 *an lgg* t *and two substitutions* σ_1 *and* σ_2 *such that* $t\sigma_1 \equiv t_1$ *and* $t\sigma_2 \equiv t_2$.

We illustrate the algorithm $\mathfrak{G}_1$ step by step on the two terms $t_1 = f(f(a,a),a)$ and $t_2 = f(f(b,a),b)$ from Example 1.1. Therefore, we start with the initial state $\{x\colon t_1 \triangleq t_2\};\ \varnothing;\ Id$ and exhaustively transform the state by applying the rules as long

as possible. For the sake of readability, we only keep track of the mapping for the generalization variable x in the substitution. We underline the AUE that is selected by the next rule application to emphasize the fact that $\mathfrak{G}_1$ is not deterministic. Notice that $\mathfrak{G}_1$ is confluent and yields a unique lgg regardless of the selection strategy.

$$
\begin{aligned}
& \{\underline{x\colon f(f(a,a),a) \triangleq f(f(b,a),b)}\};\ \varnothing;\ Id \\
\Longrightarrow_{\mathsf{Dec}}\ & \{\underline{y_1\colon f(a,a) \triangleq f(b,a)},\ y_2\colon a \triangleq b\};\ \varnothing;\ \{x \mapsto f(y_1,y_2)\} \\
\Longrightarrow_{\mathsf{Dec}}\ & \{\underline{y_2\colon a \triangleq b},\ y_3\colon a \triangleq b,\ y_4\colon a \triangleq a\};\ \varnothing;\ \{x \mapsto f(f(y_3,y_4),y_2)\} \\
\Longrightarrow_{\mathsf{Sol}}\ & \{\underline{y_3\colon a \triangleq b},\ y_4\colon a \triangleq a\};\ \{y_2\colon a \triangleq b\};\ \{x \mapsto f(f(y_3,y_4),y_2)\} \\
\Longrightarrow_{\mathsf{Sol}}\ & \{\underline{y_4\colon a \triangleq a}\};\ \{y_2\colon a \triangleq b,\ y_3\colon a \triangleq b\};\ \{x \mapsto f(f(y_3,y_4),y_2)\} \\
\Longrightarrow_{\mathsf{Dec}}\ & \varnothing;\ \{\underline{y_2\colon a \triangleq b},\ \underline{y_3\colon a \triangleq b}\};\ \{x \mapsto f(f(y_3,a),y_2)\} \\
\Longrightarrow_{\mathsf{Mer}}\ & \varnothing;\ \{y_2\colon a \triangleq b\};\ \{x \mapsto f(f(y_2,a),y_2)\}
\end{aligned}
$$

After applying Mer, there is no more rule that can be applied. This means that $\varnothing$; $\{y_2\colon a \triangleq b\}$; $\{x \mapsto f(f(y_2,a),y_2)\}$ is the final state and $f(f(y_2,a),y_2)$ the lgg of t_1 and t_2. From the store $S = \{y_2\colon a \triangleq b\}$ we get $\sigma_{\mathrm{L}}(S) = \{y_2 \mapsto a\}$ and $\sigma_{\mathrm{R}}(S) = \{y_2 \mapsto b\}$ so that $f(f(y_2,a),y_2)\sigma_{\mathrm{L}}(S) = t_1$ and $f(f(y_2,a),y_2)\sigma_{\mathrm{R}}(S) = t_2$.

To prove the correctness of $\mathfrak{G}_1$, one has to prove termination, soundness, completeness, and uniqueness. We do not prove those properties here but we state the theorems. We will prove the properties of algorithms that subsume the case of first-order anti-unification, e.g., one can use the algorithm from section 3.2.

Theorem 1.4 (Termination)**.** *The system $\mathfrak{G}_1$ terminates on any input.*

The soundness theorem states an invariant of the algorithm, namely that the substitution always "holds a generalization" of the input terms. Furthermore, it is stated in a way that it implies coherence of $\mathfrak{G}_1$.

Theorem 1.5 (Soundness)**.** *Let $\{x\colon t_1 \triangleq t_2\}$; $\varnothing$; $Id \Longrightarrow^* P$; S; σ be some derivation in $\mathfrak{G}_1$. Then $x\sigma\sigma_{\mathrm{L}}(S) = t_1$ and $x\sigma\sigma_{\mathrm{R}}(S) = t_2$.*

The completeness theorem implies that $\mathfrak{G}_1$ computes a complete set of lggs.

Theorem 1.6 (Completeness)**.** *Let t be a generalization of t_1 and t_2. Then there exists a derivation $\{x\colon t_1 \triangleq t_2\}$; $\varnothing$; $Id \Longrightarrow^* \varnothing$; S; σ obtained by $\mathfrak{G}_1$ such that $t \preceq x\sigma$.*

The uniqueness theorem says that $\mathfrak{G}_1$ computes the same result modulo $\simeq$ for each possible derivation.

Theorem 1.7 (Uniqueness modulo $\simeq$)**.** *Let $\varnothing$; S; $\sigma = \mathfrak{G}_1(x\colon t_1 \triangleq t_2)$ and $\varnothing$; S'; $\sigma' = \mathfrak{G}_1(x'\colon t_1 \triangleq t_2)$. Then $x\sigma \simeq x'\sigma'$.*

Notice that, in general, the final state is not unique. If an anti-unification algorithm computes a complete set of final states, then it should be minimized. This minimization step is usually done by matching. Nevertheless, there are term languages where it is not necessary to compute $t_1 \preceq_{\equiv} t_2$ and $t_2 \preceq_{\equiv} t_1$ in order to decide $t_1 \stackrel{?}{\preceq}_{\equiv} t_2$. For that case anti-unification can solve the matching problem:

Lemma 1.8. *Let $t, t_1, t_2 \in \mathcal{T}$ so that t is an lgg of t_1 and t_2. Then $t_1 \preceq_{\equiv} t_2$ iff $t \simeq_{\equiv} t_1$.*

Proof. Trivial by definition of an lgg. □

Corollary 1.9. *Let $t, t_1, t_2 \in \mathcal{T}$ so that t is an lgg of t_1 and t_2. Let σ_1, σ_2 be substitutions so that $t\sigma_1 \equiv t_1$ and $t\sigma_2 \equiv t_2$. If $t_1 \preceq t_2$ then $t_1\sigma_1^{-1}\sigma_2 \equiv t_2$.*

We discuss Corollary 1.9. It says that a coherent anti-unification algorithm computes a matcher σ_2 modulo $\simeq_{\equiv}$ if it exists. Depending on the term language and the anti-unification algorithm, the inverse substitution σ_1^{-1} might be easy to obtain from σ_1[†]. If so, then $\sigma_1^{-1}\sigma_2$ is the matcher computed by the anti-unification algorithm. We will use this property of a coherent anti-unification algorithm in the following chapters to minimize the complete set of generalization by the anti-unification algorithm itself.

Definition 1.8. *We say that a term is* linear *if each variables occurs only once.*

The following theorem tells us that we can use $\mathfrak{G}_1$ to obtain a join of two terms that are linear if it exists. Its proof yields a simple algorithm to compute a join that corresponds to a meet and two substitutions.

Theorem 1.10. *Let $\varnothing; S; \sigma = \mathfrak{G}_1(x\colon t_1 \triangleq t_2)$ be the final state computed by $\mathfrak{G}_1$ for two linear first-order terms t_1 and t_2 and a fresh generalization variable x. Then $\sigma_{\text{L}}(S)$, $\sigma_{\text{R}}(S)$, and $x\sigma$ yield an answer to the joinability problem of t_1 and t_2.*

Proof. Let $\varnothing; S; \sigma = \mathfrak{G}_1(x\colon t_1 \triangleq t_2)$ be the final state computed by $\mathfrak{G}_1$ for two linear first-order terms t_1 and t_2, and let $t = x\sigma$. Since x neither occurs in t_1 nor in t_2 and the variables introduced by the Dec rule are fresh, t is variable disjoint from t_1 and t_2. For the same reason, t_1 and t_2 do not contain variables from $\mathrm{Dom}(\sigma_{\text{L}}(S)) \cup \mathrm{Dom}(\sigma_{\text{R}}(S))$. Furthermore, it follows that $\mathrm{Ran}(\sigma_{\text{L}}(S))$ and $\mathrm{Ran}(\sigma_{\text{R}}(S))$ do not contain variables from $\mathrm{Dom}(\sigma_{\text{L}}(S)) \cup \mathrm{Dom}(\sigma_{\text{R}}(S))$.

Therefore, $\sigma_{\text{L}}(S)$ can be decomposed into $\sigma' = \{y \mapsto y\sigma_{\text{L}}(S) \mid y\sigma_{\text{L}}(S) \notin \mathcal{V}\}$ and σ'' such that $\sigma_{\text{L}}(S) = \sigma'\sigma''$. For all $\{y, z\} \subseteq \mathrm{Dom}(\sigma'')$ holds $y\sigma'' \neq z\sigma''$ because t_1 is linear. Similarly, we can decompose $\sigma_{\text{R}}(S)$ into $\vartheta'\vartheta''$. For that reason $t\sigma' \simeq t\sigma_{\text{L}}(S) = t_1$ and $t\vartheta' \simeq t\sigma_{\text{R}}(S) = t_2$. By $\simeq$ we get that t_1 and t_2 are joinable iff $t\sigma'$ and $t\vartheta'$ are joinable.

Case 1: The domains of σ' and ϑ' are disjoint. Since $\mathrm{Ran}(\sigma')$ and $\mathrm{Ran}(\vartheta')$ do not contain variables from $\mathrm{Dom}(\sigma') \cup \mathrm{Dom}(\vartheta')$, we get $\sigma'\vartheta' = \vartheta'\sigma'$ and $t\sigma'\vartheta'$ is an instance of both, t_1 and t_2. It is a most general instance because $t\sigma' \simeq t_1$ and $t\vartheta' \simeq t_2$.

Case 2: The domains of σ' and ϑ' are not disjoint. Then there exists $x' \in \mathcal{V}$ with $x'\sigma' = t_1'$ and $x'\vartheta' = t_2'$, so that $t_1' \not\preceq t_2'$ and $t_2' \not\preceq t_1'$ because otherwise there would be a generalization that is strictly less general than t, namely $t\{x' \mapsto t_1'\}$ or $t\{x' \mapsto t_2'\}$. It follows that both, t_1' and t_2', are of the form $f(s_1', \ldots, s_n')$ and the head symbol f is different for t_1' and t_2'. Otherwise, the generalization $t\{x' \mapsto f(y_1', \ldots, y_n')\}$ of t_1 and t_2 where y's are fresh variables would be strictly less general than t. A join does not exist in that case.

Putting this together, we get that $t\sigma'\vartheta'$ is a join of t_1 and t_2 if σ' and ϑ' have disjoint domains. Otherwise t_1 and t_2 are not joinable. □

Theorem 1.10 can be formulated for other coherent anti-unification algorithms, like the ones we discuss in the following chapters.

[†] σ_1 might be a bijection such that the inverse σ_1^{-1} can be read off easily.

Example 1.2. *We discuss the results on the coherent first-order anti-unification algorithm* $\mathfrak{G}_1$ *and the first-order terms* $t_1 = f(x_1, x_2)$, $t_2 = f(x_3, f(a, x_4))$, *and* $t_3 = f(b, f(x_5, a))$. *In the substitution we only keep track of the mapping for the generalization variable. We compute the following final states:*

$$\begin{aligned}
&\varnothing; \{z_1 \colon x_1 \triangleq x_3, z_2 \colon x_2 \triangleq f(a, x_4)\}; && \{y_1 \mapsto f(z_1, z_2)\} && = \mathfrak{G}_1(y_1 \colon t_1 \triangleq t_2)\\
&\varnothing; \{z_1 \colon x_1 \triangleq b,\ \ z_2 \colon x_2 \triangleq f(x_5, a)\}; && \{y_2 \mapsto f(z_1, z_2)\} && = \mathfrak{G}_1(y_2 \colon t_1 \triangleq t_3)\\
&\varnothing; \{z_1 \colon x_3 \triangleq b,\ \ z_2 \colon a \triangleq x_5,\ z_3 \colon x_4 \triangleq a\}; && \{y_3 \mapsto f(z_1, f(z_2, z_3))\} && = \mathfrak{G}_1(y_3 \colon t_2 \triangleq t_3)
\end{aligned}$$

In the first two cases we get a generalization $f(z_1, z_2)$ *that is equigeneral to* t_1. *Therefore* $t_1 \preceq t_2$ *and* $t_1 \preceq t_3$. *In the third case we get a generalization* $t = f(z_1, f(z_2, z_3))$ *and from the store* S *we obtain the substitutions* $\sigma_{_L}(S) = \{z_1 \mapsto x_3,\ z_2 \mapsto a,\ z_3 \mapsto x_4\}$ *and* $\sigma_{_R}(S) = \{z_1 \mapsto b,\ z_2 \mapsto x_5,\ z_3 \mapsto a\}$. *First, since* $t \not\preceq t_2$ *and* $t \not\preceq t_3$, *the terms* t_2 *and* t_3 *are incomparable with respect to* $\preceq$. *Now we split the substitutions* $\sigma_{_L}(S)$ *and* $\sigma_{_R}(S)$ *into two parts, respectively, like in the proof of Theorem 1.10:* $\sigma_{_L}(S) = \{z_2 \mapsto a\}$ $\{z_1 \mapsto x_3,\ z_3 \mapsto x_4\}$ *and* $\sigma_{_R}(S) = \{z_1 \mapsto b,\ z_3 \mapsto a\}\{z_2 \mapsto x_5\}$. *The domains of* $\{z_2 \mapsto a\}$ *and* $\{z_1 \mapsto b,\ z_3 \mapsto a\}$ *are disjoint, hence* t_2 *and* t_3 *are joinable,* t *is a skeleton of a join, and this join is* $t\{z_2 \mapsto a\}\{z_1 \mapsto b,\ z_3 \mapsto a\} = f(b, f(a, a))$.

We discussed some properties of coherent anti-unification algorithms in the general setting. This properties can be useful to minimize a complete set of generalizations to obtain the mcg if no matching algorithm exists. That was the main motivation behind this discussion. In section 2.2, we will use the anti-unification algorithm for higher-order unranked terms to minimize a complete set of generalizations it computes.

1.3 Structure of the Thesis

In Chapter 1 we introduce basic concepts and notions, and we discuss the work that is related to ours. This is an important chapter, since we use those general concepts and notions throughout the entire work. Chapter 2 deals with the anti-unification problem for unranked terms and hedges. Unranked terms are constructed from function symbols that do not have a fixed arity and hedges are sequences of unranked terms. First, we discuss and revise the unranked anti-unification problem in the first-order case that was introduced by Kutsia et al. [54]. Afterwards, we introduce higher-order power by allowing context-variables and function-variables. In Chapter 3 we study the anti-unification problem for ranked theories with binders. We consider two different term languages that involve binders: nominal terms and simply-typed lambda terms. We start discussing the problem for nominal terms which is a flavor of first-order logic with named binders. Then we turn to studying the anti-unification problem for simply-typed lambda terms. Since higher-order anti-unification is highly nondeterministic, we restrict the generalizations to be higher-order patterns. This are lambda terms where the arguments of free variables are distinct bound variables. In Chapter 4 we discuss our library of anti-unification algorithms which has been implemented in Java. We implemented four anti-unification algorithms for four different theories: unranked first-order anti-unification, unranked higher-order anti-unification, nominal anti-unification, and anti-unification for simply-typed lambda terms. Chapter 5 concludes this thesis and discusses some possible directions for future work.

Chapter 2

Anti-Unification for Unranked Terms and Hedges

In this chapter we discuss the anti-unification problem for *hedges*. Hedges are sequences of *unranked terms*, which are constructed from function symbols that do not have a fixed arity. Function symbols are denoted by the letters a, b, c, d, e, f, g, h. For instance, $f(f(a), b, f)$ is an unranked term and $(f(a), b, f(f(a), b, f))$ is a hedge. Unranked terms can be seen as ordered trees and hedges as forests, as illustrated in Figure 2.1. In addition to the unranked function symbols our grammars consider two kinds of first-order variables and two kinds of higher-order variables. The later ones can be instantiated by *contexts*. Contexts are hedges[†] with a single occurrence of the distinguished symbol "hole", denoted by $\circ$. They are functions which can apply to another context or to a hedge, which are then "plugged" in the place of the hole. The formal definitions can be found in the respective sections. Here follows an informal explanation of the different kinds of variables that are considered in this chapter:

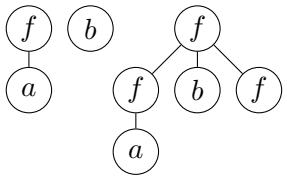

Figure 2.1: $(f(a), b, f(f(a), b, f))$

- *Term variables* correspond to the classical first-order variables that can be replaced by a term. We denote term variables by x, y, z. For instance, the term variable x in the hedge $(f(a), b, x, b))$ can be instantiated by a term $f(a, b)$, leading to the hedge $(f(a), b, f(a, b), b))$.
- *Sequence variables* (also called hedge variables), denoted by $\tilde{x}, \tilde{y}, \tilde{z}$, are first-order variables that can be instantiated by a hedge. For instance, the sequence variable $\tilde{x}$ in the term $f(\tilde{x})$ might be replaced by a hedge $(f(a), b, f)$. The result of that instantiation is the term $f(f(a), b, f)$.
- *Function variables* are higher-order variables. They appear in functional position and can be replaced by a *bounded context*. A bounded context is a term where the hole appears at level 1. For instance, $f(\circ, a)$ is a bounded context while $(a, f(\circ))$ and $f(f(\circ), a)$ are *not*. The letters F, G, H are used to denote function variables.
- *Context variables* are similar to function variables but they can be instantiated by arbitrary contexts. Context variables are denoted by $\tilde{X}, \tilde{Y}, \tilde{Z}$. For instance, consider the hedge $(f(a), \tilde{X}(a))$. The context variable $\tilde{X}$ can be instantiated by a context $(b, f(f(\circ), b, f))$, resulting in the hedge $(f(a), b, f(f(a), b, f))$.

[†]Notice that our contexts are hedges, while usually only terms are permitted.

Possible application areas of anti-unification for hedges are, for instance, analogy making [49], machine learning [6], program synthesis [47, 77], or clone detection [19, 21, 54, 56]. We choose the topic of software clone detection to illustrate our anti-unification algorithms on a possible application area.

As already discussed in the introduction, first-order hedge anti-unification algorithms can help to detect some clones of Type-3 but they fail to detect similarities that are located under distinct heads or at different depths. The reason is that they lack of higher-order power which would be needed to detect those similarities. Recall the example terms $f(a, b)$ and $g(h(a, b))$ who are generalized by a single variable.

We address this shortcoming here, permitting the use of function variables and context variables to gain higher-order power so that, e.g., $f(a, b)$ and $g(h(a, b))$ are generalized by a term $\tilde{X}(a, b)$, where $\tilde{X}$ is a context variable. The hedges $(f(a), f(b))$ and $(g(a), g(b))$ are generalized by a hedge $(F(a), F(b))$, where F is a function variable. A higher-order anti-unification algorithm is presented here and we demonstrate how it can be used to reveal software clones. We are able to detect similarities in clones of Type-3 and Type-4 which cannot be detected by first-order anti-unification. For this purpose we take some clone examples from Roy et al. [75] from the taxonomy of editing scenarios for different clone types. In fact, we will be able to find all the similarities between the original code and any clone of Type-3 from the suggested example. We will proceed in the following way to develop the theory of unranked higher-order anti-unification:

1. We discuss the work from Kutsia et al. [54] and show how one can use their anti-unification algorithm to solve matching problems of a certain kind. For simplicity, we only considering sequence variables for this first step.
2. We develop an anti-unification algorithm which generalizes the work from [54] by introducing context variables in addition to the hedge variables. The minimization step is done by the anti-unification algorithm itself.
3. We introduce term and function variables to obtain generalizations which are less general than those which only consider context and hedge variables.

Our goal is to obtain tractable classes of unranked higher-order anti-unification that are still accurate enough in practice. To obtain such classes of tractable higher-order generalizations all the algorithms that are considered in this chapter use the idea of a skeleton that guides the anti-unification process. Furthermore, some restrictions on the computed generalizations have to be introduced. For the sake of simplicity, we formulate the anti-unification algorithms for two hedges. The extension to more hedges is straightforward. Hedges to be generalized are assumed to be variable disjoint.

2.1 First-Order Unranked Anti-Unification

Here we discuss the work from Kutsia et al. [54] where the notion of *rigid generalization* has been introduced. That work deals with unranked *first-order* anti-unification. First, they give a rule-based algorithm to compute a minimal and complete set of generalizations (mcg) for two input hedges. Due to the high inherent complexity of hedge anti-unification, that universal algorithm is only of theoretical interest. For this reason, some requirements on the computed generalizations are imposed by inventing the

notion of a rigid generalization. Rigid generalizations are hedges where consecutive sequence variables are forbidden. Furthermore, rigid generalizations are classified by skeleton computation functions. Skeletons are defined recursively using a string of top symbols of a hedge. We give the necessary definitions now.

2.1.1 First-Order Unranked Terms and Hedges (Preliminaries)

Definition 2.1 (Terms and hedges)**.** *Given pairwise disjoint countable sets of unranked function symbols $\mathcal{F}$ (symbols without fixed arity) and hedge variables $\mathcal{V}_\mathsf{H}$, we define* terms *and* hedges *by the following grammar:*

$$\begin{array}{lcll} t & ::= & f(\tilde{s}) & \textit{(term)} \\ s & ::= & t \mid \tilde{x} & \textit{(hedge element)} \\ \tilde{s} & ::= & s_1, \ldots, s_n & \textit{(hedge)} \end{array}$$

where $f \in \mathcal{F}$, $\tilde{x} \in \mathcal{V}_\mathsf{H}$, and $n \geqslant 0$.

Hedges are finite sequences of terms and hedge variables, constructed over $\mathcal{F}$ and $\mathcal{V}_\mathsf{H}$. A term can be seen as a singleton hedge. The set of hedges constructed over $\mathcal{F}$ and $\mathcal{V}_\mathsf{H}$ is denoted by $\mathcal{T}(\mathcal{F}, \mathcal{V}_\mathsf{H})$, or simply by $\mathcal{T}$ if the concrete instances of $\mathcal{F}$ and $\mathcal{V}_\mathsf{H}$ are unimportant. To improve readability, we put none-singleton hedges between parenthesis. We denote function symbols by a, b, c, d, e, f, g, h, hedge variables by $\tilde{x}, \tilde{y}, \tilde{z}$, terms by t, s, u, q, r, and hedges by $\tilde{s}, \tilde{q}, \tilde{r}, \tilde{g}, \tilde{h}$. The empty hedge is denoted by ϵ and terms of the form $a(\epsilon)$ are written as just a.

Example 2.1. *For instance, $f(f(a), b)$ is a term as well as a singleton hedge and $(\tilde{x}, f(f(a), b))$ is a hedge.*

Definition 2.2 (Position)**.** *The set of* positions *of a hedge $\tilde{s} = (s_1, \ldots s_n)$, denoted* $\mathrm{Pos}(\tilde{s})$*, is a set of strings of positive integers. ϵ is the empty string and $\cdot$ stands for concatenation. It is defined as*

$$\mathrm{Pos}(\tilde{s}) ::= \bigcup_{i=1}^{n} \{i \cdot p \mid p \in \mathrm{Pos}_\mathsf{T}(s_i)\},$$

$$\mathrm{Pos}_\mathsf{T}(s) ::= \begin{cases} \{\epsilon\} & \textit{if } s = \tilde{x}, \\ \{\epsilon\} \cup \mathrm{Pos}(\tilde{q}) & \textit{if } s = f(\tilde{q}). \end{cases}$$

Note that $\mathrm{Pos}(\tilde{s})$ is a prefix closed set of strings.

Example 2.2. *For instance,* $\mathrm{Pos}(f(a, g(b, c))) = \{1, 1{\cdot}1, 1{\cdot}2, 1{\cdot}2{\cdot}1, 1{\cdot}2{\cdot}2\}$ *and* $\mathrm{Pos}(\tilde{x}, f(f(a), b), b) = \{1, 2, 2{\cdot}1, 2{\cdot}1{\cdot}1, 2{\cdot}2, 3\}$*. In the latter hedge, the term $f(a)$ stands at the position $2{\cdot}1$ and a occurs at the position $2{\cdot}1{\cdot}1$. Figure 2.2 illustrates that hedge and its positions.*

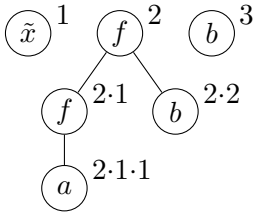

Figure 2.2: The hedge $(\tilde{x}, f(f(a), b), b)$ and its positions.

With $<$ we denote the strict *lexicographic ordering* and with $\sqsubset$ the strict *prefix relation* on positions, e.g., $1{\cdot}2{\cdot}1 < 1{\cdot}2{\cdot}2$, $1{\cdot}2{\cdot}1 < 1{\cdot}2{\cdot}1{\cdot}2$, and $1{\cdot}2{\cdot}1 \sqsubset 1{\cdot}2{\cdot}1{\cdot}2$. The relation $\sqsubseteq$ is defined as $\sqsubset \cup =$.

Definition 2.3 (Length and size)**.** *The* cardinality *of a set A is denoted by $|A|$. Similarly, the* length *of a hedge $\tilde{s} = (s_1, \dots s_n)$ is denoted by $|\tilde{s}|$ and defined as the number n, that is the number of elements in it. We define the* size *$\|\tilde{s}\|$ of a hedge $\tilde{s}$ by the cardinality of the set of positions $|\mathrm{Pos}(\tilde{s})|$, that is the number of all occurrences of function symbols and variables in it.*

Example 2.3. *For instance, $|(\tilde{x}, f(f(a), b), b)| = 3$ and $\|(\tilde{x}, f(f(a), b), b)\| = 6$.*

We denote by $\tilde{s}|_i$ the ith element of $\tilde{s}$, where $1 \leqslant i \leqslant |\tilde{s}|$. For arbitrary integers i, j and a hedge $\tilde{s}$, we define

$$\tilde{s}|_i^j ::= \begin{cases} (\tilde{s}|_i, \dots, \tilde{s}|_j) & \text{if } 1 \leqslant i \leqslant j \leqslant |\tilde{s}|, \\ \epsilon & \text{otherwise.} \end{cases}$$

Note that $\tilde{s}|_i^i = \tilde{s}|_i$. The notion of $\tilde{s}|_i$ is extended to arbitrary positions such that $\tilde{s}|_I$ denotes the subterm at position I, for all $I \in \mathrm{Pos}(\tilde{s})$.

Example 2.4. *Consider $\tilde{s} = (\tilde{x}, f(f(a), b), b)$. Then $\tilde{s}|_2 = f(f(a), b)$, $\tilde{s}|_{2\cdot 1} = f(a)$, $\tilde{s}|_2^3 = (f(f(a), b), b)$, and $\tilde{s}|_3^2 = \epsilon$.*

Definition 2.4 (Top symbols of a hedge)**.** *The top symbol of a term is defined as $\mathrm{Top}(\tilde{x}) = \tilde{x}$ for any variable $\tilde{x}$, and $\mathrm{Top}(f(\tilde{s})) = f$ for any term $f(\tilde{s})$. This notion is extended to hedges, as the following sequence of symbols: $\mathrm{Top}(\tilde{x}, \tilde{s}) = \tilde{x}\mathrm{Top}(\tilde{s})$ and $\mathrm{Top}(t, \tilde{s}) = \mathrm{Top}(t)\mathrm{Top}(\tilde{s})$ for any hedge variable $\tilde{x}$, term t, and hedge $\tilde{s}$.*

Such a string of top symbols of a hedge is written as a word, denoted by w. This definition also allows us to address the symbol at a certain position inside of a hedge. Given a symbol or variable s and a hedge $\tilde{s}$ we write $\mathrm{Pos}_s(\tilde{s})$ for the set $\{I \mid \mathrm{Top}(\tilde{s}|_I) = s \text{ and } I \in \mathrm{Pos}(\tilde{s})\}$ of all occurrences of s in $\tilde{s}$.

Example 2.5. *We demonstrate the defined notation on the hedge $\tilde{s} = (\tilde{x}, f(f(a), b), b)$. $\mathrm{Top}(\tilde{s}) = \tilde{x}fb$ and $\mathrm{Top}(\tilde{s}|_2) = \mathrm{Top}(\tilde{s}|_{2\cdot 1}) = f$. Furthermore, $\mathrm{Pos}_f(\tilde{s}) = \{2, 2\cdot 1\}$ and $\mathrm{Pos}_b(\tilde{s}) = \{2\cdot 2, 3\}$.*

Definition 2.5 (Substitution)**.** *A* substitution *is a mapping $\sigma : \mathcal{V}_\mathsf{H} \to \mathcal{T}(\mathcal{F}, \mathcal{V}_\mathsf{H})$ from hedge variables to hedges that is identity almost everywhere.*

The symbols $\sigma, \vartheta, \varphi$ are used to denote a substitution. Any substitution σ can be extended to a mapping $\hat{\sigma} : \mathcal{T} \to \mathcal{T}$. We use postfix notation for application, writing, e.g., $\tilde{s}\hat{\sigma}$ for the application of $\hat{\sigma}$ to $\tilde{s}$. The application of $\hat{\sigma}$ to $\tilde{s}$ is defined by induction on the structure of terms and hedges:

$$\tilde{s}\hat{\sigma} ::= \begin{cases} f(\tilde{q}\hat{\sigma}) & \text{if } \tilde{s} = f(\tilde{q}), \\ \sigma(\tilde{x}) & \text{if } \tilde{s} = \tilde{x}, \\ s_1\hat{\sigma}, \dots, s_n\hat{\sigma} & \text{if } |\tilde{s}| \neq 1 \text{ and } \tilde{s} = (s_1, \dots, s_n). \end{cases}$$

For a substitution σ the *domain* is the set of variables

$$\mathrm{Dom}(\sigma) ::= \{\tilde{x} \in \mathcal{V}_\mathsf{H} \mid \sigma(\tilde{x}) \neq \tilde{x}\}$$

and the *range* is the set of hedges

$$\mathrm{Ran}(\sigma) ::= \{\sigma(\tilde{x}) \mid \tilde{x} \in \mathrm{Dom}(\sigma)\}.$$

The *composition* of two substitutions σ_1 and σ_2 is written as $\sigma_1\sigma_2$ and is defined by $t\hat{\sigma}_1\hat{\sigma}_2 = (t\hat{\sigma}_1)\hat{\sigma}_2$ for any $t \in \mathcal{T}$. The corresponding mapping may be obtained by $\sigma_1\sigma_2 = \{\tilde{x} \mapsto \sigma_1(\tilde{x})\hat{\sigma}_2 \mid \tilde{x} \in \mathrm{Dom}(\sigma_1)\} \cup \{\tilde{x} \mapsto \sigma_2(\tilde{x}) \mid \tilde{x} \in \mathrm{Dom}(\sigma_2) \text{ and } \tilde{x} \notin \mathrm{Dom}(\sigma_1)\}$.

To simplify the notation, we do not distinguish between a substitution σ and its extension $\hat{\sigma}$. Notice that in Example 2.6 we would need to say that $(f(\tilde{x}), \tilde{y}, f(\tilde{y}), a)\hat{\sigma} = (f, a, \tilde{x}, f(a, \tilde{x}), a)$ without that simplification.

Example 2.6. *Let $\sigma = \{\tilde{x} \mapsto \epsilon,\ \tilde{y} \mapsto (a, \tilde{x})\}$ be a substitution, then $(f(\tilde{x}), \tilde{y}, f(\tilde{y}), a)\sigma = (f, a, \tilde{x}, f(a, \tilde{x}), a)$.*

Definition 2.6 (Renaming)**.** *A substitution is called* renaming *if $Ran(\sigma) \subseteq \mathcal{V}_\mathsf{H}$ and $|Dom(\sigma)| = |Ran(\sigma)|$.*

In other words, a renaming σ is a mapping from variables to variables. It is a bijection from $\mathrm{Dom}(\sigma)$ to $\mathrm{Ran}(\sigma)$.

Definition 2.7 (Instantiation)**.** *A hedge $\tilde{s}$ is the* instance *of a hedge $\tilde{q}$ if there exists a substitution σ with $\tilde{q}\sigma = \tilde{s}$. We say that $\tilde{q}$ is more general than $\tilde{s}$ if $\tilde{s}$ is an instance of $\tilde{q}$ and denote this by $\tilde{q} \preceq \tilde{s}$. If $\tilde{q} \preceq \tilde{s}$ and $\tilde{s} \preceq \tilde{q}$, then we write $\tilde{q} \simeq \tilde{s}$. If $\tilde{q} \preceq \tilde{s}$ and $\tilde{q} \not\simeq \tilde{s}$, then we say that $\tilde{q}$ is strictly more general than $\tilde{s}$ and write $\tilde{q} \prec \tilde{s}$.*

Definition 2.8 (Generalization)**.** *A hedge $\tilde{g}$ is a* generalization *of the hedges $\tilde{s}$ and $\tilde{q}$ if $\tilde{s}$ and $\tilde{q}$ are instances of $\tilde{g}$.*

Example 2.7. *The hedge $(f(a, \tilde{x}), c, \tilde{x})$ is a generalization of two hedges $(f(a, b, b), c, b, b)$ and $(f(a, d), c, d)$. Dashed nodes indicate differences, while the solid ones form the skeleton. The first hedge can be obtained from the generalization by replacing the hedge variable $\tilde{x}$ with the hedge (b, b). To obtain the second hedge, we need to replace $\tilde{x}$ by d.*

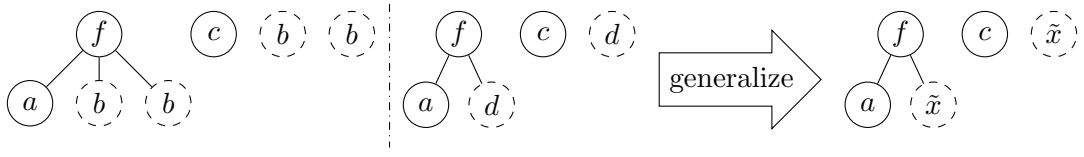

Figure 2.3: The hedges from Example 2.7 and their generalization.

The word representation of a hedge is defined by the concatenation of the depth-first pre-order traversal of the constituent terms. For instance, $a\tilde{x}fgagbb$ is the word representation of $(a, \tilde{x}, f(g(a, g(b, b))))$. Generalizations contain a common subsequence of the word representation of the input hedges. This property is used to compute a skeleton that guides the anti-unification algorithm. Observe, e.g., the hedges from Figure 2.3:

$$\begin{array}{l} (\mathbf{f}(\mathbf{a}, b, b), \mathbf{c}, b, b) \\ \underline{(\mathbf{f}(\mathbf{a}, d\ \ ), \mathbf{c}, d\ \ )} \\ (\mathbf{f}(\mathbf{a}, \tilde{x}\ \ ), \mathbf{c}, \tilde{x}\ \ ) \end{array}$$

Lemma 2.1. *Let $\tilde{s}, \tilde{q} \in \mathcal{T}$ and $f \in \mathcal{F}$. $\tilde{s} \preceq \tilde{q}$ iff $f(\tilde{s}) \preceq f(\tilde{q})$.*

Proof. Trivial, since $f(\tilde{s})\sigma = f(\tilde{s}\sigma)$ for any $f \in \mathcal{F}$ and any hedge $\tilde{s}$. □

Corollary 2.2. *A hedge $\tilde{g}$ is a generalization (respectively, a common instance) of two hedges $\tilde{s}$ and $\tilde{q}$ iff $f(\tilde{g})$ is a generalization (respectively, a common instance) of the two terms $f(\tilde{s})$ and $f(\tilde{q})$ for arbitrary $f \in \mathcal{F}$.*

Definition 2.9 (Consecutive)**.** *Two occurrences of symbols s_1, s_2 of a hedge are (horizontal)* consecutive *if the corresponding positions $I_{s_1}{\cdot}i_{s_1}$ and $I_{s_2}{\cdot}i_{s_2}$ are in the relation $I_{s_1} = I_{s_2}$ and $i_{s_1} + 1 = i_{s_2}$.*

Definition 2.10 (Vertical chain)**.** *Two occurrences of symbols s_1, s_2 of a hedge $\tilde{s}$ are in a* vertical chain *if their positions I_{s_1} and I_{s_2} are in the relation $I_{s_1}{\cdot}1 = I_{s_2}$ and $I_{s_1}{\cdot}2 \notin \mathrm{Pos}(\tilde{s})$.*

Example 2.8. *For example, in $\tilde{s} = (a, f(g(\tilde{x}, b)))$, the occurrence of a at position 1 and the occurrence of f at 2 are consecutive, as well as $\tilde{x}$ at $2{\cdot}1{\cdot}1$ and b at $2{\cdot}1{\cdot}2$. The occurrence of f at 2 and the occurrence of g at $2{\cdot}1$ are in vertical chain because $2{\cdot}2 \notin \mathrm{Pos}(\tilde{s}) = \{1, 2, 2{\cdot}1, 2{\cdot}1{\cdot}1, 2{\cdot}1{\cdot}2\}$, while the occurrence of g at $2{\cdot}1$ and the occurrence of $\tilde{x}$ at $2{\cdot}1{\cdot}1$ are* not *because $2{\cdot}1{\cdot}2 \in \mathrm{Pos}(\tilde{s})$.*

Intuitively, two occurrences of symbols s_1, s_2 are in a vertical chain, if s_1 is applied to a *term* $s_2(\tilde{q})$ where $\tilde{q}$ is an arbitrary hedge. If the symbol s_1 is applied to a hedge like $(\tilde{q}_1, s_2(\tilde{q}), \tilde{q}_2)$ with $|\tilde{q}_1, s_2(\tilde{q}), \tilde{q}_2| > 1$, then it is *not* considered a vertical chain, since it can be seen as a branching in the tree representation of the term.

Theorem 2.3. *First-order unranked anti-unification is finitary: For any hedges $\tilde{s}$ and $\tilde{q}$ there exists their minimal complete set of generalizations. This set is finite and unique modulo $\simeq$.*

Proof. The proof can be found in [54]. We will prove the more general statement that higher-order unranked anti-unification is finitary later in Theorem 2.11. □

In order to compute those "less obvious" generalizations, the complete hedge anti-unification algorithm from [54] might produce up to 3^n generalizations, where n is the size of the input. For instance, the algorithm generates 33 generalization to compute the mcg from Example 2.9. Furthermore, a hedge-matching algorithm is needed to minimize the set of all the computed generalizations. Such algorithms are known to be NP-complete (see e.g., [51, 53]). Therefore, this algorithm is only of theoretical interest and there is not much hope to compute a minimal and complete set of *higher-order* generalizations for two arbitrary input hedges in reasonable time. For that reason we concentrate on the class of rigid generalizations in the present work.

Example 2.9. *The set of hedges $\{(f(a), f(\tilde{x})), (f(\tilde{x}, \tilde{y}), f(\tilde{x})), (f(\tilde{x}, \tilde{y}), f(\tilde{y}))\}$ is the mcg of the hedges $(f(a), f(a))$ and $(f(a), f)$. Besides the "obvious" generalization $(f(a), f(\tilde{x}))$ it contains two other generalizations. All of them are pairwise incomparable with respect to $\preceq$ but only the hedge $(f(a), f(\tilde{x}))$ fulfills the requirements of a rigid generalization.*

Definition 2.11 (Alignment)**.** *Let w_1 and w_2 be strings of symbols. Then the sequence $a_1\langle i_1, j_1\rangle \cdots a_n\langle i_n, j_n\rangle$, for $n \geqslant 0$, is an* alignment *if*

- *i's and j's are positive integers such that $0 < i_1 < \cdots < i_n \leqslant |w_1|$ and $0 < j_1 < \cdots < j_n \leqslant |w_2|$, and*
- *$a_k = w_1|_{i_k} = w_2|_{j_k}$, for all $1 \leqslant k \leqslant n$.*

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

Example 2.10. *Let $\mathcal{R}$ return the set of all longest common subsequences of two strings whose length is at least 3:*

- $\mathcal{R}(ab, abc) = \mathcal{R}(abc, dd) = \varnothing$.
- $\mathcal{R}(abcda, bcad) = \{b\langle 2,1\rangle c\langle 3,2\rangle a\langle 5,3\rangle,\ b\langle 2,1\rangle c\langle 3,2\rangle d\langle 4,4\rangle\}$.

2.1.2 First-Order Unranked Anti-Unification Algorithm $\mathfrak{G}_{\mathcal{R}}$

In this subsection we discuss the central definition of a (rigid) $\mathcal{R}$-generalization and show the anti-unification algorithm from [54] that computes such generalizations for two input hedges. We will also show some examples to illustrate the algorithm.

Definition 2.12 ($\mathcal{R}$-generalization)**.** *Given two variable-disjoint hedges $\tilde{s}$ and $\tilde{q}$ and the rigidity function $\mathcal{R}$, we say that a hedge $\tilde{g}$ that generalizes both $\tilde{s}$ and $\tilde{q}$ is their (rigid) $\mathcal{R}$-*generalization*, if either $\mathcal{R}(\mathrm{Top}(\tilde{s}), \mathrm{Top}(\tilde{q})) = \varnothing$ and $\tilde{g}$ is a hedge variable, or there exists an alignment $a_1\langle i_1, j_1\rangle \cdots a_n\langle i_n, j_n\rangle \in \mathcal{R}(\mathrm{Top}(\tilde{s}), \mathrm{Top}(\tilde{q}))$ such that the following conditions are fulfilled:*

1. *The sequence $\tilde{g}$ does not contain pairs of consecutive hedge variables.*
2. *If we remove all hedge variables that occur as elements of $\tilde{g}$, we get a sequence of the form $a_1(\tilde{g}_1), \ldots, a_n(\tilde{g}_n)$.*
3. *For every $1 \leqslant k \leqslant n$, there exists a pair of sequences $\tilde{s}_k$ and $\tilde{q}_k$ such that $\tilde{s}|_{i_k} = a_k(\tilde{s}_k)$, $\tilde{q}|_{j_k} = a_k(\tilde{q}_k)$ and $\tilde{g}_k$ is an $\mathcal{R}$-generalization of $\tilde{s}_k$ and $\tilde{q}_k$.*

Since this definition forbids consecutive hedge variables, the hedges $(f(\tilde{x}, \tilde{y}), f(\tilde{x}))$ and $(f(\tilde{x}, \tilde{y}), f(\tilde{y}))$ from Example 2.9 are *not rigid* generalizations.

The intuition behind the recursion defined by item 3 of Definition 2.12 is to recursively compute skeletons by applying a rigidity function $\mathcal{R}$ to two strings of top function symbols for two hedges, while decomposing and going deeper into the terms. The skeletons are computed in a level-by-level manner for the given input hedges.

Item 2 of Definition 2.12 guarantees that a generalization contains exactly those function symbols which correspond to a recursively computed skeleton. No additional function symbols are allowed. (Allowing such extra symbols does not fit to the idea of a skeleton.)

Based on that definition, a rule-based anti-unification algorithm that solves the following problem has been developed in [54]:

Given: Two hedges $\tilde{s}$ and $\tilde{q}$ and a rigidity function $\mathcal{R}$.

Find: A minimal complete set of $\mathcal{R}$-generalizations.

The algorithm uses a data structure that consists of a variable and two hedges:

Definition 2.13 (Anti-unification equation)**.** *An* anti-unification equation*, AUE in short, is a triple* $\tilde{x}\colon \tilde{s} \triangleq \tilde{q}$*, where* $\tilde{x}$ *does not occur in* $\tilde{s}$ *and* $\tilde{q}$*. Intuitively,* $\tilde{x}$ *is a variable that stands for the most general generalization of* $\tilde{s}$ *and* $\tilde{q}$*.*

The algorithm is parametric by the rigidity function and consists of four transformation rules that transform tuples by rule application into tuples of the same form.

Definition 2.14 ($\mathcal{R}$-anti-unification algorithm)**.** *The rigid unranked first-order anti-unification algorithm is formulated in a rule-based way working on tuples* P; S; σ *and a global parameter* $\mathcal{R}$*, where*

- $\mathcal{R}$ *is a rigidity function;*
- P *and* S *are sets of AUEs such that if* $\tilde{x}\colon \tilde{s} \triangleq \tilde{q} \in P \cup S$*, then this is the sole occurrence of* $\tilde{x}$ *in* $P \cup S$*;*
- P *is the set of AUEs to be solved (the problem set);*
- S *is a set of already solved AUEs (the store);*
- σ *is a substitution (computed so far) mapping hedge variables to hedges.*

We call such a tuple a state *and the algorithm is called* $\mathfrak{G}_{\mathcal{R}}$*, where* $\mathcal{R}$ *indicates the rigidity function parameterizing the algorithm. The four transformation rules of the algorithm, which are defined below, operate on states.*

In the transformation rules, we use the symbols $\tilde{y}, \tilde{z}$ for fresh hedge variables and the symbol $\cup$ stands for disjoint union. Furthermore, i^{--} denotes $i-1$ and i^{++} denotes $i+1$. Note that we use other definition of $\tilde{s}|_i^j$ than Kutsia et al. [54].

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

$\{\tilde{x}\colon \tilde{s} \triangleq \tilde{q}\} \cup P;\ S;\ \sigma \Longrightarrow$
$\{\tilde{z}_k\colon \tilde{s}_k \triangleq \tilde{q}_k \mid 1 \leqslant k \leqslant n\} \cup P;$
$\{\tilde{y}_0\colon \tilde{s}|_1^{i_1^{--}} \triangleq \tilde{q}|_1^{j_1^{--}}\} \cup \{\tilde{y}_k\colon \tilde{s}|_{i_k^{++}}^{i_{k+1}^{--}} \triangleq \tilde{q}|_{j_k^{++}}^{j_{k+1}^{--}} \mid 1 \leqslant k \leqslant n-1\} \cup \{\tilde{y}_n\colon \tilde{s}|_{i_n^{++}}^{|\tilde{s}|} \triangleq \tilde{q}|_{j_n^{++}}^{|\tilde{q}|}\} \cup S;$
$\sigma\{\tilde{x} \mapsto (\tilde{y}_0, a_1(\tilde{z}_1), \tilde{y}_1, \ldots, \tilde{y}_{n-1}, a_n(\tilde{z}_n), \tilde{y}_n)\},$

if $\mathcal{R}(\mathrm{Top}(\tilde{s}), \mathrm{Top}(\tilde{q}))$ contains a sequence $a_1\langle i_1, j_1\rangle \cdots a_n\langle i_n, j_n\rangle$ such that for all $1 \leqslant k \leqslant n$, $\tilde{s}_{i_k} = a_k(\tilde{s}_k)$, $\tilde{q}_{i_k} = a_k(\tilde{q}_k)$.

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

$\{\tilde{x}\colon \tilde{s} \triangleq \tilde{q}\} \cup P;\ S;\ \sigma \Longrightarrow P;\ \{\tilde{x}\colon \tilde{s} \triangleq \tilde{q}\} \cup S;\ \sigma,$

if $\mathcal{R}(\mathrm{Top}(\tilde{s}), \mathrm{Top}(\tilde{q})) = \varnothing$. (Notice that this transformation is equivalent to rule $\mathcal{R}$-Dec-H where $\mathcal{R}(\mathrm{Top}(\tilde{s}), \mathrm{Top}(\tilde{q})) = \{\mathfrak{e}\}$.)

$\mathcal{R}$-Mer-S: **$\mathcal{R}$-Rigid Merge Store**

$P;\ \{\tilde{x}_1\colon \tilde{s} \triangleq \tilde{q},\ \tilde{x}_2\colon \tilde{s} \triangleq \tilde{q}\} \cup S;\ \sigma \Longrightarrow P;\ \{\tilde{x}_1\colon \tilde{s} \triangleq \tilde{q}\} \cup S;\ \sigma\{\tilde{x}_2 \mapsto \tilde{x}_1\},$

if $\tilde{x}_1 \neq \tilde{x}_2$.

$\mathcal{R}$-Clr-S: **$\mathcal{R}$-Rigid Clear Store**

$P;\ \{\tilde{x}\colon \epsilon \triangleq \epsilon\} \cup S;\ \sigma \Longrightarrow P;\ S;\ \sigma\{\tilde{x} \mapsto \epsilon\}.$

To compute $\mathcal{R}$-generalizations of $\tilde{s}$ and $\tilde{q}$, we start with the *initial state* $\{\tilde{x}\colon \tilde{s} \triangleq \tilde{q}\}; \varnothing; Id$ and apply the rules on the selected anti-unification equation(s) in all possible ways. Rules are applied exhaustively until no more rule is applicable to a certain state.

Such a state is called *final state*. (Note that in a final state the problem set is empty.) Since the algorithm is proven to be terminating, sound, and complete, the result will be a finite set of final states that corresponds to a complete set of generalizations with respect to $\mathcal{R}$. (The proofs can be found in [54].) We denote that set of final states by $\mathfrak{G}_{\mathcal{R}}(\tilde{x}\colon \tilde{s} \triangleq \tilde{q})$. The generalization that corresponds to a final state $\varnothing; S; \sigma \in \mathfrak{G}_{\mathcal{R}}(\tilde{x}\colon \tilde{s} \triangleq \tilde{q})$ can be obtained by $\tilde{x}\sigma$.

To compute a *minimal* complete set of $\mathcal{R}$-generalizations, a minimization step has to be performed. This involves a matchability test of two hedges, for we only keep the least general ones. To perform the minimization step we can use, for instance, the hedge matching algorithm from [51, 52].

2.1.3 Illustration of the Algorithm $\mathfrak{G}_{\mathcal{R}}$

Example 2.11. *Let $\tilde{s} = f(a, f(b, b))$ and $\tilde{q} = (b, f(a, b), b)$ be the input hedges and $\mathcal{R}$ be the function computing the set of all longest common subsequences. We illustrate how the algorithm $\mathfrak{G}_{\mathcal{R}}$ exhaustively transforms the initial system to compute the rigid lgg for $\tilde{s}$, $\tilde{q}$, and $\mathcal{R}$. In the substitution, we only keep track of the mapping for the generalization variables $\tilde{x}$ of the initial AUE.*

$$
\begin{aligned}
& \{\tilde{x}\colon f(a, f(b, b)) \triangleq (b, f(a, b), b)\};\ \varnothing;\ Id \\
\Longrightarrow^{f\langle 1,2\rangle}_{\mathcal{R}\text{-Dec-H}} \quad & \{\tilde{z}_1\colon (a, f(b, b)) \triangleq (a, b)\};\ \{\tilde{y}_1\colon \epsilon \triangleq b,\ \tilde{y}_2\colon \epsilon \triangleq b\};\ \{\tilde{x} \mapsto (\tilde{y}_1, f(\tilde{z}_1), \tilde{y}_2)\} \\
\Longrightarrow_{\mathcal{R}\text{-Mer-S}} \quad & \{\tilde{z}_1\colon (a, f(b, b)) \triangleq (a, b)\};\ \{\tilde{y}_1\colon \epsilon \triangleq b\};\ \{\tilde{x} \mapsto (\tilde{y}_1, f(\tilde{z}_1), \tilde{y}_1)\} \\
\Longrightarrow^{a\langle 1,1\rangle}_{\mathcal{R}\text{-Dec-H}} \quad & \{\tilde{z}_2\colon \epsilon \triangleq \epsilon\};\ \{\tilde{y}_1\colon \epsilon \triangleq b,\ \tilde{y}_3\colon f(b, b) \triangleq b\};\ \{\tilde{x} \mapsto (\tilde{y}_1, f(a(\tilde{z}_2), \tilde{y}_3), \tilde{y}_1)\} \\
\Longrightarrow_{\mathcal{R}\text{-Sol-H}} \quad & \varnothing;\ \{\tilde{y}_1\colon \epsilon \triangleq b,\ \tilde{y}_3\colon f(b, b) \triangleq b,\ \tilde{z}_2\colon \epsilon \triangleq \epsilon\};\ \{\tilde{x} \mapsto (\tilde{y}_1, f(a(\tilde{z}_2), \tilde{y}_3), \tilde{y}_1)\} \\
\Longrightarrow_{\mathcal{R}\text{-Clr-S}} \quad & \varnothing;\ \{\tilde{y}_1\colon \epsilon \triangleq b,\ \tilde{y}_3\colon f(b, b) \triangleq b\};\ \{\tilde{x} \mapsto (\tilde{y}_1, f(a, \tilde{y}_3), \tilde{y}_1)\}
\end{aligned}
$$

$\tilde{x}\sigma = (\tilde{y}_1, f(a, \tilde{y}_3), \tilde{y}_1)$ is the rigid lgg of $\tilde{s}$ and $\tilde{q}$ with respect to $\mathcal{R}$.

Notice that the least general rigid generalization from Example 2.11 is unique but, in general, the rigidity function $\mathcal{R}$ causes branching. Therefore the computational complexity highly depends on the function $\mathcal{R}$.

Example 2.12. *Now we will illustrate how the algorithm $\mathfrak{G}_{\mathcal{R}}$ can be used to detect software clones of the program from Figure 2.4. The example is composed from the taxonomy of editing scenarios for different clone types, described in [75].*

```
1 void sumProd(int n) {
2   float sum = 0.0;
3   float prod = 1.0;
4   for(int i=1; i<=n; i++) {
5     sum = sum + i;
6     prod = prod * i;
7     foo(sum,prod); }}
```

Figure 2.4: The original program used to illustrate clone detection by anti-unification.

```
1 void sumProd(int n) {
2   float sum = 0.0;
3   float prod = 1.0;
4   for(int i=1; i<=n; i++) {
5     sum = sum + (i*i);
6     prod = prod * (i*i);
7     foo(sum, prod); }}
```

```
void sumProd(int n) {
  float sum = 0.0;
  float prod = 1.0;
  for(int i=1; i<=n; i++) {
    sum = sum + i;
    // line deleted
    foo(sum,prod); }}
```

Figure 2.5: Two clones of the program from Figure 2.4.

Figure 2.5 shows two different types of clones of the program from Figure 2.4. The left one is of Type-2 and the right one of Type-3. To use the rigid anti-unification algorithm, we translate the input source-codes into hedges. It is straightforward to encode abstract syntax trees, like it was proposed in [16, 33, 48], with hedges. Figure 2.6 shows the term encodings of the abstract syntax trees of the program and its clones.

$$
\begin{aligned}
&sumProd(input(type(int), n),\\
&\qquad returnType(void),\\
&\quad =(type(float), sum, 0.0),\\
&\quad =(type(float), prod, 1.0),\\
&\quad for(=(type(int), i, 1), \leqslant(i, n), ++(i),\\
&\qquad =(sum, +(sum, i)),\\
&\qquad =(prod, *(prod, i)),\\
&\qquad foo(sum, prod)))
\end{aligned}
$$

$$
\begin{aligned}
&sumProd(input(type(int), n),\\
&\qquad returnType(void),\\
&\quad =(type(float), sum, 0.0),\\
&\quad =(type(float), prod, 1.0),\\
&\quad for(=(type(int), i, 1), \leqslant(i, n), ++(i),\\
&\qquad =(sum, +(sum, *(i, i))),\\
&\qquad =(prod, *(prod, *(i, i))),\\
&\qquad foo(sum, prod)))
\end{aligned}
$$

$$
\begin{aligned}
&sumProd(input(type(int), n),\\
&\qquad returnType(void),\\
&\quad =(type(float), sum, 0.0),\\
&\quad =(type(float), prod, 1.0),\\
&\quad for(=(type(int), i, 1), \leqslant(i, n), ++(i),\\
&\qquad =(sum, +(sum, i)),\\
&\qquad\\
&\qquad foo(sum, prod)))
\end{aligned}
$$

Figure 2.6: The original program and its clones as unranked terms.

We now use the anti-unification algorithm $\mathfrak{G}_{\mathcal{R}}$ to compare two pieces of software codes. The lgg will show how similar these pieces are and at which parts they differ. All the differences are available from the store. A clone detection tool can use this information to draw conclusions. The most suitable rigidity function for detecting software clones is the one that computes the set of longest common subsequences.

Therefore, we also choose it for this example. Figure 2.7 shows the resulting lggs for the original code and the, respective, clone. The left clone and the original code only have a singleton mcg of $\mathcal{R}$-generalizations but for the right clone and the original code the algorithm $\mathfrak{G}_{\mathcal{R}}$ yields two lggs. For such cases, a clone detection software can analyze the store to select the better generalization.

$$
\begin{array}{l}
sumProd(input(type(int), n),\\
\quad returnType(void),\\
\quad =(type(float), sum, 0.0),\\
\quad =(type(float), prod, 1.0),\\
\quad for(=(type(int), i, 1), \leqslant(i, n), {+}{+}(i),\\
\quad\quad =(sum, +(sum, \tilde{\mathbf{x}})),\\
\quad\quad =(prod, *(prod, \tilde{\mathbf{x}})),\\
\quad\quad foo(sum, prod)))
\end{array}
$$

$$
\begin{array}{l}
sumProd(input(type(int), n),\\
\quad returnType(void),\\
\quad =(type(float), sum, 0.0),\\
\quad =(type(float), prod, 1.0),\\
\quad for(=(type(int), i, 1), \leqslant(i, n), {+}{+}(i),\\
\quad\quad =(sum, +(sum, i)),\\
\quad\quad \tilde{\mathbf{x}}\\
\quad\quad foo(sum, prod)))
\end{array}
$$

$$
\begin{array}{l}
sumProd(input(type(int), n),\\
\quad returnType(void),\\
\quad =(type(float), sum, 0.0),\\
\quad =(type(float), prod, 1.0),\\
\quad for(=(type(int), i, 1), \leqslant(i, n), {+}{+}(i),\\
\quad\quad \tilde{\mathbf{x}}\\
\quad\quad =(\tilde{\mathbf{y}}),\\
\quad\quad foo(sum, prod)))
\end{array}
$$

Figure 2.7: Clone detection by applying $\mathfrak{G}_{\mathcal{R}}$ to the hedges from Figure 2.6.

Let us consider two more clones of the original program from Figure 2.4. Another one of Type-3 and one clone of Type-4. Figure 2.8 shows the clones. Again we translate the code into hedges and run the algorithm $\mathfrak{G}_{\mathcal{R}}$ on the original code and the clones. The translation of the clones into hedges will be skipped here (see Figure 2.19).

```
1  void sumProd(int n) {
2    float sum = 0.0;
3    float prod = 1.0;
4    for(int i=1; i<=n; i++) {
5      sum = sum + i;
6      prod = prod * i;
7      if (n % 2)==0 {
8        foo(sum, prod); }}}
```

```
void sumProd(int n) {
  float sum = 0.0;
  float prod = 1.0;
  int i=1; while(i<=n) {
    sum = sum + i;
    prod = prod * i;
    foo(sum,prod); i++; }}
```

Figure 2.8: Clones of Type-3 and Type-4.

Running $\mathfrak{G}_{\mathcal{R}}$ on the clones from Figure 2.8 and the original code results in both cases in a singleton mcg. We show the unique results in Figure 2.9.

Notice that in the second case the assignment `int i=1;` at line 4 from Figure 2.8 has been moved out from the loop declaration so that the three assignments of the program variables `sum`, `prod`, and `i` are at the same level. For that reason we have three possibilities to choose 2 of 3 assignments $\binom{3}{2}$ when searching for longest common subsequences. This leads to branching of the anti-unification algorithm and it computes three generalizations. Nevertheless, there is only one lgg, namely the one that has the assignments from line 2 and 3 in it.

$$
\begin{array}{ll}
sumProd(input(type(int), n), & sumProd(input(type(int), n), \\
\quad returnType(void), & \quad returnType(void), \\
\quad =(type(float), sum, 0.0), & \quad =(type(float), sum, 0.0), \\
\quad =(type(float), prod, 1.0), & \quad =(type(float), prod, 1.0), \\
\quad for(=(type(int), i, 1), \leqslant(i, n), {+}{+}(i), & \quad \tilde{\mathbf{x}}) \\
\qquad =(sum, +(sum, i)), & \\
\qquad =(prod, *(prod, i)), \tilde{\mathbf{x}})) &
\end{array}
$$

Figure 2.9: Result of running $\mathfrak{G}_{\mathcal{R}}$ to detect the clones from Figure 2.8.

In the first clone the application of foo has been nested into an if-statement and in the second one the for-loop has been replaced by a while-loop. Since we represent such control flow operations as function applications in the syntax tree, first-order anti-unification cannot detect those similarities. We will address this shortcoming in section 2.2 by introducing higher-order variables.

2.1.4 Minimization by Anti-Unification using $\mathfrak{G}_{\mathcal{R}}$

Since minimization of the set of final states $\mathfrak{G}_{\mathcal{R}}(\tilde{x}\colon \tilde{s} \triangleq \tilde{q})$ requires to solve a hedge-matching problem and universal algorithms that solve hedge-matching problems are NP-complete (see, e.g., [51, 52]), we investigate here the special case of hedges that do not contain pairs of consecutive variables.

In the introduction we already discussed how anti-unification may be used to solve matching problems if deciding $\tilde{s} \stackrel{?}{\simeq} \tilde{q}$ for two hedges $\tilde{s}$ and $\tilde{q}$ can be done without deciding $\tilde{s} \stackrel{?}{\preceq} \tilde{q}$ and $\tilde{q} \stackrel{?}{\preceq} \tilde{s}$ (see Corollary 1.9). Since $\mathfrak{G}_{\mathcal{R}}$ is not an universal algorithm but parametric by $\mathcal{R}$, it needs some more investigation. Before we start our investigation of the algorithm $\mathfrak{G}_{\mathcal{R}}$, we show that $\tilde{s} \stackrel{?}{\simeq} \tilde{q}$ can be decided by term traversal in linear time for hedges that do not contain pairs of consecutive variables.

Lemma 2.4. *Let $\tilde{s}$ and $\tilde{q}$ be hedges such that $\tilde{s} \simeq \tilde{q}$. Then $|\mathrm{Pos}_f(\tilde{s})| = |\mathrm{Pos}_f(\tilde{q})|$, for all $f \in \mathcal{F}$.*

Proof. Let $\tilde{s}$ and $\tilde{q}$ be hedges such that $\tilde{s} \simeq \tilde{q}$. Then there exist substitutions σ, ϑ such that $\tilde{s}\sigma = \tilde{q}$ and $\tilde{q}\vartheta = \tilde{s}$. W.l.o.g. we assume that a function symbol $f \in \mathcal{F}$ occurs

more often in $\tilde{s}$ than in $\tilde{q}$. This contradicts to the existence of the substitution $\tilde{s}\sigma = \tilde{q}$ because σ cannot reduce the number of occurrences of a function symbol f from $\tilde{s}$. □

Theorem 2.5. *Let $\tilde{s}$ and $\tilde{q}$ be hedges that do not contain pairs of consecutive hedge variables and let $\tilde{s} \simeq \tilde{q}$. There exists a renaming σ such that $\tilde{s}\sigma = \tilde{q}$ (and vice versa).*

Proof. Let $\tilde{s}$ and $\tilde{q}$ be hedges that do not contain pairs of consecutive hedge variables and let $\tilde{s} \simeq \tilde{q}$. We know that there is some substitution σ such that $\tilde{s}\sigma = \tilde{q}$ and $\mathrm{Dom}(\sigma)$ is a subset of the variables that occur in $\tilde{s}$. By assumption it follows that $\mathrm{Ran}(\sigma)$ does not contain pairs of consecutive hedge variables and together with Lemma 2.4 follows that $\mathrm{Ran}(\sigma) \subseteq \mathcal{V}_{\mathsf{H}}$. Let $\sigma = \{\tilde{x}_1 \mapsto \tilde{y}_1, \ldots, \tilde{x}_n \mapsto \tilde{y}_n\}$. Since $\tilde{s} \simeq \tilde{q}$, there is also a substitution ϑ so that $\tilde{q}\vartheta = \tilde{s}$. Therefore $\tilde{s}\sigma\vartheta = \tilde{s}$. It follows that $\vartheta = \{\tilde{y}_1 \mapsto \tilde{x}_1, \ldots, \tilde{y}_n \mapsto \tilde{x}_n\}$. This means that σ is a bijection from domain variables to range variables. □

Corollary 2.6. *Let $\tilde{s}$ and $\tilde{q}$ be hedges that do not contain pairs of consecutive hedge variables and let $\tilde{s} \simeq \tilde{q}$. Then $\|\tilde{s}\| = \|\tilde{q}\|$.*

In order to efficiently decide $\tilde{s} \stackrel{?}{\simeq} \tilde{q}$ for two hedges $\tilde{s}$ and $\tilde{q}$ that do not contain pairs of consecutive hedge variables, and to compute their renaming in case of $\tilde{s} \simeq \tilde{q}$, we use the idea of sharing variables by representing terms as directed, acyclic graphs (dags) from [10]. Similarly to Baader and Snyder, we define an unranked term dag as directed, acyclic graph that corresponds to an unranked term.

Definition 2.15 (Unranked term dag)**.** *An* unranked term dag *is a directed, acyclic graph that is weakly connected and whose nodes are labeled with function symbols or variables. Function symbols may have incoming and outgoing edges while variables do not have outgoing edges. There is one function symbol that does not have incoming edges, called the* root *of the term dag. The outgoing edges from any node are ordered.*

Note that dags that represent the same term differ from each other by the amount of structure they share among the subterms. Figure 2.10 shows four different dags that represent the same term.

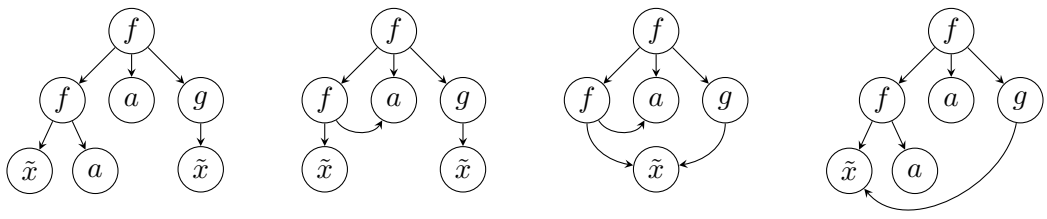

Figure 2.10: Dags of the term $f(f(\tilde{x}, a), a, g(\tilde{x}))$. Two of them are variable sharing while the others are not.

Definition 2.16 (Variable sharing term dag)**.** *A* variable sharing term dag *is a term dag where all occurrences of the same variable share the same node of the graph.*

From [10] we know that constructing a variable sharing dag that doesn't share function symbols, like the rightmost one from Figure 2.10, can be done in time linear to

the size of the term, assuming that names of variables are strings of characters whose length is bounded by a constant.

To decide $\tilde{s} \stackrel{?}{\preceq} \tilde{q}$ for two hedges $\tilde{s}$ and $\tilde{q}$ that do not contain pairs of consecutive hedge variables, we first construct the dags $f(\tilde{s})$ and $f(\tilde{q})$, where $f \in \mathcal{F}$ is arbitrary. Afterwards, the terms are traversed synchronously. Whenever we encounter any difference except for the names of variables, we know that $\tilde{s} \npreceq \tilde{q}$.

Furthermore, for each pair of variables $\tilde{x} = \tilde{s}|_I, \tilde{y} = \tilde{q}|_I$, that we encounter during the traversal process, we record their names as mappings $\tilde{x} \mapsto \tilde{y}$ and $\tilde{y} \mapsto \tilde{x}$ in hash tables. (Note that both variables occur at the same position I.) If there is already a mapping $\tilde{x} \mapsto \tilde{z}$ and $\tilde{z} \neq \tilde{y}$, then $\tilde{s} \npreceq \tilde{q}$. The same holds for $\tilde{y}$'s mapping. Since variable names are bounded by a constant, we can assume a perfect hash function.

Corollary 2.7. *For arbitrary hedges $\tilde{s}$ and $\tilde{q}$ that do not contain pairs of consecutive hedge variables, the decision problem $\tilde{s} \stackrel{?}{\preceq} \tilde{q}$ can be solved by traversal of their variable sharing term dags $f(\tilde{s})$ and $f(\tilde{q})$ in linear time, where $f \in \mathcal{F}$ is an arbitrary root symbol.*

Example 2.13. *Consider the hedges $\tilde{s} = (f(\tilde{x}_1, a, \tilde{x}_2), \tilde{x}_2, g(\tilde{x}_1))$ and $\tilde{q} = (f(\tilde{y}_1, a, \tilde{y}_2), \tilde{y}_2, g(\tilde{y}_1))$. We get the mappings $\{\tilde{x}_1 \mapsto \tilde{y}_1, \tilde{x}_2 \mapsto \tilde{y}_2\}$ and $\{\tilde{y}_1 \mapsto \tilde{x}_1, \tilde{y}_2 \mapsto \tilde{x}_2\}$. Figure 2.11 illustrates those mappings by the dashed arrows between the two variable sharing dags $f(\tilde{s})$ and $f(\tilde{q})$.*

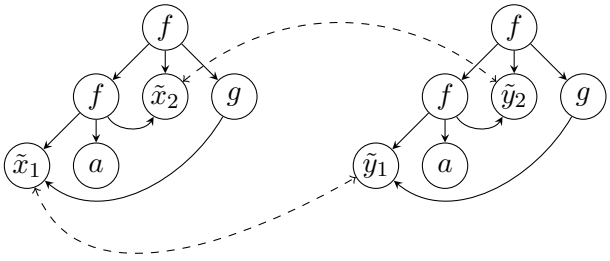

Figure 2.11: The hedges $\tilde{s}$ and $\tilde{q}$ from Example 2.13 as variable sharing term dags.

Matching with $\mathfrak{G}_{\mathcal{R}}$. Now we turn to discussing the algorithm $\mathfrak{G}_{\mathcal{R}}$. Our goal is to minimize the complete set of final states $\mathfrak{G}_{\mathcal{R}}(\tilde{x}\colon \tilde{s} \triangleq \tilde{q})$ by using the algorithm $\mathfrak{G}_{\mathcal{R}}$ itself (with the same $\mathcal{R}$). We will also show the implication that $\mathfrak{G}_{\mathcal{R}}$ solves the matching problem for arbitrary hedges that do not contain pairs of consecutive variables.

Definition 2.17. *We define two substitutions obtained by a set S of AUEs:*

$$\sigma_{L}(S) ::= \{\tilde{x} \mapsto \tilde{s} \mid \tilde{x}\colon \tilde{s} \triangleq \tilde{q} \in S\}$$
$$\sigma_{R}(S) ::= \{\tilde{x} \mapsto \tilde{q} \mid \tilde{x}\colon \tilde{s} \triangleq \tilde{q} \in S\}$$

Let $\varnothing; S; \sigma \in \mathfrak{G}_{\mathcal{R}}(\tilde{x}\colon \tilde{s} \triangleq \tilde{q})$. We assume[†] that $\mathfrak{G}_{\mathcal{R}}$ is coherent so that $\tilde{x}\sigma\sigma_{\mathrm{L}}(S) = \tilde{s}$ and $\tilde{x}\sigma\sigma_{\mathrm{R}}(S) = \tilde{q}$. The algorithm $\mathfrak{G}_{\mathcal{R}}$ computes a complete set of $\mathcal{R}$-generalizations that

[†]Note that in Kutsia et al. [54] there is no proof of the soundness of the store, i.e., $\tilde{x}\sigma\sigma_{\mathrm{L}}(S) = \tilde{s}$ and $\tilde{x}\sigma\sigma_{\mathrm{R}}(S) = \tilde{q}$ has not been proven (see, e.g., Theorem 1.5). Nevertheless, we have strong evidence that the store computed by $\mathfrak{G}_{\mathcal{R}}$ is sound, since in subsection 2.2.6, we prove the soundness of the store of a more general algorithm that subsumes the algorithm $\mathfrak{G}_{\mathcal{R}}$.

is not minimal in general. However, the following theorem tells us, that a minimization step can be performed by $\mathfrak{G}_{\mathcal{R}}$ itself. We only need to decide $\tilde{s}' \stackrel{?}{\simeq} \tilde{q}'$ for two hedges $\tilde{s}'$ and $\tilde{q}'$ that do not contain pairs of consecutive variables. This can be done as described above.

Theorem 2.8. *Let* $\{\varnothing;\, S;\, \sigma,\ \varnothing;\, S';\, \sigma'\} \subseteq \mathfrak{G}_{\mathcal{R}}(\tilde{x}\colon \tilde{s} \triangleq \tilde{q})$. *Let* $\tilde{g} = \tilde{x}\sigma$ *and* $\tilde{g}' = \tilde{x}\sigma'$ *and* $\tilde{y}$ *be a hedge variable that occurs neither in* $\tilde{g}$ *nor in* $\tilde{g}'$.

- *Then* $\tilde{g} \preceq \tilde{g}'$ *iff there is* $\varnothing;\, S'';\, \sigma'' \in \mathfrak{G}_{\mathcal{R}}(\tilde{y}\colon \tilde{g} \triangleq \tilde{g}')$ *such that* $\tilde{y}\sigma'' \simeq \tilde{g}$.
- *Then* $\tilde{g}' \preceq \tilde{g}$ *iff there is* $\varnothing;\, S'';\, \sigma'' \in \mathfrak{G}_{\mathcal{R}}(\tilde{y}\colon \tilde{g} \triangleq \tilde{g}')$ *such that* $\tilde{y}\sigma'' \simeq \tilde{g}'$.

Proof. Let $\{\varnothing;\, S;\, \sigma,\ \varnothing;\, S';\, \sigma'\} \subseteq \mathfrak{G}_{\mathcal{R}}(\tilde{x}\colon \tilde{s} \triangleq \tilde{q})$. Let $\tilde{g} = \tilde{x}\sigma$ and $\tilde{g}' = \tilde{x}\sigma'$ and $\tilde{y}$ be a hedge variable that occurs neither in $\tilde{g}$ nor in $\tilde{g}'$. We show $\tilde{g} \preceq \tilde{g}'$ iff there is $\varnothing;\, S'';\, \sigma'' \in \mathfrak{G}_{\mathcal{R}}(\tilde{y}\colon \tilde{g} \triangleq \tilde{g}')$ such that $\tilde{g} \simeq \tilde{y}\sigma''$.

($\Rightarrow$) Assume $\tilde{g} \preceq \tilde{g}'$. We know that $\tilde{g}$ is an $\mathcal{R}$-generalization. Therefore by soundness and completeness of $\mathfrak{G}_{\mathcal{R}}$ it follows that there is $\varnothing;\, S'';\, \sigma'' \in \mathfrak{G}_{\mathcal{R}}(\tilde{z}\colon \tilde{g} \triangleq \tilde{g})$ such that $\tilde{g} \simeq \tilde{y}\sigma''$. Making one of the two hedges in the AUE $\tilde{z}\colon \tilde{g} \triangleq \tilde{g}$ more specific than the other one gives by soundness and completeness of $\mathfrak{G}_{\mathcal{R}}$ again such a final state $\varnothing;\, S'';\, \sigma''$. By the fact that $\mathfrak{G}_{\mathcal{R}}$ is coherent it follows that $\tilde{g}\vartheta\sigma_{\mathrm{R}}(S'') = \tilde{g}'$ for the renaming ϑ so that $\tilde{g}\vartheta = \tilde{y}\sigma''$.

($\Leftarrow$) Assume there is $\varnothing;\, S'';\, \sigma'' \in \mathfrak{G}_{\mathcal{R}}(\tilde{y}\colon \tilde{g} \triangleq \tilde{g}')$ such that $\tilde{g} \simeq \tilde{y}\sigma''$. By coherence of $\mathfrak{G}_{\mathcal{R}}$ follows that $\tilde{y}\sigma''\sigma_{\mathrm{R}}(S'') = \tilde{g}'$. By the assumption $\tilde{g} \simeq \tilde{y}\sigma''$ there is a renaming ϑ such that $\tilde{g}\vartheta = \tilde{y}\sigma''$. For that reasons $\tilde{g}\vartheta\sigma_{\mathrm{R}}(S'') = \tilde{g}'$. □

From Theorem 2.5 follows that in Theorem 2.8 if $\tilde{g} \preceq \tilde{g}'$, then the matcher can easily be obtained by computing a *renaming* substitution:

Corollary 2.9. *Let* $\{\varnothing;\, S;\, \sigma,\ \varnothing;\, S';\, \sigma'\} \subseteq \mathfrak{G}_{\mathcal{R}}(\tilde{x}\colon \tilde{s} \triangleq \tilde{q})$. *Let* $\tilde{g} = \tilde{x}\sigma$ *and* $\tilde{g}' = \tilde{x}\sigma'$ *and* $\tilde{y}$ *be a hedge variable that occurs neither in* $\tilde{g}$ *nor in* $\tilde{g}'$.

- *If* $\tilde{g} \preceq \tilde{g}'$ *then* $\tilde{g}\vartheta\sigma_{R}(S'') = \tilde{g}'$ *where* ϑ *is the renaming* $\tilde{y}\sigma''\vartheta = \tilde{g}$.
- *If* $\tilde{g}' \preceq \tilde{g}$ *then* $\tilde{g}'\vartheta\sigma_{L}(S'') = \tilde{g}$ *where* ϑ *is the renaming* $\tilde{y}\sigma''\vartheta = \tilde{g}'$.

As the decision problem $\tilde{s} \stackrel{?}{\simeq} \tilde{q}$ for hedges that do not contain pairs of consecutive variables can be solved in linear time, the complexity of the minimization step heavily depends on the complexity of the rigidity function.

The results from Theorem 2.8 lead to a more general theorem which states that we can decide any matching problem for two arbitrary hedges that do not contain pairs of consecutive variables by the algorithm $\mathfrak{G}_{\mathcal{R}}$.

Theorem 2.10. *Let* $\mathcal{R}$ *be the rigidity function that computes the set of* all *common subsequences and let* $\tilde{s}$ *and* $\tilde{q}$ *be hedges that do not contain pairs of consecutive variables.* $\mathfrak{G}_{\mathcal{R}}$ *solves the matching problem* $\tilde{s} \stackrel{?}{\preceq} \tilde{q}$ *(and in the same run also* $\tilde{q} \stackrel{?}{\preceq} \tilde{s}$*).*

Proof. Let $\mathcal{R}$ be the rigidity function that computes the set of *all* common subsequences and let $\tilde{s}$ and $\tilde{q}$ be hedges that do not contain pairs of consecutive variables. We can assume that $\tilde{s}$ and $\tilde{q}$ do not share variables. (We compute a renaming and compose it with the matcher in case of success.) By Theorem 2.8 we can obtain a minimal complete set of $\mathcal{R}$-generalizations of $\tilde{s}$ and $\tilde{q}$ by $\mathfrak{G}_{\mathcal{R}}$. Let us call it G.

Case $\tilde{s} \not\preceq \tilde{q}$: By the soundness of $\mathfrak{G}_{\mathcal{R}}$ there is no $\tilde{g} \in G$ so that $\tilde{g} \simeq \tilde{s}$.

Case $\tilde{s} \preceq \tilde{q}$: We show that $G = \{\tilde{g}\}$ so that $\tilde{g} \simeq \tilde{s}$. Since $\mathfrak{G}_{\mathcal{R}}$ is sound and complete, there is $\tilde{g} \in G$ such that $\tilde{g} \preceq \tilde{s}$. Assume that $\tilde{g} \prec \tilde{s}$. Then there is a substitution $\vartheta = \{\tilde{x}_1 \mapsto \tilde{r}_1, \ldots, \tilde{x}_n \mapsto \tilde{r}_n\}$ such that $\tilde{g}\vartheta = \tilde{s}$ and ϑ is not a renaming. As $\tilde{s}$ does not contain consecutive variables there is $\tilde{r}_i$ $(1 \leqslant i \leqslant n)$ that contains a function application, say $f(\tilde{r}')$, for some $\tilde{r}'$. Because $\tilde{s} \preceq \tilde{q}$ the term $f(\tilde{r}')$ occurs in both hedges $\tilde{s}$ and $\tilde{q}$ at a corresponding level. Since $\mathcal{R}$ computes all the subsequences of top symbols, the subsequence that contains f would be longer, giving rise to a strictly less generalization than $\tilde{g}$. This contradicts to $\tilde{g}$ being an lgg. Since we proved that $\tilde{g} \simeq \tilde{s}$ holds, for all $\tilde{g} \in G$, it follows that G is a singleton if $\tilde{s} \preceq \tilde{q}$.

We can read $\sigma_{\mathrm{R}}(S)$ from the store S that corresponds to $\tilde{g}$ so that $\tilde{g}\sigma_{\mathrm{R}}(S) = \tilde{q}$. Putting this together leads to the matcher $\vartheta'\sigma_{\mathrm{R}}(S)$, where ϑ' is the renaming $\tilde{s}\vartheta' = \tilde{g}$. □

Example 2.14. *Let $\mathcal{R}$ be the rigidity function that computes the set of longest common subsequences. We demonstrate our results on the hedges $\tilde{s} = (f(a), f(a), f(b), a)$ and $\tilde{q} = (f(a), f(a), b)$. $\mathfrak{G}_{\mathcal{R}}(\tilde{x}\colon \tilde{s} \triangleq \tilde{q})$ computes the three generalizations*

$$\tilde{g}_1 = (f(a), f(a), \tilde{y}_1), \quad \tilde{g}_2 = (\tilde{y}_2, f(a), f(\tilde{y}_3), \tilde{y}_4), \text{ and } \tilde{g}_3 = (f(a), \tilde{y}_5, f(\tilde{y}_6), \tilde{y}_7).$$

When applying $\mathfrak{G}_{\mathcal{R}}$ to $\tilde{g}_1$ and $\tilde{g}_2$ we get the unique generalization $(\tilde{z}_1, f(a), f(\tilde{z}_2), \tilde{z}_3)$. Since $\tilde{g}_2 \simeq (\tilde{z}_1, f(a), f(\tilde{z}_2), \tilde{z}_3)$ we can drop $\tilde{g}_2$. Similarly we can detect that $\tilde{g}_3 \preceq \tilde{g}_1$.

Let us discuss the example in a bit more detail. When applying $\mathfrak{G}_{\mathcal{R}}$ to $\tilde{g}_2$ and $\tilde{g}_3$ we get the generalization $\tilde{g}_4 = (\tilde{z}_1, f(a), \tilde{z}_2, f(\tilde{z}_3), \tilde{z}_4)$ and the store $\{\tilde{z}_1\colon \tilde{y}_2 \triangleq \epsilon, \tilde{z}_2\colon \epsilon \triangleq \tilde{y}_5, \tilde{z}_3\colon \tilde{y}_3 \triangleq \tilde{y}_6, \tilde{z}_4\colon \tilde{y}_4 \triangleq \tilde{y}_7\}$. Neither $\tilde{g}_2 \simeq \tilde{g}_4$ nor $\tilde{g}_3 \simeq \tilde{g}_4$ holds. Therefore $\tilde{g}_2$ and $\tilde{g}_3$ are incomparable with respect to $\preceq$ but they are joinable by Theorem 1.10 and the join $\tilde{g}_5 = \tilde{g}_4\{\tilde{z}_1 \mapsto \epsilon, \tilde{z}_2 \mapsto \epsilon\} = (f(a), f(\tilde{z}_3), \tilde{z}_4)$ can be read from the store. Since $\tilde{g}_5 \preceq \tilde{g}_1$ it also follows that $\tilde{g}_2$ and $\tilde{g}_3$ are more general than $\tilde{g}_1$.

Notice that in case of Example 2.14 there is the unique (modulo $\simeq$) higher-order lgg $(f(a), f(a), \tilde{X}(b), \tilde{y})$, where $\tilde{X}$ is a higher-order variable. That higher-order lgg is strictly less general than those computed by $\mathfrak{G}_{\mathcal{R}}$ since it contains an additional function symbol. As we can see on the clone detection example (Figure 2.9) and on Example 2.14, there are similarities that cannot be detected by first-order anti-unification. In section 2.2 we introduce higher-order variables to detect those similarities.

2.2 Higher-Order Unranked Anti-Unification $2\mathcal{V}$

To compute unranked higher-order generalizations of two input hedges we again use the idea of guiding the generalization process by (rigid) skeletons of the input hedges. The first-order unranked anti-unification algorithm uses a rigidity function that computes the skeletons level by level. This approach does not work for computing higher-order generalizations because our goal is to detect similarities at different levels of the input hedges (see, e.g., Example 2.15). Therefore, we decouple the skeleton generation from the computation of the generalization. Separating this two steps enables us to study the complexity of the generalization process independently of the skeleton computation. That was not possible for the algorithm from subsection 2.1.2.

A skeleton is given in the form of an admissible alignment, which is a certain sequence of symbols occurring in both hedges. We need to restrict variable occurrences in the generalization to guarantee that for each admissible alignment a unique lgg is computed. We develop an algorithm that computes a rigid lgg of two hedges with respect to a certain admissible alignment. The algorithm runs in quadratic time and requires linear space with respect to the size of the input. This result means that, for instance, if the skeleton is a constrained longest common subhedge of the input hedges in the sense of [87], then both skeleton and generalization computation can be done in quadratic time, because the time complexity of computing a constrained lcs is quadratic. One can also compute an lgg which contains, for instance, a constrained longest common subforest [5], or an agreement subhedge/subtree [46] of the input hedges.

The 2$\mathcal{V}$ in the title stands for the two kinds of variables we permit: Context variables to abstract vertical differences between terms, and hedge variables to abstract horizontal differences. Contexts that we consider here are hedges with a single occurrence of the distinguished symbol "hole". They are functions which can apply to another context or to a hedge, which are then "plugged" in the place of the hole.

Example 2.15. *The hedge* $(\tilde{X}(a), f(\tilde{X}(g(a,\tilde{x}),c),\tilde{x}))$ *is a generalization of two hedges* $(h(a), f(h(g(a,b,b),c),b,b))$ *and* $(a, f(g(a,d),c,d))$*. Dotted and dashed nodes indicate differences, while the solid ones form the admissible alignment. The first hedge can be obtained from the generalization by replacing the context variable* $\tilde{X}$ *with the context* $h(\circ)$ *and the hedge variable* $\tilde{x}$ *with the hedge* (b,b)*. To obtain the second hedge, we need to replace* $\tilde{X}$ *with the hole (i.e., to eliminate* $\tilde{X}$*) and to replace* $\tilde{x}$ *by* d*.*

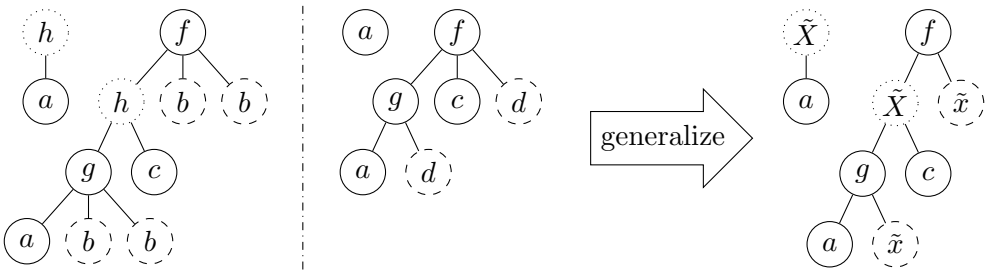

Figure 2.12: Two hedges and their higher-order lgg.

Similarly to the first-order case, we exploit the fact that generalizations contain a common subsequence of the word representation of the input hedges. An admissible alignment is such a subsequence that additionally contains the positions of the symbols at the input hedges. Observe, e.g., the hedges from Example 2.15. The admissible alignment contains the bold symbols:

$$\begin{array}{l}(\ h(\mathbf{a}), \mathbf{f}(\ h(\mathbf{g}(\mathbf{a},b,b),\mathbf{c}),b,b))\\ \underline{(\quad \mathbf{a}\ , \mathbf{f}(\quad \mathbf{g}(\mathbf{a},d\quad),\mathbf{c}\ ,d\quad))}\\ (\tilde{X}(\mathbf{a}), \mathbf{f}(\tilde{X}(\mathbf{g}(\mathbf{a},\tilde{x}\quad),\mathbf{c}),\tilde{x}\quad))\end{array}$$

In some cases, the skeleton can be constructed in multiple ways, giving rise to several admissible alignments. It requires that the generalizations computed for each alignment should be compared to each other, to make the obtained set minimal. We show how that minimization step can be done by the anti-unification algorithm itself.

2.2.1 Higher-Order Unranked Terms and Hedges $2\mathcal{V}$ (Preliminaries)

Many definitions from subsection 2.1.1 are also valid for the more general theory we consider here. In this subsection we give some additional definitions that are needed. Furthermore, we overload those definitions that do not trivially generalize to the following higher-order unranked terms, hedges, and contexts.

Definition 2.18 (Terms, hedges, contexts)**.** *Given pairwise disjoint countable sets of unranked function symbols $\mathcal{F}$ (symbols without fixed arity), hedge variables $\mathcal{V}_{\mathsf{H}}$, context variables $\mathcal{V}_{\mathsf{C}}$, and a special symbol $\circ$ (the hole), we define* terms, hedges, *and* contexts *by the following grammar:*

$$\begin{array}{lll} t & ::= & f(\tilde{s}) \qquad\qquad\qquad\qquad\qquad\qquad\quad \textit{(term)} \\ s & ::= & t \mid \tilde{x} \mid \tilde{X}(\tilde{s}) \qquad\qquad\qquad\qquad\quad \textit{(hedge element)} \\ \tilde{s} & ::= & s_1, \ldots, s_n \qquad\qquad\qquad\qquad\qquad\quad \textit{(hedge)} \\ \tilde{c} & ::= & \tilde{s}_1, \circ, \tilde{s}_2 \mid \tilde{s}_1, f(\tilde{c}), \tilde{s}_2 \mid \tilde{s}_1, \tilde{X}(\tilde{c}), \tilde{s}_2 \qquad \textit{(context)} \end{array}$$

where $f \in \mathcal{F}$, $\tilde{x} \in \mathcal{V}_{\mathsf{H}}$, $\tilde{X} \in \mathcal{V}_{\mathsf{C}}$, and $n \geqslant 0$.

In contrast to the hedges from Definition 2.1, hedges we consider here may contain context variables at functional position. The set of all variables $\mathcal{V}_{\mathsf{H}} \cup \mathcal{V}_{\mathsf{C}}$ is denoted by $\mathcal{V}$. A context can be seen as a hedge over $\mathcal{F} \cup \{\circ\}$ and $\mathcal{V}$, where the hole occurs exactly once. A *singleton context* is a hedge element over $\mathcal{F} \cup \{\circ\}$ and $\mathcal{V}$ with a single hole in it. The set of hedges and contexts constructed over $\mathcal{F} \cup \{\circ\}$ and $\mathcal{V}$ is denoted by $\mathcal{T}(\mathcal{F}, \mathcal{V})$, or simply by $\mathcal{T}$ if the concrete instances of $\mathcal{F}$ and $\mathcal{V}$ are unimportant. We denote arbitrary contexts by $\tilde{c}, \tilde{d}$ and singleton contexts by $\dot{c}, \dot{d}$. The symbols $\tilde{X}, \tilde{Y}, \tilde{Z}$ denote context variables, $\mathcal{h}, \mathcal{g}$ denote a function symbol or a higher-order variable, and by $\mathcal{s}$ we denote an arbitrary symbol from $\mathcal{F} \cup \{\circ\} \cup \mathcal{V}$. E.g., $\mathcal{h}(a, b)$ denotes some application to the argument hedge (a, b) where we do not care whether $\mathcal{h}$ is a function symbol or a higher-order variable.

The brackets $[\,]$, are used for context application. A context $\tilde{c}$ can apply to a hedge $\tilde{s}$, denoted by $\tilde{c}[\tilde{s}]$, obtaining a hedge by replacing the hole in $\tilde{c}$ with $\tilde{s}$. Application of a context to a context is defined similarly.

Example 2.16. *Examples of a term, a hedge, and a context are, respectively, $f(\tilde{X}(a), b)$, $(\tilde{x}, \tilde{X}(a, b), f(f(a), b))$, and $(\tilde{x}, \tilde{X}(a, b), f(f(\circ), b))$. The latter can be applied to a hedge $(a, \tilde{X}(a))$, resulting in $(\tilde{x}, \tilde{X}(a, b), f(f(\circ), b))[a, \tilde{X}(a)] = (\tilde{x}, \tilde{X}(a, b), f(f(a, \tilde{X}(a)), b))$.*

The set of all function symbols which appear in a hedge $\tilde{s}$ (resp., in a context $\tilde{c}$) is denoted by $\mathcal{F}(\tilde{s})$ (resp., by $\mathcal{F}(\tilde{c})$). We overload the notation $\mathcal{F}(A)$ for the set of all function symbols which appear in a set of hedges and contexts $A \subseteq \mathcal{T}$. Similarly we use $\mathcal{V}(\cdot)$, $\mathcal{V}_{\mathsf{H}}(\cdot)$, and $\mathcal{V}_{\mathsf{C}}(\cdot)$ to obtain the set of all variables, all the hedge variables, and all the context variables, respectively.

Example 2.17. *For instance, $\mathcal{F}(\tilde{x}, \tilde{X}(a, b), f(f(a), b)) = \{a, b, f\}$, and $\mathcal{V}(A) = \{\tilde{x}, \tilde{y}, \tilde{z}, \tilde{X}, \tilde{Y}\}$ for $A = \{(\tilde{x}, \tilde{X}(a, b), f(f(a), b)), (\tilde{Y}(a, b), \tilde{x}, f(\tilde{y}, b), \tilde{z})\}$. Furthermore, $\mathcal{V}_{\mathsf{H}}(A) = \{\tilde{x}, \tilde{y}, \tilde{z}\}$ and $\mathcal{V}_{\mathsf{C}}(A) = \{\tilde{X}, \tilde{Y}\}$.*

We naturally extend the notions of the set of positions $\mathrm{Pos}(\tilde{c})$, the top symbols $\mathrm{Top}(\tilde{c})$, the length $|\tilde{c}|$, the size $\|\tilde{c}\|$, and the subterm $\tilde{c}|_I$ to contexts $\tilde{c}$ and to occurrences of context variables. The hole $\circ$ is treated like a function symbol. It follows that the set of positions of hole occurrences $\mathrm{Pos}_\circ(\tilde{c})$ in a context $\tilde{c}$ is a singleton set. E.g., $\mathrm{Pos}_\circ(\tilde{x}, \tilde{X}(a,b), f(\tilde{X}(\circ), b)) = \{3{\cdot}1{\cdot}1\}$ and $\mathrm{Pos}_{\tilde{X}}(\tilde{x}, \tilde{X}(a,b), f(\tilde{X}(\circ), b)) = \{2, 3{\cdot}1\}$.

Definition 2.19 (Position triangle)**.** *Given three positions I_1, I_2 and I_3, the* position triangle *relation $\bowtie$ is defined as*

$$I_1 \bowtie_{I_3} I_2 :\Longleftrightarrow \text{there is } I_4 \neq \epsilon \text{ such that } I_4 \sqsubset I_1 \text{ and } I_4 \sqsubset I_2 \text{ and } I_4 \not\sqsubset I_3 \text{ and } I_1, I_2, I_3 \text{ are pairwise not in } \sqsubseteq .$$

This relation tests whether I_1 and I_2 have a common prefix which is not a prefix of I_3. None of these positions should be a prefix of another.

Example 2.18. *For instance,* $1{\cdot}1 \bowtie_2 1{\cdot}2$*, but* neither $1 \bowtie_3 2$*, nor* $1{\cdot}1 \bowtie_2 1{\cdot}1{\cdot}2$*, nor* $1{\cdot}1 \bowtie_2 1{\cdot}1$*, nor* $1{\cdot}1 \bowtie_{1.3} 1{\cdot}2$*. A real world example of this relation would be two sisters and one of their uncles.*

Definition 2.20 (Substitution)**.** *A* substitution *is a mapping $\sigma : \mathcal{V} \to \mathcal{T}(\mathcal{F}, \mathcal{V})$ from hedge variables to hedges and from context variables to contexts, which is identity*[†] *almost everywhere. When substituting a context variable $\tilde{X}$ by a context, the context will be applied to the argument hedge of $\tilde{X}$.*

Any substitution σ can be extended to a mapping $\hat{\sigma} : \mathcal{T} \to \mathcal{T}$ that can be applied to hedges and contexts. Similarly to subsection 2.1.1, the application is defined by induction on the structure of hedges and contexts:

$$\tilde{s}\hat{\sigma} ::= \begin{cases} f(\tilde{q}\hat{\sigma}) & \text{if } \tilde{s} = f(\tilde{q}), \\ \sigma(\tilde{x}) & \text{if } \tilde{s} = \tilde{x}, \\ \sigma(\tilde{X})[\tilde{q}\hat{\sigma}] & \text{if } \tilde{s} = \tilde{X}(\tilde{q}), \\ \circ & \text{if } \tilde{s} = \circ, \\ s_1\hat{\sigma}, \ldots, s_n\hat{\sigma} & \text{if } |\tilde{s}| \neq 1 \text{ and } \tilde{s} = (s_1, \ldots, s_n). \end{cases}$$

The application of a context can be defined by induction on the structure of contexts:

$$\tilde{c}[\tilde{s}] ::= \begin{cases} \hbar(\tilde{d}[\tilde{s}]) & \text{if } \tilde{c} = \hbar(\tilde{d}), \\ \tilde{s} & \text{if } \tilde{c} = \circ, \\ (\tilde{q}_1, \dot{c}[\tilde{s}], \tilde{q}_2) & \text{if } |\tilde{c}| \neq 1 \text{ and } \tilde{c} = (\tilde{q}_1, \dot{c}, \tilde{q}_2). \end{cases}$$

For a substitution σ the *domain* is the set of variables

$$\mathrm{Dom}(\sigma) ::= \{\tilde{x} \in \mathcal{V}_\mathsf{H} \mid \sigma(\tilde{x}) \neq \tilde{x}\} \cup \{\tilde{X} \in \mathcal{V}_\mathsf{C} \mid \sigma(\tilde{X}) \neq \tilde{X}(\circ)\}$$

and the *range* is the set of hedges and contexts

$$\mathrm{Ran}(\sigma) ::= \{\sigma(\tilde{x}) \mid \tilde{x} \in \mathrm{Dom}(\sigma)\} \cup \{\sigma(\tilde{X}) \mid \tilde{X} \in \mathrm{Dom}(\sigma)\}.$$

To simplify the notation, we do not distinguish between a substitution σ and its extension $\hat{\sigma}$.

[†]The identity mapping of a context variable to a context is defined as the mapping from a variable to its application to the hole. E.g., $\{\tilde{X} \mapsto \tilde{X}(\circ)\}$ is the identity mapping for the context variable $\tilde{X}$.

Example 2.19. *Let* $\sigma = \{\tilde{x} \mapsto \epsilon,\ \tilde{y} \mapsto (a, \tilde{x}),\ \tilde{X} \mapsto g(\circ),\ \tilde{Y} \mapsto (b, g(g(\circ, b)))\}$ *be a substitution, then* $(\tilde{X}(\tilde{x}), \tilde{y}, f(\tilde{X}(\tilde{y})), \tilde{Y}(f(a), a))\sigma = (g, a, \tilde{x}, f(g(a, \tilde{x}), b, g(g(f(a), a, b))))$.

We extend the notion of a renaming from Definition 2.6 so that the idea of having an injective mapping from variables to variables is preserved by considering the identity $\tilde{X}()\{\tilde{X} \mapsto \tilde{Y}(\circ)\} = \tilde{Y}()$. By that consideration a renaming σ in the sense of Definition 2.21 is again a bijection from $\mathrm{Dom}(\sigma)$ to $\mathrm{Ran}(\sigma)$.

Definition 2.21 (Renaming)**.** *A substitution is called* renaming *if* $Ran(\sigma) \subseteq \mathcal{V}_{\mathsf{H}} \cup \{\tilde{X}(\circ) \mid \tilde{X} \in \mathcal{V}_{\mathsf{C}}\}$ *and* $|Dom(\sigma)| = |Ran(\sigma)|$.

The notions of instantiation and generalization are the same as in Definition 2.7 and Definition 2.8. They are defined by the existence of substitutions to compare hedges with respect to the instantiation quasi ordering. Lemma 2.1 and Corollary 2.2 are also directly applicable to this richer term language.

Theorem 2.11. *Higher-order unranked anti-unification is finitary: For any hedges* $\tilde{s}$ *and* $\tilde{q}$ *there exists their minimal complete set of generalizations. This set is finite and unique modulo* $\simeq$.

Proof. Note that a context variable applied to the empty hedge $\tilde{X}()$ may be used as substitute for a hedge variable $\tilde{x}$ because $\tilde{X}() \simeq \tilde{x}$, i.e., $\tilde{X}()\{\tilde{X} \mapsto (\tilde{x}, \circ)\} = \tilde{x}$ and $\tilde{x}\{\tilde{x} \mapsto \tilde{X}()\} = \tilde{X}()$. Therefore, it suffices to consider only context variables.

Since $\tilde{X}()$ acts as generalization for every pair of hedges, we assume there is a nonempty set of hedges G for two arbitrary hedges $\tilde{s}$ and $\tilde{q}$ so that for all $\tilde{g} \in G$ holds that $\tilde{g} \preceq \tilde{s}$ and $\tilde{g} \preceq \tilde{q}$.

Since substitutions cannot eliminate function symbols and there is a σ such that $\tilde{g}\sigma = \tilde{s}$ for all $\tilde{g} \in G$, the number of function symbols of $\tilde{g}$ is bound by the number of function symbols in $\tilde{s}$. Furthermore, from $\tilde{g}\sigma = \tilde{s}$ we know that a variable is either eliminated, or it stands for a certain part of the hedge $\tilde{s}$. There are only finitely many variables in $\tilde{g}$ that stand for some part of $\tilde{s}$ and all the others are eliminated from $\tilde{g}$ when applying σ (like $\{\tilde{Y} \mapsto \circ\}$). In fact the number is bound by $\|\tilde{s}\|$. The same holds for $\tilde{q}$.

Putting this together, there are only finitely many variables in $\tilde{g}$ that indicate some differences at $\tilde{s}$ and $\tilde{q}$. Eliminating all those superfluous variables we obtain $\tilde{g}' = \tilde{g}\{\tilde{X}_1 \mapsto \circ, \tilde{X}_2 \mapsto \circ, \dots\}$, such that the number of variables in $\tilde{g}'$ is bound by $\|\tilde{s}\| + \|\tilde{q}\|$ and $\tilde{g}'$ is a generalization of $\tilde{s}$ and $\tilde{q}$. Since $\tilde{g} \preceq \tilde{g}'$ it can be removed from G if $\tilde{g} \neq \tilde{g}'$. As the size of an lgg is bounded by $\|\tilde{s}\| + \|\tilde{q}\|$, also the number of possible lggs modulo $\simeq$ is bound. This leads to G being finite. □

In the next subsection we generalize the notion of an alignment from Definition 2.11 and introduce the concept of admissible alignments. This are the skeletons that appear in a generalization of two input hedges. We prove that one does not loose generality by considering only generalizations that contain such a skeleton of the input hedges.

2.2.2 The Skeletons: Admissible Alignments

Given two input strings of symbols, an alignment has been defined by Definition 2.11 as a common subsequence of the input strings which is coupled with the information of

the respective positions of the symbol occurrences. Notice that the positions there are integer values. A rigidity function returns a set of alignments for two given strings of symbols and the anti-unification process is guided by recursively applying the rigidity function to two strings of top function symbols for two hedges, while decomposing and going deeper into the terms.

Since this approach does not work for higher-order generalization, we redefine the alignment as being a common subsequence of the word representation (i.e., the string of symbols obtained by depth-first pre-order traversal of the constituent terms) of two input hedges which is coupled with the information of the respective positions of the symbol occurrences. Notice that the positions are strings of integers.

Definition 2.22 (Alignment)**.** *Given two hedges $\tilde{s}$ and $\tilde{q}$, an* alignment *is a sequence of the form $a_1\langle I_1, J_1\rangle \dots a_n\langle I_n, J_n\rangle$ such that:*

- *$I_1 < \dots < I_n$ and $J_1 < \dots < J_n$, and*
- *$a_k = \mathrm{Top}(\tilde{s}|_{I_k}) = \mathrm{Top}(\tilde{q}|_{J_k})$, for all $1 \leqslant k \leqslant n$.*

An alignment represents common function symbols inside of two hedges with the corresponding positions, respecting the ordering $<$. It is a common subsequence of the word representation of those hedges with some additional information about the positions. The length of an alignment $\mathfrak{a}$ is the number of elements in it and we write $|\mathfrak{a}|$. The empty alignment is denoted by $\mathfrak{e}$.

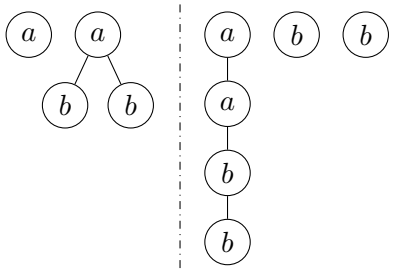

Figure 2.13: $\tilde{s} = (a, a(b, b))$ and $\tilde{q} = (a(a(b(b))), b, b)$.

Example 2.20. *The two hedges $\tilde{s}$ and $\tilde{q}$ from Figure 2.13 have many different alignments. $a\langle 1, 1\rangle a\langle 2, 1{\cdot}1\rangle b\langle 2{\cdot}1, 1{\cdot}1{\cdot}1\rangle b\langle 2{\cdot}2, 3\rangle$ and $a\langle 1, 1{\cdot}1\rangle b\langle 2{\cdot}1, 2\rangle b\langle 2{\cdot}2, 3\rangle$ and $a\langle 2, 1\rangle$ are three of them, while $b\langle 2{\cdot}2, 1{\cdot}1{\cdot}1\rangle a\langle 1, 1\rangle$ is not an alignment.*

The skeletons that are computed level-by-level by a rigidity function may be transformed into a set of (higher-order) alignments by the following transformation rule. First, we initialize the set A of alignments by $A = \mathcal{R}(\mathrm{Top}(\tilde{s}), \mathrm{Top}(\tilde{q}))$. Then the transformation rule $\mathcal{R}$-App is applied to A as long as possible. The given hedges $\tilde{s}$ and $\tilde{q}$ are considered global parameters of the transformation process.

$\mathcal{R}$-App: **Recursive Application of $\mathcal{R}$**

$\{a_1\langle I_1, J_1\rangle \dots a_k\langle I_k, J_k\rangle \dots a_n\langle I_n, J_n\rangle\} \uplus A \Longrightarrow$
$\quad \{a_1\langle I_1, J_1\rangle \dots a_k\langle I_k, J_k\rangle a'_1\langle I_k{\cdot}i_1, J_k{\cdot}j_1\rangle \dots a'_m\langle I_k{\cdot}i_m, J_k{\cdot}j_m\rangle \dots a_n\langle I_n, J_n\rangle \mid$
$\quad a'_1\langle i_1, j_1\rangle \dots a'_m\langle i_m, j_m\rangle \in A'\} \cup A,$
if $(|\mathfrak{a}| = k$ or $I_k \not\dashv I_{k+1})$ and $A' \neq \varnothing$ where $A' = \mathcal{R}(\mathrm{Top}(\tilde{s}_k), \mathrm{Top}(\tilde{q}_k))\backslash\{\mathfrak{e}\}$ and $a_k(\tilde{s}_k) = \tilde{s}|_{I_k}$ and $a_k(\tilde{q}_k) = \tilde{q}|_{J_k}$.

Example 2.21. *To illustrate this transformation, we take the hedges $\tilde{s} = (f(a), f(a), f(b), a)$ and $\tilde{q} = (f(a), f(a), b)$ and $\mathcal{R}$ from Example 2.14, where $\mathcal{R}$ is the rigidity function that computes the set of longest common subsequences. We underline the alignment that is selected by the next rule application. Notice that the choice is not deterministic but $\mathcal{R}$-App is confluent and yields a unique set of alignments.*

$$\{\underline{f\langle 1,1\rangle f\langle 2,2\rangle},\ f\langle 1,1\rangle f\langle 3,2\rangle,\ f\langle 2,1\rangle f\langle 3,2\rangle\} = \mathcal{R}(\mathrm{Top}(\tilde{s}), \mathrm{Top}(\tilde{q}))$$
$$\Longrightarrow^{I_k=1}_{J_k=1} \{\underline{f\langle 1,1\rangle a\langle 1{\cdot}1, 1{\cdot}1\rangle f\langle 2,2\rangle},\ f\langle 1,1\rangle f\langle 3,2\rangle,\ f\langle 2,1\rangle f\langle 3,2\rangle\}$$
$$\Longrightarrow^{I_k=2}_{J_k=2} \{f\langle 1,1\rangle a\langle 1{\cdot}1, 1{\cdot}1\rangle f\langle 2,2\rangle a\langle 2{\cdot}1, 2{\cdot}1\rangle,\ \underline{f\langle 1,1\rangle f\langle 3,2\rangle},\ f\langle 2,1\rangle f\langle 3,2\rangle\}$$
$$\Longrightarrow^{I_k=1}_{J_k=1} \{f\langle 1,1\rangle a\langle 1{\cdot}1, 1{\cdot}1\rangle f\langle 2,2\rangle a\langle 2{\cdot}1, 2{\cdot}1\rangle,\ f\langle 1,1\rangle a\langle 1{\cdot}1, 1{\cdot}1\rangle f\langle 3,2\rangle,\ \underline{f\langle 2,1\rangle f\langle 3,2\rangle}\}$$
$$\Longrightarrow^{I_k=1}_{J_k=2} \{f\langle 1,1\rangle a\langle 1{\cdot}1, 1{\cdot}1\rangle f\langle 2,2\rangle a\langle 2{\cdot}1, 2{\cdot}1\rangle,\ f\langle 1,1\rangle a\langle 1{\cdot}1, 1{\cdot}1\rangle f\langle 3,2\rangle,\ f\langle 2,1\rangle a\langle 2{\cdot}1, 1{\cdot}1\rangle f\langle 3,2\rangle\}$$

Since the rule $\mathcal{R}$-App is not applicable anymore, the set A that corresponds to an exhaustive level-by-level computation of skeletons by $\mathcal{R}$ consists of the last 3 alignments.

Definition 2.23 (Collision)**.** Collisions *in an alignment $\mathfrak{a}$ of two hedges are defined as follows:*

- *A collision appears at two elements $a_k\langle I_k, J_k\rangle$, $a_l\langle I_l, J_l\rangle$ of $\mathfrak{a}$ if either ($I_k \sqsubset I_l$ and $J_k \not\sqsubset J_l$) or ($I_k \not\sqsubset I_l$ and $J_k \sqsubset J_l$).*
- *A collision appears at three elements $a_k\langle I_k, J_k\rangle$, $a_l\langle I_l, J_l\rangle$, $a_n\langle I_n, J_n\rangle$ of $\mathfrak{a}$ if $I_k \bowtie_{I_n} I_l$ and $J_l \bowtie_{J_k} J_n$.*

Example 2.22. *For instance, the alignment $f\langle 2,1\rangle b\langle 2{\cdot}1, 1{\cdot}2\rangle c\langle 2{\cdot}2, 2\rangle$ of the hedges from Figure 2.14 contains a collision at the two elements $f\langle 2,1\rangle$ and $c\langle 2{\cdot}2, 2\rangle$. The alignment $a\langle 1, 1{\cdot}1\rangle b\langle 2{\cdot}1, 1{\cdot}2\rangle c\langle 2{\cdot}2, 2\rangle$ of the same hedges has a collision at its three elements.*

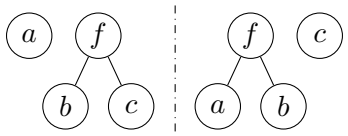

Figure 2.14: $(a, f(b, c))$ and $(f(a, b), c)$.

Definition 2.24 (Admissible alignment)**.** *An alignment of two hedges is called* admissible *if there are no collisions in it.*

Note that for any two elements $a_k\langle I_k, J_k\rangle$ and $a_l\langle I_l, J_l\rangle$ of an admissible alignment, $I_k < I_l$ iff $J_k < J_l$ and $I_k \sqsubset I_l$ iff $J_k \sqsubset J_l$. Alignments that are transformed by $\mathcal{R}$-App are always admissible since the level-by-level construction cannot lead to a collision.

Example 2.23. *For instance $a\langle 1,1\rangle a\langle 2, 1{\cdot}1\rangle b\langle 2{\cdot}1, 1{\cdot}1{\cdot}1\rangle b\langle 2{\cdot}2, 3\rangle$ is an alignment of the hedges from Figure 2.13 but* not *admissible, while the alignments $a\langle 1, 1{\cdot}1\rangle b\langle 2{\cdot}1, 2\rangle b\langle 2{\cdot}2, 3\rangle$ and $a\langle 2,1\rangle b\langle 2{\cdot}2, 1{\cdot}1{\cdot}1{\cdot}1\rangle$ are admissible.*

In order to show that we do not loose generality by considering only generalizations that contain an admissible alignment of the input hedges, we need to uniquely identify those function symbols at the input hedges that appear in the skeleton. Therefore, the following definition is used to uniquely rename all the symbols in a given admissible alignment and in the corresponding hedges. Notice that the renaming is only necessary for theoretical considerations in Theorem 2.12. It is not of practical use for our developing of an anti-unification algorithm in the following subsection.

Definition 2.25 (Distinct alignment renaming)**.** *Given an alignment $\mathfrak{a} = a_1\langle I_1, J_1\rangle \ldots a_m\langle I_m, J_m\rangle$ of two hedges $\tilde{s}$ and $\tilde{q}$, and a sequence of pairwise distinct (fresh) symbols $\acute{a}_1, \ldots, \acute{a}_m$ which occur neither in $\tilde{s}$ nor in $\tilde{q}$. A* distinct alignment renaming *is a renaming of all the symbols in $\mathfrak{a}$ by the fresh ones $\acute{a}_1\langle I_1, J_1\rangle \ldots \acute{a}_m\langle I_m, J_m\rangle$. Furthermore, the top symbol a_k at $\tilde{s}|_{I_k}$ and $\tilde{q}|_{J_k}$ is replaced by $\acute{a}_k$, for all $1 \leqslant k \leqslant m$.*

Notice that a distinct alignment renaming does not impose a loss of generality. One can simply maintain the mapping $\{\acute{a}_1 \mapsto a_1, \ldots, \acute{a}_m \mapsto a_m\}$ and restore the original function symbols at any time. For a given hedge $\tilde{s}$, we denote this symbol renaming by $\tilde{s}\{\acute{a}_1 \mapsto a_1, \ldots, \acute{a}_m \mapsto a_m\}$.

Admissible alignments are related to generalization by the following theorem. It states that one does not loose generality by considering only generalizations that contain an admissible alignment as their skeleton.

Theorem 2.12. *Consider an alignment $\mathfrak{a}$ of two hedges $\tilde{s}_1$ and $\tilde{q}_1$. Let $\acute{a}_1\langle I_1, J_1\rangle \ldots \acute{a}_m\langle I_m, J_m\rangle$, $\tilde{s}_2$, and $\tilde{q}_2$ be a distinct alignment renaming of $\mathfrak{a}$, $\tilde{s}_1$, and $\tilde{q}_1$, by the fresh symbols $\acute{a}_1, \ldots, \acute{a}_m$. The given alignment $\mathfrak{a}$ is admissible iff there exists a generalization $\tilde{g}$ of $\tilde{s}_2$ and $\tilde{q}_2$ with $\mathcal{F}(\tilde{g}) = \{\acute{a}_1, \ldots, \acute{a}_m\}$.*

Notice that in Theorem 2.12 the alignment $\mathfrak{a}$ is admissible iff $\acute{a}_1\langle I_1, J_1\rangle \ldots \acute{a}_m\langle I_m, J_m\rangle$ is admissible, because collisions only depend on the positions and not on the function symbols.

Proof. Let $\mathfrak{a} = a_1\langle I_1, J_1\rangle \ldots a_m\langle I_m, J_m\rangle$ be an alignment of $\tilde{s}$ and $\tilde{q}$ such that for all $1 \leqslant k \leqslant m$ the function symbol a_k is unique in $\tilde{s}$ and unique in $\tilde{q}$.

($\Leftarrow$) Assume $\tilde{g}$ is a generalization of $\tilde{s}$ and $\tilde{q}$ with $\mathcal{F}(\tilde{g}) = \{a_1, \ldots, a_m\}$. We will prove by contradiction that there are no collisions in $\mathfrak{a}$ (see definition of admissible alignment). Furthermore, we assume that there are at least two elements in $\mathfrak{a}$ because the other cases are trivial by definition.

Case 1: Assume there is a collision at two elements of $\mathfrak{a}$. Then there exist $a_i, a_j \in \{a_1, \ldots, a_m\}$ such that a_i is an ancestor of a_j in $\tilde{s}$, while it is not an ancestor of a_j in $\tilde{q}$. We know that $\tilde{g}$ contains both symbols a_i and a_j.

Case 1.1: a_i is an ancestor of a_j in $\tilde{g}$. Then we have $a_i(\tilde{r}_1, t, \tilde{r}_2)$ being a subterm of $\tilde{g}$, where t is the term which contains a_j, and $\tilde{r}_1, \tilde{r}_2$ are arbitrary hedges. By assumption, there exists a substitution σ with $a_i, a_j \notin \mathcal{F}(\mathrm{Ran}(\sigma))$ such that a_i is not an ancestor of a_j in $\tilde{g}\sigma$. However, by the rule of substitution application $a_i(\tilde{r}_1, t, \tilde{r}_2)\sigma = a_i(\tilde{r}_1\sigma, t\sigma, \tilde{r}_2\sigma)$ the ancestor-descendant relation is preserved, which is a contradiction.

Case 1.2: a_i is not an ancestor of a_j in $\tilde{g}$. Then we have $(\tilde{r}_1, t_1, \tilde{r}_2, t_2, \tilde{r}_3)$ being a subhedge of $\tilde{g}$, where t_1 is the term which contains a_i, t_2 is the term which contains a_j, and $\tilde{r}_1, \tilde{r}_2, \tilde{r}_3$ are arbitrary hedges. By assumption, there exists a substitution σ with $a_i, a_j \notin \mathcal{F}(\mathrm{Ran}(\sigma))$ such that a_i is an ancestor of a_j in $\tilde{g}\sigma$, but this contradicts the rule of substitution application $(\tilde{r}_1, t_1, \tilde{r}_2, t_2, \tilde{r}_3)\sigma = (\tilde{r}_1\sigma, t_1\sigma, \tilde{r}_2\sigma, t_2\sigma, \tilde{r}_3\sigma)$ again.

Case 2: A collision appears at three elements. Let a_i, a_j, a_k be those elements. Without loss of generality, assume that a_i, a_j have a common ancestor $\hbar$ that is not an ancestor of a_k in $\tilde{s}$ and let a_j, a_k have a common ancestor $\jmath$ that is not an ancestor of a_i in $\tilde{q}$. By assumption, $\tilde{g}$ contains all three symbols exactly once. It follows that there are substitutions σ_1, σ_2 with $a_i, a_j, a_k \notin \mathcal{F}(\mathrm{Ran}(\sigma_1) \cup \mathrm{Ran}(\sigma_2))$, where $\tilde{g}\sigma_1 = \tilde{s}$ and $\tilde{g}\sigma_2 = \tilde{q}$. By assumption, we know that $\tilde{g}\sigma_1$ contains a subhedge $(t_{ij}, \tilde{s}_k)$, with t_{ij} being the term that contains the symbols $\hbar, a_i, a_j$, and $\tilde{s}_k$ being a hedge that contains the symbol a_k. This implies that $\tilde{g}$ contains either $\hbar$ or a context variable that can be instantiated to introduce $\hbar$. It follows that $\tilde{g}$ also contains a subhedge $(t'_{ij}, \tilde{s}'_k)$, with t'_{ij} being the term that contains the symbols a_i, a_j, and $\tilde{s}'_k$ being a hedge that contains the symbol a_k. Similarly, $\tilde{g}\sigma_2$ contains a subhedge $(\tilde{q}_i, t_{jk})$, with $\tilde{q}_i$ being a hedge that contains the

symbol a_i, and t_{jk} being the term that contains the symbols g, a_j, a_k. Further on, $\tilde{g}$ either contains g or a context variable, say $\tilde{X}$, which can be instantiated to introduce g. Let us call this metavariable χ. As g is an ancestor of both, a_j and a_k in $\tilde{q}$, χ has to be above t'_{ij}. This is a contradiction to the assumption that g is not an ancestor of a_i in $\tilde{q}$.

($\Rightarrow$) Proof by construction of an algorithm which computes such a generalization for a given admissible alignment of two hedges. In subsection 2.2.3 we describe this algorithm and prove its properties. □

Notice that in Theorem 2.12, by restoring the original symbol names in $\tilde{g}$, one obtains a generalization $\tilde{h}$ of the given input hedges.

Definition 2.26 (Supporting generalization)**.** *Consider an alignment* $\mathfrak{a} = a_1\langle I_1, J_1\rangle \ldots a_m\langle I_m, J_m\rangle$ *of two hedges* $\tilde{s}_1$ *and* $\tilde{q}_1$*. Let* $\acute{a}_1\langle I_1, J_1\rangle \ldots \acute{a}_m\langle I_m, J_m\rangle$*,* $\tilde{s}_2$*, and* $\tilde{q}_2$ *be a distinct alignment renaming of* $\mathfrak{a}$*,* $\tilde{s}_1$*, and* $\tilde{q}_1$*, by the fresh symbols* $\acute{a}_1, \ldots, \acute{a}_m$*. Then, for any generalization* $\tilde{g}$ *of* $\tilde{s}_2$ *and* $\tilde{q}_2$ *with* $\mathcal{F}(\tilde{g}) = \{\acute{a}_1, \ldots, \acute{a}_m\}$*, the generalization* $\tilde{g}\{\acute{a}_1 \mapsto a_1, \ldots, \acute{a}_m \mapsto a_m\}$ *of* $\tilde{s}_1$ *and* $\tilde{q}_1$ *is called a* supporting generalization *of* $\tilde{s}_1$ *and* $\tilde{q}_1$ *with respect to* $\mathfrak{a}$*.*

Example 2.24. *Let* $\tilde{s}$ *and* $\tilde{q}$ *be the hedges from Figure 2.13. Then* $(\tilde{x}, a(y, \tilde{Y}(b)), \tilde{z})$ *is a supporting generalization of* $\tilde{s}$ *and* $\tilde{q}$*, with respect to* $a\langle 2, 1\rangle b\langle 2{\cdot}2, 1{\cdot}1{\cdot}1{\cdot}1\rangle$*, while it is* not *a supporting generalization of* $\tilde{s}$ *and* $\tilde{q}$ *with respect to* $a\langle 1, 1\rangle b\langle 2{\cdot}2, 1{\cdot}1{\cdot}1{\cdot}1\rangle$*. The hedge* $(\tilde{X}(a(\tilde{x})), \tilde{Y}(b, b))$ *is a supporting generalization of* $\tilde{s}$ *and* $\tilde{q}$ *with respect to* $a\langle 1, 1{\cdot}1\rangle b\langle 2{\cdot}1, 2\rangle b\langle 2{\cdot}2, 3\rangle$*.*

Corollary 2.13. *For any admissible alignment of two hedges there exists a supporting generalization of those hedges with respect to the given alignment.*

Corollary 2.14. *For any generalization of two hedges there exists an admissible alignment of those hedges containing all the function symbols which appear in the generalization.*

2.2.3 Higher-Order Unranked Anti-Unification Algorithm $\mathfrak{G}^{2\mathcal{V}}_{\mathfrak{a}}$

Given two hedges $\tilde{s}$ and $\tilde{q}$ and their admissible alignment $\mathfrak{a}$, we aim at computing a least general supporting generalization of $\tilde{s}$ and $\tilde{q}$ with respect to $\mathfrak{a}$. Without restrictions, any algorithm that computes a complete set of supporting generalizations for two hedges and their admissible alignment would yield a *complete* unranked higher-order anti-unification algorithm. This follows by Theorem 2.12 if we run such an algorithm for all admissible alignments of two hedges.

We already saw in the first-order case that a universal algorithm without forbidding consecutive hedge variables is highly nondeterministic. This is also the case in higher-order anti-unification, hence it comes with no surprise that the least general supporting generalization of $\tilde{s}$ and $\tilde{q}$ with respect to $\mathfrak{a}$ is not unique.

Example 2.25. *For instance, for* (a, b, a) *and* (b, c) *with the admissible alignment* $b\langle 2, 1\rangle$*, we have two supporting lggs* $(\tilde{x}, b, \tilde{x}, \tilde{y})$ *and* $(\tilde{x}, b, \tilde{y}, \tilde{x})$*. For* $a(b(a))$ *and* $b(c)$ *with the admissible alignment* $b\langle 1{\cdot}1, 1\rangle$*, we also have two supporting lggs* $\tilde{X}(b(\tilde{X}(\tilde{y})))$ *and* $\tilde{X}(b(\tilde{Y}(\tilde{X}())))$*.*

Our goal is to compute a *unique* supporting generalization for two hedges and an admissible alignment which captures the common structure of both input hedges. Therefore, we introduce some restrictions to ensure uniqueness of a supporting generalization and call such generalizations *rigid*, overloading the notion of Definition 2.12. By forbidding consecutive hedge variables for the first case of Example 2.25 we get one *rigid* lgg $(\tilde{x}, b, \tilde{y})$ which is a unique supporting generalization. Since we introduce context variables in addition to first-order variables, the restriction has to be extended for the "vertical direction". The following two definitions introduce this restrictions.

Definition 2.27 (Rigid hedge)**.** *A hedge $\tilde{s}$ is* rigid *if the following conditions hold:*

1. *No context variable in $\tilde{s}$ applies to the empty hedge.*
2. *$\tilde{s}$ doesn't contain consecutive hedge variables.*
3. *$\tilde{s}$ doesn't contain vertical chains of (context)[†] variables.*
4. *$\tilde{s}$ doesn't contain context variables with a hedge variable as the first or the last argument (i.e., no subterms of the form $\tilde{X}(\tilde{x}, \dots)$ and $\tilde{X}(\dots, \tilde{x})$).*

Definition 2.28 (Rigid generalization)**.** *Given two variable-disjoint hedges $\tilde{s}, \tilde{q}$ and their admissible alignment $\mathfrak{a}$, a rigid hedge $\tilde{g}$ is called a* rigid generalization *of $\tilde{s}$ and $\tilde{q}$ with respect to $\mathfrak{a}$, if $\tilde{g}$ is a supporting generalization of $\tilde{s}$ and $\tilde{q}$ with respect to $\mathfrak{a}$ so that:*

5. *There are substitutions σ, ϑ with $\tilde{g}\sigma = \tilde{s}$ and $\tilde{g}\vartheta = \tilde{q}$ such that all the contexts in σ and ϑ are singleton contexts.*

Intuitively, *item 1* and *item 5* together forbid that, in a rigid generalization, context variables capture horizontal disagreements of the input hedges. We want to use hedge variables for generalizing horizontal disagreements. For instance, consider the hedges $(f(b), a)$ and (b, c) with the admissible alignment $b\langle 1{\cdot}1, 1\rangle$. The three supporting generalizations $\tilde{X}(b)$, $(\tilde{X}(b), \tilde{Y}())$ and $(\tilde{X}(b), \tilde{y})$ are pairwise distinct in the relation $\simeq$. Nevertheless, the latter one tells us more about the common structure. Those two restrictions are solely for the purpose of picking one (the most natural) case out of some equi-general solutions, and we provide the user some additional information about the solution we compute.

The last three restrictions of Definition 2.27 are needed to compute a unique supporting generalization, as discussed above.

Example 2.26. *For instance, $\tilde{X}(a, b)$ is a rigid generalization of $f(g(a, b, c))$ and (a, b) with respect to $a\langle 1{\cdot}1{\cdot}1, 1\rangle b\langle 1{\cdot}1{\cdot}2, 2\rangle$, while $\tilde{X}(a, b, \tilde{x})$ and $\tilde{X}(\tilde{Y}(a, b))$ are not rigid generalizations.*

Definition 2.29 (Rigid lgg)**.** *A rigid generalization $\tilde{g}$ of $\tilde{s}$ and $\tilde{q}$ with respect to $\mathfrak{a}$ is called a* least general rigid generalization *(rigid lgg) of $\tilde{s}$ and $\tilde{q}$ with respect to $\mathfrak{a}$, if there is no rigid generalization $\tilde{h}$ of $\tilde{s}$ and $\tilde{q}$ with respect to $\mathfrak{a}$ which satisfies $\tilde{g} \prec \tilde{h}$.*

Note that two hedges might have a supporting generalization which is strictly less general than their rigid lgg with respect to the same admissible alignment. Example 2.27 illustrates such a case.

[†]Vertical chains that consist of a context variable and a hedge variable are barred by item 4.

Example 2.27. *For instance,* $\tilde{X}(a) \prec \tilde{X}(\tilde{X}(a))$ *and both of them are supporting generalizations of* $f(f(a))$ *and* $g(g(g(g(a))))$ *with respect to* $a\langle 1{\cdot}1{\cdot}1, 1{\cdot}1{\cdot}1{\cdot}1{\cdot}1{\cdot}1\rangle$*, but only* $\tilde{X}(a)$ *is a rigid generalization.*

Based on Definition 2.28, we present a rule-based anti-unification algorithm that solves the following problem:

Given: Two hedges $\tilde{s}$ and $\tilde{q}$ and their admissible alignment $\mathfrak{a}$.

Find: A least general rigid generalization $\tilde{g}$ of $\tilde{s}$ and $\tilde{q}$ with respect to $\mathfrak{a}$.

The algorithm uses the following data structure that consists of three parts. A horizontal part of the form $\tilde{x}\colon \tilde{s} \triangleq \tilde{q}$, a vertical part $\tilde{X}\colon \tilde{c} \triangleq \tilde{d}$, and an admissible alignment. Intuitively, the horizontal and the vertical part together represent a decomposition of the hedges $\tilde{c}[\tilde{s}]$ and $\tilde{d}[\tilde{q}]$.

Definition 2.30 (Anti-unification equation)**.** *An* anti-unification equation*, AUE in short, is a triple of the form* $\tilde{x}\colon \tilde{s} \triangleq \tilde{q};\ \tilde{X}\colon \tilde{c} \triangleq \tilde{d};\ \mathfrak{a}$*, where*

- $\tilde{x}$ *is a hedge variable and* $\tilde{s}, \tilde{q}$ *are hedges,*
- $\tilde{X}$ *is a context variable and* $\tilde{c}, \tilde{d}$ *are contexts,*
- $\mathfrak{a}$ *is an admissible alignment of* $\tilde{s}$ *and* $\tilde{q}$*.*

The algorithm consists of eight transformation rules that transform tuples by rule application into tuples of the same form.

Definition 2.31 (Rigid higher-order anti-unification algorithm)**.** *The rigid unranked higher-order anti-unification algorithm is formulated in a rule-based way working on tuples* $P;\ S;\ \sigma$*, where*

- P *is the set of AUEs to be solved (the problem set);*
- S *is a set of already solved AUEs (the store);*
- σ *is a substitution (computed so far) mapping variables to hedges and contexts.*
- *for all pairs of AUEs* $\{\tilde{x}\colon \tilde{s}_1 \triangleq \tilde{q}_1;\ \tilde{X}\colon \tilde{c}_1 \triangleq \tilde{d}_1;\ \mathfrak{a}_1,\ \tilde{y}\colon \tilde{s}_2 \triangleq \tilde{q}_2;\ \tilde{Y}\colon \tilde{c}_2 \triangleq \tilde{d}_2;\ \mathfrak{a}_2\} \subseteq P \cup S$ *holds* $\tilde{x} \neq \tilde{y}$ *and* $\tilde{X} \neq \tilde{Y}$*.*

We call such a tuple a state *and the algorithm is called* $\mathfrak{G}^{2\mathcal{V}}_{\mathfrak{a}}$*, where* $\mathfrak{a}$ *indicates the skeleton that appears in the generalization and* $2\mathcal{V}$ *indicates that two different kinds of variables are considered. The eight transformation rules of the algorithm, which are defined below, operate on states.*

As all the AUEs in S have the empty alignment, we write $\tilde{x}\colon \tilde{s} \triangleq \tilde{q};\ \tilde{X}\colon \tilde{c} \triangleq \tilde{d}$ instead of $\tilde{x}\colon \tilde{s} \triangleq \tilde{q};\ \tilde{X}\colon \tilde{c} \triangleq \tilde{d};\ \mathfrak{e}$ for an AUE of S. In the transformation rules below, we use the symbols $\tilde{Y}, \tilde{Z}$ for fresh context variables and $\tilde{y}, \tilde{z}$ for fresh hedge variables. The symbol $\uplus$ stands for disjoint union. Furthermore, i^{--} denotes $i-1$ and i^{++} denotes $i+1$.

Spl-H: **Split Hedge**

$\{\tilde{x}\colon \tilde{s} \triangleq \tilde{q};\ \tilde{X}\colon \tilde{c} \triangleq \tilde{d};\ a_1\langle i_1{\cdot}I_1, j_1{\cdot}J_1\rangle \dots a_k\langle i_k{\cdot}I_k, j_k{\cdot}J_k\rangle$
$a_{k+1}\langle i_{k+1}{\cdot}I_{k+1}, j_{k+1}{\cdot}J_{k+1}\rangle \dots a_m\langle i_m{\cdot}I_m, j_m{\cdot}J_m\rangle\} \cup P;\ S;\ \sigma \implies$
$\{\tilde{y}\colon \tilde{s}|_{i_1}^{i_k} \triangleq \tilde{q}|_{j_1}^{j_k};\ \tilde{Y}\colon \circ \triangleq \circ;\ a_1\langle (i_1 - i_1^{--}){\cdot}I_1, (j_1 - j_1^{--}){\cdot}J_1\rangle \dots$
$a_k\langle (i_k - i_1^{--}){\cdot}I_k, (j_k - j_1^{--}){\cdot}J_k\rangle\} \cup$
$\{\tilde{z}\colon \tilde{s}|_{i_k^{++}}^{i_m} \triangleq \tilde{q}|_{j_k^{++}}^{j_m};\ \tilde{Z}\colon \circ \triangleq \circ;\ a_{k+1}\langle (i_{k+1} - i_k){\cdot}I_{k+1}, (j_{k+1} - j_k){\cdot}J_{k+1}\rangle \dots$
$a_m\langle (i_m - i_k){\cdot}I_m, (j_m - j_k){\cdot}J_m\rangle\} \cup P;$
$\{\tilde{x}\colon \epsilon \triangleq \epsilon;\ \tilde{X}\colon \tilde{c}[\tilde{s}|_1^{i_1^{--}}, \circ, \tilde{s}|_{i_m^{++}}^{|\tilde{s}|}] \triangleq \tilde{d}[\tilde{q}|_1^{j_1^{--}}, \circ, \tilde{q}|_{j_m^{++}}^{|\tilde{q}|}]\} \cup S;\ \sigma\{\tilde{x} \mapsto (\tilde{Y}(\tilde{y}), \tilde{Z}(\tilde{z}))\},$

If $i_1 \neq i_{k+1}$ and $j_1 \neq j_{k+1}$, and, moreover, $i_1 = i_k$ or $j_1 = j_k$, for $1 \leqslant k < m$.

Abs-L: **Abstract Left Context**

$\{\tilde{x}\colon (\tilde{s}_l, \hbar(\tilde{s}), \tilde{s}_r) \triangleq \tilde{q};\ \tilde{X}\colon \tilde{c} \triangleq \tilde{d};\ a_1\langle i{\cdot}I_1, J_1\rangle \dots a_m\langle i{\cdot}I_m, J_m\rangle\} \cup P;\ S;\ \sigma \implies$
$\{\tilde{x}\colon \tilde{s} \triangleq \tilde{q};\ \tilde{X}\colon \tilde{c}[\tilde{s}_l, \hbar(\circ), \tilde{s}_r] \triangleq \tilde{d};\ a_1\langle I_1, J_1\rangle \dots a_m\langle I_m, J_m\rangle\} \cup P;\ S;\ \sigma,$

where $I_1 \neq \epsilon$, $\hbar(\tilde{s}) = (\tilde{s}_l, \hbar(\tilde{s}), \tilde{s}_r)|_i$, and $\tilde{s}_l, \tilde{s}_r$ are hedges.

Abs-R: **Abstract Right Context**

$\{\tilde{x}\colon \tilde{s} \triangleq (\tilde{q}_l, \hbar(\tilde{q}), \tilde{q}_r);\ \tilde{X}\colon \tilde{c} \triangleq \tilde{d};\ a_1\langle I_1, j{\cdot}J_1\rangle \dots a_m\langle I_m, j{\cdot}J_m\rangle\} \cup P;\ S;\ \sigma \implies$
$\{\tilde{x}\colon \tilde{s} \triangleq \tilde{q};\ \tilde{X}\colon \tilde{c} \triangleq \tilde{d}[\tilde{q}_l, \hbar(\circ), \tilde{q}_r];\ a_1\langle I_1, J_1\rangle \dots a_m\langle I_m, J_m\rangle\} \cup P;\ S;\ \sigma,$

where $J_1 \neq \epsilon$, $\hbar(\tilde{q}) = (\tilde{q}_l, \hbar(\tilde{q}), \tilde{q}_r)|_j$, and $\tilde{q}_l, \tilde{q}_r$ are hedges.

App-A: **Apply Alignment**

$\{\tilde{x}\colon (\tilde{s}_l, a_1(\tilde{s}), \tilde{s}_r) \triangleq (\tilde{q}_l, a_1(\tilde{q}), \tilde{q}_r);\ \tilde{X}\colon \tilde{c} \triangleq \tilde{d};$
$a_1\langle i, j\rangle a_2\langle i{\cdot}I_2, j{\cdot}J_2\rangle \dots a_m\langle i{\cdot}I_m, j{\cdot}J_m\rangle\} \cup P;\ S;\ \sigma \implies$
$\{\tilde{y}\colon \tilde{s} \triangleq \tilde{q};\ \tilde{Y}\colon \circ \triangleq \circ;\ a_2\langle I_2, J_2\rangle \dots a_m\langle I_m, J_m\rangle\} \cup P;$
$\{\tilde{x}\colon \epsilon \triangleq \epsilon;\ \tilde{X}\colon \tilde{c}[\tilde{s}_l, \circ, \tilde{s}_r] \triangleq \tilde{d}[\tilde{q}_l, \circ, \tilde{q}_r]\} \cup S;\ \sigma\{\tilde{x} \mapsto a_1(\tilde{Y}(\tilde{y}))\},$

where $a_1(\tilde{s}), a_1(\tilde{q})$ are the terms at the positions i, j and $\tilde{s}_l, \tilde{s}_r, \tilde{q}_l, \tilde{q}_r$ are hedges.

Sol-H: **Solve Hedge**

$\{\tilde{x}\colon \tilde{s} \triangleq \tilde{q};\ \tilde{X}\colon \circ \triangleq \circ;\ \mathfrak{e}\} \cup P;\ S;\ \sigma \implies P;\ \{\tilde{x}\colon \tilde{s} \triangleq \tilde{q};\ \tilde{X}\colon \circ \triangleq \circ\} \cup S;\ \sigma\{\tilde{X} \mapsto \circ\}.$

Res-C: **Restore Context**

$P;\ \{\tilde{x}\colon \epsilon \triangleq \epsilon;\ \tilde{X}\colon (\tilde{s}_l, \dot{c}, \tilde{s}_r) \triangleq (\tilde{q}_l, \dot{d}, \tilde{q}_r)\} \cup S;\ \sigma \implies$
$P;\ \{\tilde{x}\colon \epsilon \triangleq \epsilon;\ \tilde{X}\colon \dot{c} \triangleq \dot{d},\ \tilde{y}\colon \tilde{s}_l \triangleq \tilde{q}_l;\ \tilde{Y}\colon \circ \triangleq \circ,\ \tilde{z}\colon \tilde{s}_r \triangleq \tilde{q}_r;\ \tilde{Z}\colon \circ \triangleq \circ\} \cup S;$
$\sigma\{\tilde{X} \mapsto (\tilde{y}, \tilde{X}(\circ), \tilde{z})\},$

if not $\epsilon = \tilde{s}_l = \tilde{s}_r = \tilde{q}_l = \tilde{q}_r$. $\dot{c}, \dot{d}$ are singleton contexts.

Mer-S: **Merge Store**

$P;\ \{\tilde{x}_1\colon \tilde{s} \triangleq \tilde{q};\ \tilde{X}_1\colon \tilde{c} \triangleq \tilde{d},\ \tilde{x}_2\colon \tilde{s} \triangleq \tilde{q};\ \tilde{X}_2\colon \tilde{c} \triangleq \tilde{d}\} \cup S;\ \sigma \implies$
$P;\ \{\tilde{x}_1\colon \tilde{s} \triangleq \tilde{q};\ \tilde{X}_1\colon \tilde{c} \triangleq \tilde{d}\} \cup S;\ \sigma\{\tilde{x}_2 \mapsto \tilde{x}_1, \tilde{X}_2 \mapsto \tilde{X}_1(\circ)\}.$

Clr-S: **Clear Store**

$P;\ \{\tilde{x}\colon \epsilon \triangleq \epsilon;\ \tilde{X}\colon \circ \triangleq \circ\} \cup S;\ \sigma \implies P;\ S;\ \sigma\{\tilde{x} \mapsto \epsilon, \tilde{X} \mapsto \circ\}.$

To compute a generalization of two hedges $\tilde{s}$ and $\tilde{q}$ with respect to an admissible alignment $\mathfrak{a}$, the procedure starts with the *initial state* $\{\tilde{x}\colon \tilde{s} \triangleq \tilde{q};\ \tilde{X}\colon \circ \triangleq \circ;\ \mathfrak{a}\};\ \varnothing;\ Id$,

where $\tilde{x}$ and $\tilde{X}$ are fresh variables, and applies the rules exhaustively. A state where no more rule is applicable is called *final state*. In the final state, the problem set is empty. We will prove termination, soundness, and completeness of $\mathfrak{G}_{\mathfrak{a}}^{2\mathcal{V}}$, as well as uniqueness (modulo $\simeq$) of the final state in subsection 2.2.6. The unique final state for two hedges $\tilde{s}$ and $\tilde{q}$ with respect to an admissible alignment $\mathfrak{a}$ is denoted by $\mathfrak{G}_{\mathfrak{a}}^{2\mathcal{V}}(\tilde{X}(\tilde{x})\colon \tilde{s} \triangleq \tilde{q})$, where $\tilde{x}$ and $\tilde{X}$ are the fresh generalization variables. The rigid lgg that corresponds to a final state $\varnothing; S; \sigma = \mathfrak{G}_{\mathfrak{a}}^{2\mathcal{V}}(\tilde{X}(\tilde{x})\colon \tilde{s} \triangleq \tilde{q})$ can be obtained by $\tilde{X}(\tilde{x})\sigma$. The store S keeps track of already solved AUEs in order to generalize the same AUEs in the same way. In the final state it contains all the differences of the input hedges.

Definition 2.32. *We define two substitutions obtained by a set S of AUEs:*

$$\sigma_{L}(S) ::= \{\tilde{x} \mapsto \tilde{s}, \tilde{X} \mapsto \tilde{c} \mid \tilde{x}\colon \tilde{s} \triangleq \tilde{q};\ \tilde{X}\colon \tilde{c} \triangleq \tilde{d};\ \mathfrak{a} \in S\}$$
$$\sigma_{R}(S) ::= \{\tilde{x} \mapsto \tilde{q}, \tilde{X} \mapsto \tilde{d} \mid \tilde{x}\colon \tilde{s} \triangleq \tilde{q};\ \tilde{X}\colon \tilde{c} \triangleq \tilde{d};\ \mathfrak{a} \in S\}$$

Let $\varnothing; S; \sigma = \mathfrak{G}_{\mathfrak{a}}^{2\mathcal{V}}(\tilde{X}(\tilde{x})\colon \tilde{s} \triangleq \tilde{q})$. In Theorem 2.21 (Soundness) we show that the algorithm $\mathfrak{G}_{\mathfrak{a}}^{2\mathcal{V}}$ is coherent so that $\tilde{X}(\tilde{x})\sigma\sigma_{\mathrm{L}}(S) = \tilde{s}$ and $\tilde{X}(\tilde{x})\sigma\sigma_{\mathrm{R}}(S) = \tilde{q}$.

2.2.4 Explanation of the Transformation Rules of $\mathfrak{G}_{\mathfrak{a}}^{2\mathcal{V}}$

Before illustrating $\mathfrak{G}_{\mathfrak{a}}^{2\mathcal{V}}$ on some examples and discussing its properties, we briefly explain informally what the rules do. At each step, each AUE $\tilde{x}\colon \tilde{s} \triangleq \tilde{q};\ \tilde{X}\colon \tilde{c} \triangleq \tilde{d};\ \mathfrak{a}$ in P represents the hedges $\tilde{c}[\tilde{s}]$ and $\tilde{d}[\tilde{q}]$ which are to be generalized such that the final generalization contains the function symbols from $\mathfrak{a}$. They are split according to the occurrences of alignment elements: All symbols from $\mathfrak{a}$ are in $\tilde{s}$ and $\tilde{q}$. None of them appear in $\tilde{c}$ and $\tilde{d}$. Such an AUE can be transformed by one of the first four rules: Spl-H, Abs-L, Abs-R, or App-A. The eventual goal of these transformations is to reach the occurrences of the first alignment element in $\tilde{s}$ and $\tilde{q}$. In the course of the transformation, $\tilde{c}$ and $\tilde{d}$ are getting extended with contexts above those occurrences.

Spl-H. When the symbols in $\mathfrak{a}$ are distributed in more than one term both in $\tilde{s}$ and in $\tilde{q}$, then we use the Spl-H rule to select subhedges of $\tilde{s}$ and $\tilde{q}$ which contain all the alignment elements. (The other parts of $\tilde{s}$ and $\tilde{q}$ are moved to the store, since they will not contribute a symbol to the generalization.) Furthermore, by this rule, each of these subhedges are split into two smaller subhedges: From the $\tilde{s}$ side these are $\tilde{s}|_{i_1}^{i_k}$ and $\tilde{s}|_{i_k^{++}}^{i_m}$, and from the $\tilde{q}$ side they are $\tilde{q}|_{j_1}^{j_k}$ and $\tilde{q}|_{j_k^{++}}^{j_m}$. The split point k is decided by the following criteria:

- $\tilde{s}|_{i_1}^{i_k}$ and $\tilde{q}|_{j_1}^{j_k}$ contain the first $k > 0$ elements of $\mathfrak{a}$.
- $\tilde{s}|_{i_k^{++}}^{i_m}$ and $\tilde{q}|_{j_k^{++}}^{j_m}$ contain the elements of $\mathfrak{a}$ starting from $k+1$. There exists at least one such element.
- $\tilde{s}|_{i_1}^{i_k}$ or $\tilde{q}|_{j_1}^{j_k}$ is a term (a singleton hedge), and the $k+1$'st element of $\mathfrak{a}$ does not belong to it.

The process will continue by generalizing $\tilde{s}|_{i_1}^{i_k}$ and $\tilde{q}|_{j_1}^{j_k}$ with respect to the first k-element prefix of $\mathfrak{a}$, and generalizing $\tilde{s}|_{i_k^{++}}^{i_m}$ and $\tilde{q}|_{j_k^{++}}^{j_m}$ with respect to the elements

of $\mathfrak{a}$ starting from $k+1$. Note that in the next step Spl-H is not applicable to the AUE with $\tilde{s}|_{i_1}^{i_k}$ and $\tilde{q}|_{j_1}^{j_k}$. This is because at least one of them is a single term which completely contains the alignment elements. Therefore either Abs-L, Abs-R, or App-A applies.

Example 2.28. *Consider the hedges* $(g(a), f(a, g(b)), c, g(b), e)$ *and* $(e, e, h(a, e), f(b), a, c, d, b)$ *and the admissible alignment* $a\langle 2{\cdot}1, 3{\cdot}1\rangle b\langle 2{\cdot}2{\cdot}1, 4{\cdot}1\rangle c\langle 3, 6\rangle b\langle 4{\cdot}1, 8\rangle$ *of them.*

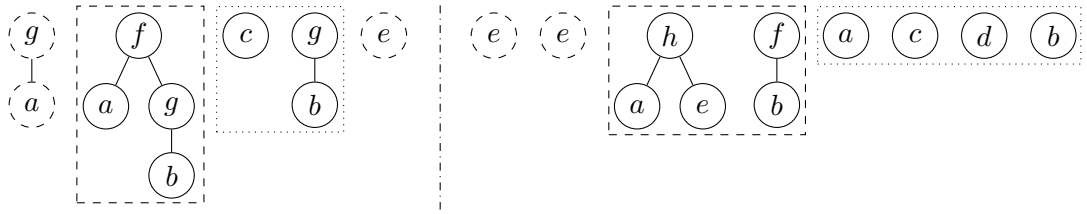

Figure 2.15: The hedges from Example 2.28.

The dashed nodes in Figure 2.15 denote the parts which are moved into the store. The dashed rectangle denotes $\tilde{s}|_{i_1}^{i_k}$ and $\tilde{q}|_{j_1}^{j_k}$ and the dotted one $\tilde{s}|_{i_k^{++}}^{i_m}$ and $\tilde{q}|_{j_k^{++}}^{j_m}$.

Abs-L, Abs-R. When all symbols in $\mathfrak{a}$ belong to one term in $\tilde{s}$ or in $\tilde{q}$ (or maybe both), but the root of that term is *not* the symbol a_1 from the first element of $\mathfrak{a}$, then an attempt is made to get deeper to that term, to reach the subterm whose top symbol is the a_1 from $\mathfrak{a}$. This descent is carried out by Abs-L or Abs-R, depending whether we are searching for the subterm with a_1 in the top in $\tilde{s}$ or in $\tilde{q}$.

To illustrate the rule Abs-L which serves as a representative of both, we use the above example and apply Abs-L to the AUE $\tilde{x}\colon f(a, g(b)) \triangleq (h(a, e), f(b))$; $\tilde{X}\colon \circ \triangleq \circ$; $a\langle 1{\cdot}1, 1{\cdot}1\rangle b\langle 1{\cdot}2{\cdot}1, 2{\cdot}1\rangle$ following an Spl-H application. The Abs-L transformation decomposes the left term $f(a, g(b))$ into a context $f(\circ)$ and a hedge $(a, g(b))$, resulting in the AUE $\tilde{x}\colon (a, g(b)) \triangleq (h(a, e), f(b))$; $\tilde{X}\colon f(\circ) \triangleq \circ$; $a\langle 1, 1{\cdot}1\rangle b\langle 2{\cdot}1, 2{\cdot}1\rangle$. Figure 2.16 demonstrates this decomposition step.

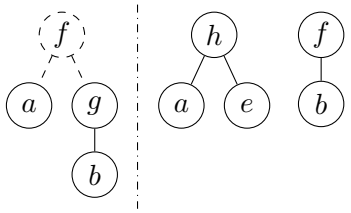

Figure 2.16: $f(a, g(b))$ and $(h(a, e), f(b))$.

App-A. When all symbols in $\mathfrak{a}$ belong to one term in $\tilde{s}$ and one term in $\tilde{q}$, and these terms have the same root symbol which is exactly the a_1 from the first element of $\mathfrak{a}$, then a_1 is moved to the generalization. This is what the App-A rule does. The process will continue with generalizing the hedges under the occurrences of a_1 in $\tilde{s}$ and $\tilde{q}$.

Sol-H. When the alignment is empty in $\tilde{x}\colon \tilde{s} \triangleq \tilde{q}$; $\tilde{X}\colon \tilde{c} \triangleq \tilde{d}$; $\mathfrak{e}$ in P, then the hedge there will not contribute a symbol in the generalization. Moreover, both $\tilde{c}$ and $\tilde{d}$ are holes, because only App-A can make the alignment empty, and it makes the contexts in

the obtained AUE the hole. Such AUEs are considered solved, as their generalization is just $\tilde{x}$ they contain. They should be put in the store, which keeps information about the differences between the hedges to be generalized. At the same time, the context variable $\tilde{X}$ can be deleted, as it just stand for the hole. This is what the Sol-H rule does.

Transformation of the store. The other three rules work on the store. Clr-S removes the empty AUE from the store and eliminates the corresponding variables form the generalization. Mer-S guarantees that the same AUEs are generalized with the same variables, making sure that the same differences in the input hedges are generalized uniformly. Finally, the Res-C rule guarantees that each context variable in the generalization generalizes singleton contexts in the input hedges: A property required for rigid generalizations.

2.2.5 Illustration of the Algorithm $\mathfrak{G}_{\mathfrak{a}}^{2\mathcal{V}}$

First, we illustrate the algorithm $\mathfrak{G}_{\mathfrak{a}}^{2\mathcal{V}}$ step-by-step on some small examples. Then we show how $\mathfrak{G}_{\mathfrak{a}}^{2\mathcal{V}}$ can be used to detect software clones that can not be detected by first-order anti-unification algorithms.

Example 2.29. *We illustrate the transformation steps performed by $\mathfrak{G}_{\mathfrak{a}}^{2\mathcal{V}}$ on the hedges $\tilde{s} = f(a, f(b,b))$ and $\tilde{q} = (b, f(a,b), b)$ and their admissible alignment $\mathfrak{a} = f\langle 1,2\rangle\, a\langle 1{\cdot}1, 2{\cdot}1\rangle b\langle 1{\cdot}2{\cdot}1, 2{\cdot}2\rangle$. In the substitution, we only keep the mappings for the two generalization variables $\tilde{x}$ and $\tilde{X}$ of the initial AUE.*

$$
\begin{aligned}
& \{\tilde{x}\colon f(a,f(b,b)) \triangleq (b,f(a,b),b);\ \tilde{X}\colon \circ \triangleq \circ;\ f\langle 1,2\rangle a\langle 1{\cdot}1, 2{\cdot}1\rangle b\langle 1{\cdot}2{\cdot}1, 2{\cdot}2\rangle\};\ \varnothing;\ \mathit{Id} \\
\Longrightarrow_{\textsf{App-A}}\ & \{\tilde{y}_1\colon (a,f(b,b)) \triangleq (a,b);\ \tilde{Y}_1\colon \circ \triangleq \circ;\ a\langle 1,1\rangle b\langle 2{\cdot}1, 2\rangle\}; \\
& \{\tilde{x}\colon \epsilon \triangleq \epsilon;\ \tilde{X}\colon \circ \triangleq (b,\circ,b)\};\ \{\tilde{x} \mapsto f(\tilde{Y}_1(\tilde{y}_1))\} \\
\Longrightarrow^{\textsf{Res-C}}_{\textsf{Clr-S}}\ & \{\tilde{y}_1\colon (a,f(b,b)) \triangleq (a,b);\ \tilde{Y}_1\colon \circ \triangleq \circ;\ a\langle 1,1\rangle b\langle 2{\cdot}1, 2\rangle\}; \\
& \{\tilde{z}_1\colon \epsilon \triangleq b;\ \tilde{Z}_1\colon \circ \triangleq \circ, \tilde{z}_2\colon \epsilon \triangleq b;\ \tilde{Z}_2\colon \circ \triangleq \circ\};\ \{\tilde{x} \mapsto f(\tilde{Y}_1(\tilde{y}_1)), \tilde{X} \mapsto (\tilde{z}_1,\circ,\tilde{z}_2)\} \\
\Longrightarrow_{\textsf{Mer-S}}\ & \{\tilde{y}_1\colon (a,f(b,b)) \triangleq (a,b);\ \tilde{Y}_1\colon \circ \triangleq \circ;\ a\langle 1,1\rangle b\langle 2{\cdot}1, 2\rangle\}; \\
& \{\tilde{z}_1\colon \epsilon \triangleq b;\ \tilde{Z}_1\colon \circ \triangleq \circ\};\ \{\tilde{x} \mapsto f(\tilde{Y}_1(\tilde{y}_1)), \tilde{X} \mapsto (\tilde{z}_1,\circ,\tilde{z}_1)\} \\
\Longrightarrow^{\textsf{Spl-H}}_{\textsf{Clr-S}}\ & \{\tilde{y}_2\colon a \triangleq a;\ \tilde{Y}_2\colon \circ \triangleq \circ;\ a\langle 1,1\rangle,\ \tilde{y}_3\colon (f(b,b)) \triangleq b;\ \tilde{Y}_3\colon \circ \triangleq \circ;\ b\langle 1{\cdot}1, 1\rangle\}; \\
& \{\tilde{z}_1\colon \epsilon \triangleq b;\ \tilde{Z}_1\colon \circ \triangleq \circ\};\ \{\tilde{x} \mapsto f(\tilde{Y}_2(\tilde{y}_2), \tilde{Y}_3(\tilde{y}_3)), \tilde{X} \mapsto (\tilde{z}_1,\circ,\tilde{z}_1)\} \\
\Longrightarrow^{\textsf{App-A}}_{\textsf{Clr-S}}\ & \{\tilde{y}_4\colon \epsilon \triangleq \epsilon;\ \tilde{Y}_4\colon \circ \triangleq \circ;\ \mathfrak{e},\ \tilde{y}_3\colon (f(b,b)) \triangleq (b);\ \tilde{Y}_3\colon \circ \triangleq \circ;\ b\langle 1{\cdot}1, 1\rangle\}; \\
& \{\tilde{z}_1\colon \epsilon \triangleq b;\ \tilde{Z}_1\colon \circ \triangleq \circ\};\ \{\tilde{x} \mapsto f(a(\tilde{Y}_4(\tilde{y}_4)), \tilde{Y}_3(\tilde{y}_3)), \tilde{X} \mapsto (\tilde{z}_1,\circ,\tilde{z}_1)\} \\
\Longrightarrow^{\textsf{Sol-H}}_{\textsf{Clr-S}}\ & \{\tilde{y}_3\colon f(b,b) \triangleq b;\ \tilde{Y}_3\colon \circ \triangleq \circ;\ b\langle 1{\cdot}1, 1\rangle\}; \\
& \{\tilde{z}_1\colon \epsilon \triangleq b;\ \tilde{Z}_1\colon \circ \triangleq \circ\};\ \{\tilde{x} \mapsto f(a, \tilde{Y}_3(\tilde{y}_3)), \tilde{X} \mapsto (\tilde{z}_1,\circ,\tilde{z}_1)\} \\
\Longrightarrow_{\textsf{Abs-L}}\ & \{\tilde{y}_3\colon (b,b) \triangleq b;\ \tilde{Y}_3\colon f(\circ) \triangleq \circ;\ b\langle 1,1\rangle\}; \\
& \{\tilde{z}_1\colon \epsilon \triangleq b;\ \tilde{Z}_1\colon \circ \triangleq \circ\};\ \{\tilde{x} \mapsto f(a, \tilde{Y}_3(\tilde{y}_3)), \tilde{X} \mapsto (\tilde{z}_1,\circ,\tilde{z}_1)\} \\
\Longrightarrow_{\textsf{App-A}}\ & \{\tilde{y}_5\colon \epsilon \triangleq \epsilon;\ \tilde{Y}_5\colon \circ \triangleq \circ;\ \mathfrak{e}\};\ \{\tilde{y}_3\colon \epsilon \triangleq \epsilon;\ \tilde{Y}_3\colon f(\circ,b) \triangleq \circ,\ \tilde{z}_1\colon \epsilon \triangleq b;\ \tilde{Z}_1\colon \circ \triangleq \circ\}; \\
& \{\tilde{x} \mapsto f(a, \tilde{Y}_3(b(\tilde{Y}_5(\tilde{y}_5)))), \tilde{X} \mapsto (\tilde{z}_1,\circ,\tilde{z}_1)\} \\
\Longrightarrow^{\textsf{Sol-H}}_{\textsf{Clr-S}}\ & \varnothing;\{\tilde{y}_3\colon \epsilon \triangleq \epsilon;\ \tilde{Y}_3\colon f(\circ,b) \triangleq \circ,\ \tilde{z}_1\colon \epsilon \triangleq b;\ \tilde{Z}_1\colon \circ \triangleq \circ\}; \\
& \{\tilde{x} \mapsto f(a, \tilde{Y}_3(b)), \tilde{X} \mapsto (\tilde{z}_1,\circ,\tilde{z}_1)\}.
\end{aligned}
$$

$\tilde{X}(\tilde{x})\sigma = (\tilde{z}_1, f(a, \tilde{Y}_3(b)), \tilde{z}_1)$ *generalizes* $\tilde{s}$ *and* $\tilde{q}$ *with respect to* $\mathfrak{a}$. *From the store* S *we can read* $\sigma_{_L}(S) = \{\tilde{z}_1 \mapsto \epsilon, \tilde{Y}_3 \mapsto f(\circ, b), \ldots\}$ *and* $\sigma_{_R}(S) = \{\tilde{z}_1 \mapsto b, \tilde{Y}_3 \mapsto \circ, \ldots\}$. *Then we have* $\tilde{X}(\tilde{x})\sigma\sigma_{_L}(S) = \tilde{s}$ *and* $\tilde{X}(\tilde{x})\sigma\sigma_{_R}(S) = \tilde{q}$.

Example 2.30. *We illustrate the computation of a rigid lgg with respect to the given alignment* $a\langle 1{\cdot}1, 1\rangle f\langle 2, 2\rangle g\langle 2{\cdot}1{\cdot}1, 2{\cdot}1\rangle a\langle 2{\cdot}1{\cdot}1{\cdot}1, 2{\cdot}1{\cdot}1\rangle c\langle 2{\cdot}1{\cdot}2, 2{\cdot}2\rangle$ *of the two hedges* $\tilde{s} = (\tilde{U}(a), f(\tilde{U}(g(a, b, b), c), b, b))$ *and* $\tilde{q} = (a, f(g(a, d), c, d))$. *Notice that the considered alignment is of longest possible length for the two input hedges. The symbol* $\tilde{U}$ *denotes a context variable. All the other symbols are function symbols.*

$$
\begin{aligned}
& \{\tilde{x}\colon (\tilde{U}(a), f(\tilde{U}(g(a,b,b),c),b,b)) \triangleq (a, f(g(a,d),c,d));\\
& \tilde{X}\colon \circ \triangleq \circ;\ a\langle 1{\cdot}1, 1\rangle f\langle 2, 2\rangle g\langle 2{\cdot}1{\cdot}1, 2{\cdot}1\rangle a\langle 2{\cdot}1{\cdot}1{\cdot}1, 2{\cdot}1{\cdot}1\rangle c\langle 2{\cdot}1{\cdot}2, 2{\cdot}2\rangle\};\ \varnothing;\ \mathit{Id}\\
\Longrightarrow^{\substack{\mathsf{Spl\text{-}H}\\ \mathsf{Clr\text{-}S}}}\ & \{\tilde{y}_1\colon \tilde{U}(a) \triangleq a;\ \tilde{Y}_1\colon \circ \triangleq \circ;\ a\langle 1{\cdot}1, 1\rangle, \tilde{z}_1\colon f(\tilde{U}(g(a,b,b),c),b,b) \triangleq f(g(a,d),c,d);\\
& \tilde{Z}_1\colon \circ \triangleq \circ;\ f\langle 1, 1\rangle g\langle 1{\cdot}1{\cdot}1, 1{\cdot}1\rangle a\langle 1{\cdot}1{\cdot}1{\cdot}1, 1{\cdot}1{\cdot}1\rangle c\langle 1{\cdot}1{\cdot}2, 1{\cdot}2\rangle\};\\
& \varnothing;\ \{\tilde{x} \mapsto (\tilde{Y}_1(\tilde{y}_1), \tilde{Z}_1(\tilde{z}_1)), \tilde{X} \mapsto \circ\}\\
\Longrightarrow_{\mathsf{Abs\text{-}L}}\ & \{\tilde{y}_1\colon a \triangleq a;\ \tilde{Y}_1\colon \tilde{U}(\circ) \triangleq \circ;\ a\langle 1, 1\rangle;\ \tilde{z}_1\colon f(\tilde{U}(g(a,b,b),c),b,b) \triangleq f(g(a,d),c,d);\\
& \tilde{Z}_1\colon \circ \triangleq \circ;\ f\langle 1, 1\rangle g\langle 1{\cdot}1{\cdot}1, 1{\cdot}1\rangle a\langle 1{\cdot}1{\cdot}1{\cdot}1, 1{\cdot}1{\cdot}1\rangle c\langle 1{\cdot}1{\cdot}2, 1{\cdot}2\rangle\};\\
& \varnothing;\ \{\tilde{x} \mapsto (\tilde{Y}_1(\tilde{y}_1), \tilde{Z}_1(\tilde{z}_1)), \tilde{X} \mapsto \circ\}\\
\Longrightarrow^{\substack{\mathsf{App\text{-}A}\\ \mathsf{Sol\text{-}H}}}\ & \{\tilde{z}_1\colon f(\tilde{U}(g(a,b,b),c),b,b) \triangleq f(g(a,d),c,d);\\
& \tilde{Z}_1\colon \circ \triangleq \circ;\ f\langle 1, 1\rangle g\langle 1{\cdot}1{\cdot}1, 1{\cdot}1\rangle a\langle 1{\cdot}1{\cdot}1{\cdot}1, 1{\cdot}1{\cdot}1\rangle c\langle 1{\cdot}1{\cdot}2, 1{\cdot}2\rangle\};\\
& \{\tilde{y}_1\colon \epsilon \triangleq \epsilon;\ \tilde{Y}_1\colon \tilde{U}(\circ) \triangleq \circ\};\ \{\tilde{x} \mapsto (\tilde{Y}_1(a), \tilde{Z}_1(\tilde{z}_1)), \tilde{X} \mapsto \circ\}\\
\Longrightarrow^{\substack{\mathsf{App\text{-}A}\\ \mathsf{Clr\text{-}S}}}\ & \{\tilde{z}_2\colon (\tilde{U}(g(a,b,b),c),b,b) \triangleq (g(a,d),c,d);\\
& \tilde{Z}_2\colon \circ \triangleq \circ;\ g\langle 1{\cdot}1, 1\rangle a\langle 1{\cdot}1{\cdot}1, 1{\cdot}1\rangle c\langle 1{\cdot}2, 2\rangle\};\\
& \{\tilde{y}_1\colon \epsilon \triangleq \epsilon;\ \tilde{Y}_1\colon \tilde{U}(\circ) \triangleq \circ\};\ \{\tilde{x} \mapsto (\tilde{Y}_1(a), f(\tilde{Z}_2(\tilde{z}_2))), \tilde{X} \mapsto \circ\}\\
\Longrightarrow_{\mathsf{Abs\text{-}L}}\ & \{\tilde{z}_2\colon (g(a,b,b),c) \triangleq (g(a,d),c,d);\ \tilde{Z}_2\colon (\tilde{U}(\circ),b,b) \triangleq \circ;\ g\langle 1, 1\rangle a\langle 1{\cdot}1, 1{\cdot}1\rangle c\langle 2, 2\rangle\};\\
& \{\tilde{y}_1\colon \epsilon \triangleq \epsilon;\ \tilde{Y}_1\colon \tilde{U}(\circ) \triangleq \circ\};\ \{\tilde{x} \mapsto (\tilde{Y}_1(a), f(\tilde{Z}_2(\tilde{z}_2))), \tilde{X} \mapsto \circ\}\\
\Longrightarrow^{\substack{\mathsf{Spl\text{-}H}\\ \mathsf{Clr\text{-}S}}}\ & \{\tilde{z}_3\colon g(a,b,b) \triangleq g(a,d);\ \tilde{Z}_3\colon \circ \triangleq \circ;\ g\langle 1, 1\rangle a\langle 1{\cdot}1, 1{\cdot}1\rangle, \tilde{z}_4\colon c \triangleq c;\ \tilde{Z}_4\colon \circ \triangleq \circ;\ c\langle 1, 1\rangle\};\\
& \{\tilde{y}_1\colon \epsilon \triangleq \epsilon;\ \tilde{Y}_1\colon \tilde{U}(\circ) \triangleq \circ,\ \tilde{z}_2\colon \epsilon \triangleq \epsilon;\ \tilde{Z}_2\colon (\tilde{U}(\circ),b,b) \triangleq (\circ,d)\};\\
& \{\tilde{x} \mapsto (\tilde{Y}_1(a), f(\tilde{Z}_2(\tilde{Z}_3(\tilde{z}_3), \tilde{Z}_4(\tilde{z}_4)))), \tilde{X} \mapsto \circ\}\\
\Longrightarrow^{\substack{\mathsf{App\text{-}A}\\ \mathsf{Clr\text{-}S}}}\ & \{\tilde{z}_5\colon (a,b,b) \triangleq (a,d);\ \tilde{Z}_5\colon \circ \triangleq \circ;\ a\langle 1, 1\rangle, \tilde{z}_4\colon c \triangleq c;\ \tilde{Z}_4\colon \circ \triangleq \circ;\ c\langle 1, 1\rangle\};\\
& \{\tilde{y}_1\colon \epsilon \triangleq \epsilon;\ \tilde{Y}_1\colon \tilde{U}(\circ) \triangleq \circ,\ \tilde{z}_2\colon \epsilon \triangleq \epsilon;\ \tilde{Z}_2\colon (\tilde{U}(\circ),b,b) \triangleq (\circ,d)\};\\
& \{\tilde{x} \mapsto (\tilde{Y}_1(a), f(\tilde{Z}_2(g(\tilde{Z}_5(\tilde{z}_5)), \tilde{Z}_4(\tilde{z}_4)))), \tilde{X} \mapsto \circ\}\\
\Longrightarrow^{\substack{\mathsf{Res\text{-}C}\\ \mathsf{Clr\text{-}S}}}\ & \{\tilde{z}_5\colon (a,b,b) \triangleq (a,d);\ \tilde{Z}_5\colon \circ \triangleq \circ;\ a\langle 1, 1\rangle, \tilde{z}_4\colon c \triangleq c;\ \tilde{Z}_4\colon \circ \triangleq \circ;\ c\langle 1, 1\rangle\};\\
& \{\tilde{y}_1\colon \epsilon \triangleq \epsilon;\ \tilde{Y}_1\colon \tilde{U}(\circ) \triangleq \circ,\ \tilde{z}_2\colon \epsilon \triangleq \epsilon;\ \tilde{Z}_2\colon \tilde{U}(\circ) \triangleq \circ,\ \tilde{y}_2\colon (b,b) \triangleq d;\ \tilde{Y}_2\colon \circ \triangleq \circ\};\\
& \{\tilde{x} \mapsto (\tilde{Y}_1(a), f(\tilde{Z}_2(g(\tilde{Z}_5(\tilde{z}_5)), \tilde{Z}_4(\tilde{z}_4)), \tilde{y}_2)), \tilde{X} \mapsto \circ\}\\
\Longrightarrow_{\mathsf{Mer\text{-}S}}\ & \{\tilde{z}_5\colon (a,b,b) \triangleq (a,d);\ \tilde{Z}_5\colon \circ \triangleq \circ;\ a\langle 1, 1\rangle, \tilde{z}_4\colon c \triangleq c;\ \tilde{Z}_4\colon \circ \triangleq \circ;\ c\langle 1, 1\rangle\};\\
& \{\tilde{y}_1\colon \epsilon \triangleq \epsilon;\ \tilde{Y}_1\colon \tilde{U}(\circ) \triangleq \circ,\ \tilde{y}_2\colon (b,b) \triangleq d;\ \tilde{Y}_2\colon \circ \triangleq \circ\};\\
& \{\tilde{x} \mapsto (\tilde{Y}_1(a), f(\tilde{Y}_1(g(\tilde{Z}_5(\tilde{z}_5)), \tilde{Z}_4(\tilde{z}_4)), \tilde{y}_2)), \tilde{X} \mapsto \circ\}\\
\Longrightarrow^{\substack{\mathsf{App\text{-}A}\\ \mathsf{Sol\text{-}H}}}\ & \{\tilde{z}_5\colon (a,b,b) \triangleq (a,d);\ \tilde{Z}_5\colon \circ \triangleq \circ;\ a\langle 1, 1\rangle\};\ \{\tilde{y}_1\colon \epsilon \triangleq \epsilon;\ \tilde{Y}_1\colon \tilde{U}(\circ) \triangleq \circ,\\
& \tilde{y}_2\colon (b,b) \triangleq d;\ \tilde{Y}_2\colon \circ \triangleq \circ\};\ \{\tilde{x} \mapsto (\tilde{Y}_1(a), f(\tilde{Y}_1(g(\tilde{Z}_5(\tilde{z}_5)), c), \tilde{y}_2)), \tilde{X} \mapsto \circ\}\\
\Longrightarrow^{\substack{\mathsf{App\text{-}A}\\ \mathsf{Sol\text{-}H}}}\ & \varnothing;\ \{\tilde{y}_1\colon \epsilon \triangleq \epsilon;\ \tilde{Y}_1\colon \tilde{U}(\circ) \triangleq \circ,\ \tilde{y}_2\colon (b,b) \triangleq d;\ \tilde{Y}_2\colon \circ \triangleq \circ,
\end{aligned}
$$

$$
\begin{aligned}
& \tilde{z}_5 \colon \epsilon \triangleq \epsilon;\ \tilde{Z}_5 \colon (\circ, b, b) \triangleq (\circ, d)\};\ \{\tilde{x} \mapsto (\tilde{Y}_1(a), f(\tilde{Y}_1(g(\tilde{Z}_5(a)), c), \tilde{y}_2)), \tilde{X} \mapsto \circ\} \\
\Longrightarrow^{\text{Res-C}}_{\text{Clr-S}}\ & \varnothing;\ \{\tilde{y}_1 \colon \epsilon \triangleq \epsilon;\ \tilde{Y}_1 \colon \tilde{U}(\circ) \triangleq \circ,\ \tilde{y}_2 \colon (b, b) \triangleq d;\ \tilde{Y}_2 \colon \circ \triangleq \circ,\ \tilde{y}_3 \colon (b, b) \triangleq d;\ \tilde{Y}_3 \colon \circ \triangleq \circ\}; \\
& \{\tilde{x} \mapsto (\tilde{Y}_1(a), f(\tilde{Y}_1(g(a, \tilde{y}_3), c), \tilde{y}_2)), \tilde{X} \mapsto \circ\} \\
\Longrightarrow_{\text{Mer-S}}\ & \varnothing;\ \{\tilde{y}_1 \colon \epsilon \triangleq \epsilon;\ \tilde{Y}_1 \colon \tilde{U}(\circ) \triangleq \circ,\ \tilde{y}_2 \colon (b, b) \triangleq d;\ \tilde{Y}_2 \colon \circ \triangleq \circ\}; \\
& \{\tilde{x} \mapsto (\tilde{Y}_1(a), f(\tilde{Y}_1(g(a, \tilde{y}_2), c), \tilde{y}_2)), \tilde{X} \mapsto \circ\}
\end{aligned}
$$

$\tilde{X}(\tilde{x})\sigma = (\tilde{Y}_1(a), f(\tilde{Y}_1(g(a, \tilde{y}_2), c), \tilde{y}_2))$ *is a rigid lgg of the two input hedges with respect to the given alignment and the store* S *contains all the information about the differences of the input hedges so that* $\tilde{X}(\tilde{x})\sigma\sigma_{_L}(S) = (\tilde{Y}_1(a), f(\tilde{Y}_1(g(a, \tilde{y}_2), c), \tilde{y}_2))\{\tilde{Y}_1 \mapsto \tilde{U}(\circ), \tilde{y}_2 \mapsto (b, b)\} = \tilde{s}$ *and* $\tilde{X}(\tilde{x})\sigma\sigma_{_R}(S) = (\tilde{Y}_1(a), f(\tilde{Y}_1(g(a, \tilde{y}_2), c), \tilde{y}_2))\{\tilde{Y}_1 \mapsto \circ, \tilde{y}_2 \mapsto d\} = \tilde{q}$.

Example 2.31. *Again we discuss the algorithm* $\mathfrak{G}_{\mathfrak{a}}^{2\mathcal{V}}$ *on the example composed from the taxonomy of editing scenarios for different clone types from [75]. In Figure 2.17 we recall the original code and its representation as an unranked term. It is used to illustrate the application of* $\mathfrak{G}_{\mathfrak{a}}^{2\mathcal{V}}$ *in order to detect its software clones.*

```
1  void sumProd(int n) {
2    float sum = 0.0;
3    float prod = 1.0;
4    for(int i=1; i<=n; i++) {
5      sum = sum + i;
6      prod = prod * i;
7      foo(sum,prod); }}
```

$$
\begin{aligned}
& sumProd(input(type(int), n), \\
& \quad returnType(void), \\
& =(type(float), sum, 0.0), \\
& =(type(float), prod, 1.0), \\
& for(=(type(int), i, 1), \leqslant(i, n), {+}{+}(i), \\
& \quad =(sum, +(sum, i)), \\
& \quad =(prod, *(prod, i)), \\
& \quad foo(sum, prod)))
\end{aligned}
$$

Figure 2.17: The original program used to illustrate clone detection by anti-unification.

Figure 2.18 shows the two clones we already discussed in subsection 2.1.3. The left one is of Type-3 and the right one of Type-4. First-order anti-unification was not able to detect the similarities that appear at different levels of the abstract syntax trees.

```
1  void sumProd(int n) {
2    float sum = 0.0;
3    float prod = 1.0;
4    for(int i=1; i<=n; i++) {
5      sum = sum + i;
6      prod = prod * i;
7      if (n % 2)==0 {
8        foo(sum, prod); }}}
```

```
void sumProd(int n) {
  float sum = 0.0;
  float prod = 1.0;
  int i=1; while(i<=n) {
    sum = sum + i;
    prod = prod * i;
    foo(sum,prod); i++; }}
```

Figure 2.18: Clones of Type-3 and Type-4.

In the first clone the application of foo has been nested into an if-statement and in the second one the for-loop has been replaced by a while-loop. $\mathfrak{G}_{\mathcal{R}}$ was not able to detect those similarities but by using $\mathfrak{G}_{\mathfrak{a}}^{2\mathcal{V}}$ we can reveal them.

$$
\begin{array}{l}
sumProd(input(type(int), n), \\
\quad returnType(void), \\
\quad =(type(float), sum, 0.0), \\
\quad =(type(float), prod, 1.0), \\
\quad for(=(type(int), i, 1), \leqslant(i, n), ++(i), \\
\qquad =(sum, +(sum, i)), \\
\qquad =(prod, *(prod, i)), \\
\qquad if(==(\%(n, 2), 0), \\
\qquad\quad foo(sum, prod))))
\end{array}
\qquad
\begin{array}{l}
sumProd(input(type(int), n), \\
\quad returnType(void), \\
\quad =(type(float), sum, 0.0), \\
\quad =(type(float), prod, 1.0), \\
\quad =(type(int), i, 1), while(\leqslant(i, n), \\
\qquad =(sum, +(sum, i)), \\
\qquad =(prod, *(prod, i)), \\
\qquad foo(sum, prod), ++(i)))
\end{array}
$$

Figure 2.19: The clones from Figure 2.18 as unranked terms.

Figure 2.19 shows the term encodings of the abstract syntax trees for two clones. For generalizing the (term representation of the) original code and one of the clones, we compute the admissible alignment of longest length (laa). We will discuss the computation of alignments in subsection 2.2.8. In both examples, the laa is unique. (In general this is not the case.) The length of the laa for the original code and the first clone is 41 which is the number of all symbols in the term representation of the original program. For the original code and the second clone it contains 38 symbols. Figure 2.20 shows the results after applying $\mathfrak{G}_{\mathfrak{a}}^{2\mathcal{V}}$ to the term representation of the original code and either of the clones with respect to the corresponding laa.

$$
\begin{array}{l}
sumProd(input(type(int), n), \\
\quad returnType(void), \\
\quad =(type(float), sum, 0.0), \\
\quad =(type(float), prod, 1.0), \\
\quad for(=(type(int), i, 1), \leqslant(i, n), ++(i), \\
\qquad =(sum, +(sum, i)), \\
\qquad =(prod, *(prod, i)), \\
\qquad \tilde{\mathbf{X}}(foo(sum, prod)))),
\end{array}
\qquad
\begin{array}{l}
sumProd(input(type(int), n), \\
\quad returnType(void), \\
\quad =(type(float), sum, 0.0), \\
\quad =(type(float), prod, 1.0), \\
\quad \tilde{\mathbf{X}}(=(type(int), i, 1), \tilde{\mathbf{Y}}(\leqslant(i, n), \tilde{\mathbf{x}}, \\
\qquad =(sum, +(sum, i)), \\
\qquad =(prod, *(prod, i)), \\
\qquad foo(sum, prod)))
\end{array}
$$

Figure 2.20: Result of running $\mathfrak{G}_{\mathfrak{a}}^{2\mathcal{V}}$ to detect the clones from Figure 2.19.

The first result is pretty clear. The input terms can be obtained from the generalization by instantiating the variable $\tilde{X}$ either with $if(==(\%(n, 2), 0), \circ)$ or the hole. From the second generalization, the term representation of the original code can be obtained by applying the substitution $\{\tilde{X} \mapsto for(\circ), \tilde{Y} \mapsto \circ, \tilde{x} \mapsto ++(i)\}$, and the term representation of the second clone can be obtained by applying the substitution $\{\tilde{X} \mapsto \circ,$

$\tilde{Y} \mapsto while(\circ, ++(i)), \tilde{x} \mapsto \epsilon\}$. The information about the differences is available from the store, as usual.

We consider two more examples: the left clone from Figure 2.5 which is of Type-2 and another cone of Type-3. The clones are illustrated in Figure 2.21.

```
1  void sumProd(int n) {
2    float sum = 0.0;
3    float prod = 1.0;
4    for(int i=1; i<=n; i++) {
5      sum = sum + (i*i);
6      prod = prod * (i*i);
7      foo(sum, prod); }}
```

```
void sumProd(int n) {
  float sum = 0.0;
  float prod = 1.0;
  for(int i=1; i<=n; i++) {
    sum = sum + i;
    prod = prod * i;
    bar(sum,prod); }}
```

Figure 2.21: Two clones of the program from Figure 2.19.

The translation into an unranked term of the left clone can be found at Figure 2.6 and in the right clone the function application $foo(sum, prod)$ from the original code has to be replaced by $bar(sum, prod)$. Running the algorithm $\mathfrak{G}_{\mathfrak{a}}^{2\mathcal{V}}$ with the laas of the original program and one of the clones gives in both cases a unique result modulo $\simeq$. The results are presented in Figure 2.22.

$$
\begin{array}{l}
sumProd(input(type(int), n), \\
\quad returnType(void), \\
\quad =(type(float), sum, 0.0), \\
\quad =(type(float), prod, 1.0), \\
\quad for(=(type(int), i, 1), \leqslant(i, n), ++(i), \\
\qquad =(sum, +(sum, \tilde{\mathbf{X}}(i))), \\
\qquad =(prod, *(prod, \tilde{\mathbf{X}}(i))), \\
\qquad foo(sum, prod)))
\end{array}
\qquad
\begin{array}{l}
sumProd(input(type(int), n), \\
\quad returnType(void), \\
\quad =(type(float), sum, 0.0), \\
\quad =(type(float), prod, 1.0), \\
\quad for(=(type(int), i, 1), \leqslant(i, n), ++(i), \\
\qquad =(sum, +(sum, i)), \\
\qquad =(prod, *(prod, i)), \\
\qquad \tilde{\mathbf{X}}(sum, prod)))
\end{array}
$$

Figure 2.22: Result of running $\mathfrak{G}_{\mathfrak{a}}^{2\mathcal{V}}$ to detect the clones from Figure 2.21.

In the first case there are again laas of length 41, they contain all symbols from the term representation of the original program. Notice that the laa is not unique, there are 4 laas, but all of them lead to the same result. The generalization $\mathfrak{G}_{\mathfrak{a}}^{2\mathcal{V}}$ computes in this case is strictly less general than any generalization computed by $\mathfrak{G}_{\mathcal{R}}$ (see Figure 2.7) for the same input. In the second clone, the application of foo has been changed into bar, hence the similarities appear under different heads. This cannot be detected by first-order generalization. However, the context variable $\tilde{X}$ does not tell us that the similarities appear at the same level of the input syntax trees. In section 2.3 we introduce function variables along with term variables to preserve such information.

2.2.6 Properties of the Algorithm $\mathfrak{G}_{\mathfrak{a}}^{2\mathcal{V}}$

Here we show that $\mathfrak{G}_{\mathfrak{a}}^{2\mathcal{V}}$ terminates and indeed computes a rigid lgg that is unique modulo $\simeq$ for two hedges and their admissible alignment. We also show that $\mathfrak{G}_{\mathfrak{a}}^{2\mathcal{V}}$ is coherent and its computational complexity is quadratic in time and linear in space on the size of the input. The algorithm $\mathfrak{G}_{\mathfrak{a}}^{2\mathcal{V}}$ maintains the following four invariants. We will use them to prove soundness of the algorithm.

Lemma 2.15 (Invariant 1)**.** *If* $\{\tilde{x}_0 \colon \tilde{s}_0 \triangleq \tilde{q}_0;\ \tilde{X}_0 \colon \circ \triangleq \circ;\ \mathfrak{a}_0\};\ S_0;\ \sigma_0 \Longrightarrow^* P_n;\ S_n;\ \sigma_n$ *is a derivation in* $\mathfrak{G}_{\mathfrak{a}}^{2\mathcal{V}}$*, then for all* $\tilde{x}_n \colon \tilde{s}_n \triangleq \tilde{q}_n;\ \tilde{X}_n \colon \tilde{c}_n \triangleq \tilde{d}_n;\ \mathfrak{a}_n \in P_n$ *either* $\mathfrak{a}_n \neq \mathfrak{e}$ *or* $\tilde{c}_n = \tilde{d}_n = \circ$*.*

Proof. The only rules that reduce the length of an alignment are Spl-H and App-A. The rule Spl-H splits an AUE such that the alignments of the two new AUEs are not empty. Therefore App-A is the only rule which transforms an AUE with a nonempty alignment into one with an empty alignment. Every AUE derived by App-A has the form $\tilde{x}_n \colon \tilde{s}_n \triangleq \tilde{q}_n;\ \tilde{X}_n \colon \circ \triangleq \circ;\ \mathfrak{a}_n$. □

Lemma 2.16 (Invariant 2)**.** *Let* $P_0; S_0; \sigma_0 \Longrightarrow^* P_n; S_n; \sigma_n$ *be a derivation in* $\mathfrak{G}_{\mathfrak{a}}^{2\mathcal{V}}$*. If for all* $\{\tilde{x}_1 \colon \tilde{s}_1 \triangleq \tilde{q}_1;\ \tilde{X}_1 \colon \tilde{c}_1 \triangleq \tilde{d}_1;\ \mathfrak{a}_1,\ \tilde{x}_2 \colon \tilde{s}_2 \triangleq \tilde{q}_2;\ \tilde{X}_2 \colon \tilde{c}_2 \triangleq \tilde{d}_2;\ \mathfrak{a}_2\} \subseteq P_0 \cup S_0$ *holds* $\tilde{x}_1 \neq \tilde{x}_2$ *and* $\tilde{X}_1 \neq \tilde{X}_2$*, then this implies* $\tilde{x}_3 \neq \tilde{x}_4$ *and* $\tilde{X}_3 \neq \tilde{X}_4$ *for all* $\{\tilde{x}_3 \colon \tilde{s}_3 \triangleq \tilde{q}_3;\ \tilde{X}_3 \colon \tilde{c}_3 \triangleq \tilde{d}_3;\ \mathfrak{a}_3,\ \tilde{x}_4 \colon \tilde{s}_4 \triangleq \tilde{q}_4;\ \tilde{X}_4 \colon \tilde{c}_4 \triangleq \tilde{d}_4;\ \mathfrak{a}_4\} \subseteq P_n \cup S_n$*.*

Proof. Looking at the rules, it is easy to see that no rule application duplicates a variable of an AUE, and every fresh variable is only used in one AUE. □

Lemma 2.17 (Invariant 3)**.** *Let* $P_0; S_0; \sigma_0 \Longrightarrow^* P_n; S_n; \sigma_n$ *be a derivation in* $\mathfrak{G}_{\mathfrak{a}}^{2\mathcal{V}}$*. If for all* $\tilde{x}_0 \colon \tilde{s}_0 \triangleq \tilde{q}_0;\ \tilde{X}_0 \colon \tilde{c}_0 \triangleq \tilde{d}_0;\ \mathfrak{a}_0 \in P_0$ *the variables* $\tilde{x}_0, \tilde{X}_0$ *only appear together as term* $\tilde{X}_0(\tilde{x}_0)$ *in* σ_0 *then this implies that for all* $\tilde{x}_n \colon \tilde{s}_n \triangleq \tilde{q}_n;\ \tilde{X}_n \colon \tilde{c}_n \triangleq \tilde{d}_n;\ \mathfrak{a}_n \in P_n$ *the variables* $\tilde{x}_n, \tilde{X}_n$ *only appear together as term* $\tilde{X}_n(\tilde{x}_n)$ *in* σ_n*. (This implies that they do not appear in* $Dom(\sigma_n)$*.)*

Proof. The rules Abs-L/Abs-R are trivial. From Lemma 2.16 we know that the variables of all the AUEs in $P_i \cup S_i$, $0 \leqslant i \leqslant n$ are pairwise disjoint. Furthermore, once a generalization variable appears in S_i, it will never appear in P_j again, for all $i \leqslant j \leqslant n$, because there is no rule which moves an AUE from the store S_j back to the problem set P_j. Therefore, for any generalization variable occurring in P_j, the rules Res-C, Mer-S, Clr-S (which only operate on generalization variables within S_i) have no effect on their appearance in σ_j. The rule Sol-H trivially maintains this property, as it moves an AUE to the store and therefore the condition is lifted for the corresponding variables. The rule App-A moves the selected AUE to the store such that the condition is lifted for those generalization variables, e.g. $\tilde{x}$ and $\tilde{X}$. Furthermore it introduces a new AUE with two fresh variables, say $\tilde{y}$ and $\tilde{Y}$, and it composes the mapping $\sigma_i\{\tilde{x} \mapsto a_1(\tilde{Y}(\tilde{y}))\}$, which again maintains the property that $\tilde{y}, \tilde{Y}$ only appear together as term $\tilde{Y}(\tilde{y})$ in σ_{i+1}. The reasoning for the rule Spl-H is very similar. It introduces two new AUEs and also maintains this property. □

Lemma 2.18 (Invariant 4)**.** *Let* $P_0;\ S_0;\ \sigma_0$ *such that for all* $\tilde{x}_0 \colon \tilde{s}_0 \triangleq \tilde{q}_0;\ \tilde{X}_0 \colon \tilde{c}_0 \triangleq \tilde{d}_0;\ \mathfrak{a}_0 \in P_0$ *the variables* $\tilde{x}_0, \tilde{X}_0$ *only appear together as term* $\tilde{X}_0(\tilde{x}_0)$ *in* σ_0*. If* $P_0;\ S_0;$

$\sigma_0 \Longrightarrow^* P_n; S_n; \sigma_n$ is a derivation in $\mathfrak{G}_{\mathfrak{a}}^{2\mathcal{V}}$ then for all $\tilde{x}_0 \colon \tilde{s}_0 \triangleq \tilde{q}_0; \tilde{X}_0 \colon \tilde{c}_0 \triangleq \tilde{d}_0$; $\mathfrak{a}_0 \in P_0 \cup S_0$ holds

- $\tilde{X}_0(\tilde{x}_0)\sigma_0\sigma_{_L}(P_0 \cup S_0) = \tilde{X}_0(\tilde{x}_0)\sigma_n\sigma_{_L}(P_n \cup S_n)$,
- $\tilde{X}_0(\tilde{x}_0)\sigma_0\sigma_{_R}(P_0 \cup S_0) = \tilde{X}_0(\tilde{x}_0)\sigma_n\sigma_{_R}(P_n \cup S_n)$.

Proof. By induction on the length of derivations. The trivial base case is the derivation of length zero. Let $P_0; S_0; \sigma_0$ such that for all $\tilde{x}_0 \colon \tilde{s}_0 \triangleq \tilde{q}_0; \tilde{X}_0 \colon \tilde{c}_0 \triangleq \tilde{d}_0$; $\mathfrak{a}_0 \in P_0$ the variables $\tilde{x}_0, \tilde{X}_0$ only appear together as term $\tilde{X}_0(\tilde{x}_0)$ in σ_0. Note that by Lemma 2.17 this property is an invariant of $\mathfrak{G}_{\mathfrak{a}}^{2\mathcal{V}}$. Let $P_0; S_0; \sigma_0 \Longrightarrow^* P_{n-1}; S_{n-1}; \sigma_{n-1} \Longrightarrow P_n; S_n; \sigma_n$ be a derivation in $\mathfrak{G}_{\mathfrak{a}}^{2\mathcal{V}}$. As induction hypothesis (IH) we assume that for all $\tilde{x}_0 \colon \tilde{s}_0 \triangleq \tilde{q}_0; \tilde{X}_0 \colon \tilde{c}_0 \triangleq \tilde{d}_0$; $\mathfrak{a}_0 \in P_0 \cup S_0$ holds

- $\tilde{X}_0(\tilde{x}_0)\sigma_0\sigma_{\mathrm{L}}(P_0 \cup S_0) = \tilde{X}_0(\tilde{x}_0)\sigma_{n-1}\sigma_{\mathrm{L}}(P_{n-1} \cup S_{n-1})$,
- $\tilde{X}_0(\tilde{x}_0)\sigma_0\sigma_{\mathrm{R}}(P_0 \cup S_0) = \tilde{X}_0(\tilde{x}_0)\sigma_{n-1}\sigma_{\mathrm{R}}(P_{n-1} \cup S_{n-1})$.

By case analysis on the applied rule, we will show that

- $\tilde{X}_0(\tilde{x}_0)\sigma_{n-1}\sigma_{\mathrm{L}}(P_{n-1} \cup S_{n-1}) = \tilde{X}_0(\tilde{x}_0)\sigma_n\sigma_{\mathrm{L}}(P_n \cup S_n)$,
- $\tilde{X}_0(\tilde{x}_0)\sigma_{n-1}\sigma_{\mathrm{R}}(P_{n-1} \cup S_{n-1}) = \tilde{X}_0(\tilde{x}_0)\sigma_n\sigma_{\mathrm{R}}(P_n \cup S_n)$.

We will only illustrate the proof for the left hand side $\tilde{X}_0(\tilde{x}_0)\sigma_{n-1}\sigma_{\mathrm{L}}(P_{n-1} \cup S_{n-1}) = \tilde{X}_0(\tilde{x}_0)\sigma_n\sigma_{\mathrm{L}}(P_n \cup S_n)$, serving as representative of both sides and skip the rule Abs-R which is a mirror image of Abs-L. For the sake of readability we will omit writing the alignments as we do not care about them in this proof.

Spl-H. $P_{n-1} = P \cup \{\tilde{x} \colon \tilde{s} \triangleq \tilde{q}; \tilde{X} \colon \tilde{c} \triangleq \tilde{d}\}$,
$P_n = P \cup \{\tilde{y} \colon \tilde{s}|_{i_1}^{i_k} \triangleq \tilde{q}|_{j_1}^{j_k}; \tilde{Y} \colon \circ \triangleq \circ\} \cup \{\tilde{z} \colon \tilde{s}|_{i_k^{++}}^{i_m} \triangleq \tilde{q}|_{j_k^{++}}^{j_m}; \tilde{Z} \colon \circ \triangleq \circ\}$,
$S_n = S_{n-1} \cup \{\tilde{x} \colon \epsilon \triangleq \epsilon; \tilde{X} \colon \tilde{c}[\tilde{s}|_1^{i_1^{--}}, \circ, \tilde{s}|_{i_m^{++}}^{|\tilde{s}|}] \triangleq \tilde{d}[\tilde{q}|_1^{j_1^{--}}, \circ, \tilde{q}|_{j_m^{++}}^{|\tilde{q}|}]\}$,
$\sigma_n = \sigma_{n-1}\{\tilde{x} \mapsto (\tilde{Y}(\tilde{y}), \tilde{Z}(\tilde{z}))\}$,
$\tilde{s} = (\tilde{s}|_1^{i_1^{--}}, \tilde{s}|_{i_1}^{i_k}, \tilde{s}|_{i_k^{++}}^{i_m}, \tilde{s}|_{i_m^{++}}^{|\tilde{s}|})$, $\tilde{q} = (\tilde{q}|_1^{j_1^{--}}, \tilde{q}|_{j_1}^{j_k}, \tilde{q}|_{j_k^{++}}^{j_m}, \tilde{q}|_{j_m^{++}}^{|\tilde{q}|})$.

Using Definition 2.32 we have the equality

$$\begin{aligned}
&\sigma_{n-1}\sigma_{\mathrm{L}}(P_{n-1} \cup S_{n-1}) \\
=\,&\sigma_{n-1}\sigma_{\mathrm{L}}(\{\tilde{x} \colon \tilde{s} \triangleq \tilde{q}; \tilde{X} \colon \tilde{c} \triangleq \tilde{d}\} \cup P \cup S_{n-1}) \\
=\,&\sigma_{n-1}\{\tilde{x} \mapsto \tilde{s}, \tilde{X} \mapsto \tilde{c}\}\sigma_{\mathrm{L}}(P \cup S_{n-1}) \\
=\,&\sigma_{n-1}\{\tilde{x} \mapsto (\tilde{s}|_1^{i_1^{--}}, \tilde{s}|_{i_1}^{i_k}, \tilde{s}|_{i_k^{++}}^{i_m}, \tilde{s}|_{i_m^{++}}^{|\tilde{s}|}), \tilde{X} \mapsto \tilde{c}\}\sigma_{\mathrm{L}}(P \cup S_{n-1}).
\end{aligned}$$

The variables $\tilde{y}, \tilde{Y}, \tilde{z}, \tilde{Z}$ are fresh and therefore it holds for all $\tilde{x}_0 \colon \tilde{s}_0 \triangleq \tilde{q}_0; \tilde{X}_0 \colon \tilde{c}_0 \triangleq \tilde{d}_0 \in P_0 \cup S_0$ that

$$\begin{aligned}
&\tilde{X}_0(\tilde{x}_0)\sigma_{n-1}\{\tilde{x} \mapsto (\tilde{s}|_1^{i_1^{--}}, \tilde{s}|_{i_1}^{i_k}, \tilde{s}|_{i_k^{++}}^{i_m}, \tilde{s}|_{i_m^{++}}^{|\tilde{s}|}), \tilde{X} \mapsto \tilde{c}\}\sigma_{\mathrm{L}}(P \cup S_{n-1}) \\
=\,&\tilde{X}_0(\tilde{x}_0)\sigma_{n-1}\{\tilde{x} \mapsto (\tilde{s}|_1^{i_1^{--}}, \tilde{Y}(\tilde{y}), \tilde{Z}(\tilde{z}), \tilde{s}|_{i_m^{++}}^{|\tilde{s}|}), \tilde{X} \mapsto \tilde{c}\} \\
&\{\tilde{y} \mapsto \tilde{s}|_{i_1}^{i_k}, \tilde{Y} \mapsto \circ\}\{\tilde{z} \mapsto \tilde{s}|_{i_k^{++}}^{i_m}, \tilde{Z} \mapsto \circ\}\sigma_{\mathrm{L}}(P \cup S_{n-1})
\end{aligned}$$

$$
\begin{aligned}
&= \tilde{X}_0(\tilde{x}_0)\sigma_{n-1}\{\tilde{x} \mapsto (\tilde{s}|_1^{i_1^{--}}, \tilde{Y}(\tilde{y}), \tilde{Z}(\tilde{z}), \tilde{s}|_{i_m^{++}}^{|\tilde{s}|}), \tilde{X} \mapsto \tilde{c}\} \\
&\quad \sigma_{\mathrm{L}}(\{\tilde{y}\colon \tilde{s}|_{i_1}^{i_k} \triangleq \tilde{q}|_{j_1}^{j_k};\ \tilde{Y}\colon \circ \triangleq \circ\} \cup \{\tilde{z}\colon \tilde{s}|_{i_k^{++}}^{i_m} \triangleq \tilde{q}|_{j_k^{++}}^{j_m};\ \tilde{Z}\colon \circ \triangleq \circ\} \cup P \cup S_{n-1}) \\
&= \tilde{X}_0(\tilde{x}_0)\sigma_{n-1}\{\tilde{x} \mapsto (\tilde{s}|_1^{i_1^{--}}, \tilde{Y}(\tilde{y}), \tilde{Z}(\tilde{z}), \tilde{s}|_{i_m^{++}}^{|\tilde{s}|}), \tilde{X} \mapsto \tilde{c}\}\sigma_{\mathrm{L}}(P_n \cup S_{n-1}).
\end{aligned}
$$

Further on, by Lemma 2.17 the variables $\tilde{x}, \tilde{X}$ only appear together as term $\tilde{X}(\tilde{x})$ in σ_{n-1} which leads to

$$
\begin{aligned}
&\tilde{X}_0(\tilde{x}_0)\sigma_{n-1}\{\tilde{x} \mapsto (\tilde{s}|_1^{i_1^{--}}, \tilde{Y}(\tilde{y}), \tilde{Z}(\tilde{z}), \tilde{s}|_{i_m^{++}}^{|\tilde{s}|}), \tilde{X} \mapsto \tilde{c}\}\sigma_{\mathrm{L}}(P_n \cup S_{n-1}) \\
&= \tilde{X}_0(\tilde{x}_0)\sigma_{n-1}\{\tilde{x} \mapsto (\tilde{Y}(\tilde{y}), \tilde{Z}(\tilde{z})), \tilde{X} \mapsto \tilde{c}[\tilde{s}|_1^{i_1^{--}}, \circ, \tilde{s}|_{i_m^{++}}^{|\tilde{s}|}]\}\sigma_{\mathrm{L}}(P_n \cup S_{n-1}) \\
&= \tilde{X}_0(\tilde{x}_0)\sigma_{n-1}\{\tilde{x} \mapsto (\tilde{Y}(\tilde{y}), \tilde{Z}(\tilde{z}))\}\{\tilde{x} \mapsto \epsilon, \tilde{X} \mapsto \tilde{c}[\tilde{s}|_1^{i_1^{--}}, \circ, \tilde{s}|_{i_m^{++}}^{|\tilde{s}|}]\}\sigma_{\mathrm{L}}(P_n \cup S_{n-1}) \\
&= \tilde{X}_0(\tilde{x}_0)\sigma_n\sigma_{\mathrm{L}}(\{\tilde{x}\colon \epsilon \triangleq \epsilon;\ \tilde{X}\colon \tilde{c}[\tilde{s}|_1^{i_1^{--}}, \circ, \tilde{s}|_{i_m^{++}}^{|\tilde{s}|}] \triangleq \tilde{d}[\tilde{q}|_1^{j_1^{--}}, \circ, \tilde{q}|_{j_m^{++}}^{|\tilde{q}|}]\} \cup P_n \cup S_{n-1}) \\
&= \tilde{X}_0(\tilde{x}_0)\sigma_n\sigma_{\mathrm{L}}(P_n \cup S_n).
\end{aligned}
$$

App-A. $P_{n-1} = \{\tilde{x}\colon (\tilde{s}_l, a_1(\tilde{s}), \tilde{s}_r) \triangleq (\tilde{q}_l, a_1(\tilde{q}), \tilde{q}_r);\ \tilde{X}\colon \tilde{c} \triangleq \tilde{d}\} \cup P$,
$P_n = \{\tilde{y}\colon \tilde{s} \triangleq \tilde{q};\ \tilde{Y}\colon \circ \triangleq \circ\} \cup P$,
$S_n = S_{n-1} \cup \{\tilde{x}\colon \epsilon \triangleq \epsilon;\ \tilde{X}\colon \tilde{c}[\tilde{s}_l, \circ, \tilde{s}_r] \triangleq \tilde{d}[\tilde{q}_l, \circ, \tilde{q}_r]\}$,
$\sigma_n = \sigma_{n-1}\{\tilde{x} \mapsto a_1(\tilde{Y}(\tilde{y}))\}$.

By Definition 2.32 we get

$$
\begin{aligned}
&\sigma_{n-1}\sigma_{\mathrm{L}}(P_{n-1} \cup S_{n-1}) \\
&= \sigma_{n-1}\sigma_{\mathrm{L}}(\{\tilde{x}\colon (\tilde{s}_l, a_1(\tilde{s}), \tilde{s}_r) \triangleq (\tilde{q}_l, a_1(\tilde{q}), \tilde{q}_r);\ \tilde{X}\colon \tilde{c} \triangleq \tilde{d}\} \cup P \cup S_{n-1}) \\
&= \sigma_{n-1}\{\tilde{x} \mapsto (\tilde{s}_l, a_1(\tilde{s}), \tilde{s}_r), \tilde{X} \mapsto \tilde{c}\}\sigma_{\mathrm{L}}(P \cup S_{n-1}).
\end{aligned}
$$

The variables $\tilde{y}, \tilde{Y}$ are fresh and therefore it holds for all $\tilde{x}_0\colon \tilde{s}_0 \triangleq \tilde{q}_0;\ \tilde{X}_0\colon \tilde{c}_0 \triangleq \tilde{d}_0 \in P_0 \cup S_0$ that

$$
\begin{aligned}
&\tilde{X}_0(\tilde{x}_0)\sigma_{n-1}\{\tilde{x} \mapsto (\tilde{s}_l, a_1(\tilde{s}), \tilde{s}_r), \tilde{X} \mapsto \tilde{c}\}\sigma_{\mathrm{L}}(P \cup S_{n-1}) \\
&= \tilde{X}_0(\tilde{x}_0)\sigma_{n-1}\{\tilde{x} \mapsto (\tilde{s}_l, a_1(\tilde{Y}(\tilde{y})), \tilde{s}_r), \tilde{X} \mapsto \tilde{c}\}\{\tilde{y} \mapsto \tilde{s}, \tilde{Y} \mapsto \circ\}\sigma_{\mathrm{L}}(P \cup S_{n-1}) \\
&= \tilde{X}_0(\tilde{x}_0)\sigma_{n-1}\{\tilde{x} \mapsto (\tilde{s}_l, a_1(\tilde{Y}(\tilde{y})), \tilde{s}_r), \tilde{X} \mapsto \tilde{c}\}\sigma_{\mathrm{L}}(\{\tilde{y}\colon \tilde{s} \triangleq \tilde{q};\ \tilde{Y}\colon \circ \triangleq \circ\} \cup P \cup S_{n-1}) \\
&= \tilde{X}_0(\tilde{x}_0)\sigma_{n-1}\{\tilde{x} \mapsto (\tilde{s}_l, a_1(\tilde{Y}(\tilde{y})), \tilde{s}_r), \tilde{X} \mapsto \tilde{c}\}\sigma_{\mathrm{L}}(P_n \cup S_{n-1}).
\end{aligned}
$$

Further on, by Lemma 2.17 the variables $\tilde{x}, \tilde{X}$ only appear together as term $\tilde{X}(\tilde{x})$ in σ_{n-1} which leads to

$$
\begin{aligned}
&\tilde{X}_0(\tilde{x}_0)\sigma_{n-1}\{\tilde{x} \mapsto (\tilde{s}_l, a_1(\tilde{Y}(\tilde{y})), \tilde{s}_r), \tilde{X} \mapsto \tilde{c}\}\sigma_{\mathrm{L}}(P_n \cup S_{n-1}) \\
&= \tilde{X}_0(\tilde{x}_0)\sigma_{n-1}\{\tilde{x} \mapsto a_1(\tilde{Y}(\tilde{y})), \tilde{X} \mapsto \tilde{c}[\tilde{s}_l, \circ, \tilde{s}_r]\}\sigma_{\mathrm{L}}(P_n \cup S_{n-1}) \\
&= \tilde{X}_0(\tilde{x}_0)\sigma_{n-1}\{\tilde{x} \mapsto a_1(\tilde{Y}(\tilde{y}))\}\{\tilde{x} \mapsto \epsilon, \tilde{X} \mapsto \tilde{c}[\tilde{s}_l, \circ, \tilde{s}_r]\}\sigma_{\mathrm{L}}(P_n \cup S_{n-1}) \\
&= \tilde{X}_0(\tilde{x}_0)\sigma_n\sigma_{\mathrm{L}}(\{\tilde{x}\colon \epsilon \triangleq \epsilon;\ \tilde{X}\colon \tilde{c}[\tilde{s}_l, \circ, \tilde{s}_r] \triangleq \tilde{d}[\tilde{q}_l, \circ, \tilde{q}_r]\} \cup P_n \cup S_{n-1}) \\
&= \tilde{X}_0(\tilde{x}_0)\sigma_n\sigma_{\mathrm{L}}(P_n \cup S_n).
\end{aligned}
$$

Abs-L. $P_{n-1} = \{\tilde{x}\colon (\tilde{s}_l, \hbar(\tilde{s}), \tilde{s}_r) \triangleq \tilde{q};\ \tilde{X}\colon \tilde{c} \triangleq \tilde{d}\} \cup P,$
$P_n = \{\tilde{x}\colon \tilde{s} \triangleq \tilde{q};\ \tilde{X}\colon \tilde{c}[\tilde{s}_l, \hbar(\circ), \tilde{s}_r] \triangleq \tilde{d}\} \cup P,$
$S_n = S_{n-1},\ \sigma_n = \sigma_{n-1}.$

Starting with Definition 2.32 we get

$$\begin{aligned}
&\sigma_{n-1}\sigma_{\mathrm{L}}(P_{n-1} \cup S_{n-1})\\
=\ &\sigma_n\sigma_{\mathrm{L}}(\{\tilde{x}\colon (\tilde{s}_l, \hbar(\tilde{s}), \tilde{s}_r) \triangleq \tilde{q};\ \tilde{X}\colon \tilde{c} \triangleq \tilde{d}\} \cup P \cup S_n)\\
=\ &\sigma_n\{\tilde{x} \mapsto (\tilde{s}_l, \hbar(\tilde{s}), \tilde{s}_r), \tilde{X} \mapsto \tilde{c}\}\sigma_{\mathrm{L}}(P \cup S_n).
\end{aligned}$$

By Lemma 2.17 the variables $\tilde{x}, \tilde{X}$ only appear together as term $\tilde{X}(\tilde{x})$ in $\sigma_{n-1} = \sigma_n$. Therefore it follows that for all $\tilde{x}_0\colon \tilde{s}_0 \triangleq \tilde{q}_0;\ \tilde{X}_0\colon \tilde{c}_0 \triangleq \tilde{d}_0 \in P_0 \cup S_0$ holds

$$\begin{aligned}
&\tilde{X}_0(\tilde{x}_0)\sigma_n\{\tilde{x} \mapsto (\tilde{s}_l, \hbar(\tilde{s}), \tilde{s}_r), \tilde{X} \mapsto \tilde{c}\}\sigma_{\mathrm{L}}(P \cup S_n)\\
=\ &\tilde{X}_0(\tilde{x}_0)\sigma_n\{\tilde{x} \mapsto \tilde{s}, \tilde{X} \mapsto \tilde{c}[\tilde{s}_l, \hbar(\circ), \tilde{s}_r]\}\sigma_{\mathrm{L}}(P \cup S_n)\\
=\ &\tilde{X}_0(\tilde{x}_0)\sigma_n\sigma_{\mathrm{L}}(\{\tilde{x}\colon \tilde{s} \triangleq \tilde{q};\ \tilde{X}\colon \tilde{c}[\tilde{s}_l, \hbar(\circ), \tilde{s}_r] \triangleq \tilde{d}\} \cup P \cup S_n)\\
=\ &\tilde{X}_0(\tilde{x}_0)\sigma_n\sigma_{\mathrm{L}}(P_n \cup S_n).
\end{aligned}$$

Sol-H. $P_n = P_{n-1}\backslash\{\tilde{x}\colon \tilde{s} \triangleq \tilde{q};\ \tilde{X}\colon \circ \triangleq \circ\},$
$S_n = S_{n-1} \cup \{\tilde{x}\colon \tilde{s} \triangleq \tilde{q};\ \tilde{X}\colon \circ \triangleq \circ\},$
$\sigma_n = \sigma_{n-1}\{\tilde{X} \mapsto \circ\}.$

Obviously the sets $P_{n-1} \cup S_{n-1}$ and $P_n \cup S_n$ are equal such that $\sigma_{\mathrm{L}}(P_{n-1} \cup S_{n-1}) = \sigma_{\mathrm{L}}(P_n \cup S_n)$ and furthermore $\{\tilde{x}\colon \tilde{s} \triangleq \tilde{q};\ \tilde{X}\colon \circ \triangleq \circ;\ \mathfrak{e}\} \in P_n \cup S_n$ leads to $\tilde{X}\sigma_{\mathrm{L}}(P_n \cup S_n) = \circ$, by Definition 2.32. Finally we get

$$\begin{aligned}
&\sigma_{n-1}\sigma_{\mathrm{L}}(P_n \cup S_n)\\
=\ &\sigma_{n-1}\{\tilde{X} \mapsto \circ\}\sigma_{\mathrm{L}}(P_n \cup S_n)\\
=\ &\sigma_n\sigma_{\mathrm{L}}(P_n \cup S_n).
\end{aligned}$$

Res-C. $P_n = P_{n-1},$
$S_n\backslash\{\tilde{x}\colon \epsilon \triangleq \epsilon;\ \tilde{X}\colon \dot{c} \triangleq \dot{d},\ \tilde{y}\colon \tilde{s}_l \triangleq \tilde{q}_l;\ \tilde{Y}\colon \circ \triangleq \circ,\ \tilde{z}\colon \tilde{s}_r \triangleq \tilde{q}_r;\ \tilde{Z}\colon \circ \triangleq \circ\}$
$= S_{n-1}\backslash\{\tilde{x}\colon \epsilon \triangleq \epsilon;\ \tilde{X}\colon (\tilde{s}_l, \dot{c}, \tilde{s}_r) \triangleq (\tilde{q}_l, \dot{d}, \tilde{q}_r)\},$
$\sigma_n = \sigma_{n-1}\{\tilde{X} \mapsto (\tilde{y}, \tilde{X}(\circ), \tilde{z})\}.$

Therefore by Definition 2.32 holds

$$\begin{aligned}
&\sigma_{n-1}\sigma_{\mathrm{L}}(P_{n-1} \cup S_{n-1})\\
=\ &\sigma_{n-1}\{\tilde{X} \mapsto (\tilde{s}_l, \dot{c}, \tilde{s}_r), \tilde{x} \mapsto \epsilon\}\sigma_{\mathrm{L}}(P_{n-1} \cup\\
&S_{n-1}\backslash\{\tilde{x}\colon \epsilon \triangleq \epsilon;\ \tilde{X}\colon (\tilde{s}_l, \dot{c}, \tilde{s}_r) \triangleq (\tilde{q}_l, \dot{d}, \tilde{q}_r)\})\\
=\ &\sigma_{n-1}\{\tilde{X} \mapsto (\tilde{s}_l, \dot{c}, \tilde{s}_r), \tilde{x} \mapsto \epsilon\}\sigma_{\mathrm{L}}(P_{n-1} \cup\\
&S_n\backslash\{\tilde{x}\colon \epsilon \triangleq \epsilon;\ \tilde{X}\colon \dot{c} \triangleq \dot{d},\ \tilde{y}\colon \tilde{s}_l \triangleq \tilde{q}_l;\ \tilde{Y}\colon \circ \triangleq \circ,\ \tilde{z}\colon \tilde{s}_r \triangleq \tilde{q}_r;\ \tilde{Z}\colon \circ \triangleq \circ\})\\
=\ &\sigma_{n-1}\{\tilde{X} \mapsto (\tilde{s}_l, \tilde{X}(\circ), \tilde{s}_r)\}\{\tilde{x} \mapsto \epsilon, \tilde{X} \mapsto \dot{c}\}\sigma_{\mathrm{L}}(P_{n-1} \cup\\
&S_n\backslash\{\tilde{x}\colon \epsilon \triangleq \epsilon;\ \tilde{X}\colon \dot{c} \triangleq \dot{d},\ \tilde{y}\colon \tilde{s}_l \triangleq \tilde{q}_l;\ \tilde{Y}\colon \circ \triangleq \circ,\ \tilde{z}\colon \tilde{s}_r \triangleq \tilde{q}_r;\ \tilde{Z}\colon \circ \triangleq \circ\})\\
=\ &\sigma_{n-1}\{\tilde{X} \mapsto (\tilde{s}_l, \tilde{X}(\circ), \tilde{s}_r)\}\sigma_{\mathrm{L}}(P_{n-1} \cup
\end{aligned}$$

$$S_n\backslash\{\tilde{y}\colon \tilde{s}_l \triangleq \tilde{q}_l;\ \tilde{Y}\colon \circ \triangleq \circ,\ \tilde{z}\colon \tilde{s}_r \triangleq \tilde{q}_r;\ \tilde{Z}\colon \circ \triangleq \circ\}).$$

The variables $\tilde{y}, \tilde{z}, \tilde{Y}, \tilde{Z}$ are fresh and therefore it holds for all $\tilde{x}_0\colon \tilde{s}_0 \triangleq \tilde{q}_0;\ \tilde{X}_0\colon \tilde{c}_0 \triangleq \tilde{d}_0 \in P_0 \cup S_0$ that

$$\begin{aligned}
&\tilde{X}_0(\tilde{x}_0)\sigma_{n-1}\{\tilde{X} \mapsto (\tilde{s}_l, \tilde{X}(\circ), \tilde{s}_r)\}\sigma_{\mathrm{L}}(P_{n-1}\cup\\
&S_n\backslash\{\tilde{y}\colon \tilde{s}_l \triangleq \tilde{q}_l;\ \tilde{Y}\colon \circ \triangleq \circ,\ \tilde{z}\colon \tilde{s}_r \triangleq \tilde{q}_r;\ \tilde{Z}\colon \circ \triangleq \circ\})\\
=\ &\tilde{X}_0(\tilde{x}_0)\sigma_{n-1}\{\tilde{X} \mapsto (\tilde{y}, \tilde{X}(\circ), \tilde{z})\}\{\tilde{y} \mapsto \tilde{s}_l, \tilde{z} \mapsto \tilde{s}_r, \tilde{Y} \mapsto \circ, \tilde{Z} \mapsto \circ\}\sigma_{\mathrm{L}}(P_{n-1}\cup\\
&S_n\backslash\{\tilde{y}\colon \tilde{s}_l \triangleq \tilde{q}_l;\ \tilde{Y}\colon \circ \triangleq \circ,\ \tilde{z}\colon \tilde{s}_r \triangleq \tilde{q}_r;\ \tilde{Z}\colon \circ \triangleq \circ\})\\
=\ &\tilde{X}_0(\tilde{x}_0)\sigma_{n-1}\{\tilde{X} \mapsto (\tilde{y}, \tilde{X}(\circ), \tilde{z})\}\sigma_{\mathrm{L}}(P_{n-1} \cup S_n)\\
=\ &\tilde{X}_0(\tilde{x}_0)\sigma_n\sigma_{\mathrm{L}}(P_{n-1} \cup S_n)\\
=\ &\tilde{X}_0(\tilde{x}_0)\sigma_n\sigma_{\mathrm{L}}(P_n \cup S_n).
\end{aligned}$$

Mer-S. $P_n = P_{n-1}$,
$S_n = S_{n-1}\backslash\{\tilde{x}_2\colon \tilde{s} \triangleq \tilde{q};\ \tilde{X}_2\colon \tilde{c} \triangleq \tilde{d}\}$,
$\tilde{x}_1\colon \tilde{s} \triangleq \tilde{q};\ \tilde{X}_1\colon \tilde{c} \triangleq \tilde{d} \in S_n$,
$\sigma_n = \sigma_{n-1}\{\tilde{x}_2 \mapsto \tilde{x}_1, \tilde{X}_2 \mapsto \tilde{X}_1(\circ)\}$.
Therefore by Definition 2.32 holds

$$\begin{aligned}
&\sigma_{\mathrm{L}}(P_{n-1} \cup S_{n-1})\\
=\ &\{\tilde{x}_2 \mapsto \tilde{s}, \tilde{X}_2 \mapsto \tilde{c}\}\sigma_{\mathrm{L}}(P_{n-1} \cup S_{n-1}\backslash\{\tilde{x}_2\colon \tilde{s} \triangleq \tilde{q};\ \tilde{X}_2\colon \tilde{c} \triangleq \tilde{d}\})\\
=\ &\{\tilde{x}_2 \mapsto \tilde{s}, \tilde{X}_2 \mapsto \tilde{c}\}\sigma_{\mathrm{L}}(P_{n-1} \cup S_n).
\end{aligned}$$

From the fact $\tilde{x}_1\colon \tilde{s} \triangleq \tilde{q};\ \tilde{X}_1\colon \tilde{c} \triangleq \tilde{d} \in S_n$ follows that $\tilde{x}_1\sigma_{\mathrm{L}}(P_{n-1} \cup S_n) = \tilde{s}$ and $\tilde{X}_1\sigma_{\mathrm{L}}(P_{n-1} \cup S_n) = \tilde{c}$, which finally leads to

$$\begin{aligned}
&\sigma_{n-1}\{\tilde{x}_2 \mapsto \tilde{s}, \tilde{X}_2 \mapsto \tilde{c}\}\sigma_{\mathrm{L}}(P_{n-1} \cup S_n)\\
=\ &\sigma_{n-1}\{\tilde{x}_2 \mapsto \tilde{x}_1, \tilde{X}_2 \mapsto \tilde{X}_1(\circ)\}\sigma_{\mathrm{L}}(P_{n-1} \cup S_n)\\
=\ &\sigma_n\sigma_{\mathrm{L}}(P_{n-1} \cup S_n)\\
=\ &\sigma_n\sigma_{\mathrm{L}}(P_n \cup S_n).
\end{aligned}$$

Clr-S. $P_n = P_{n-1}$,
$S_n = S_{n-1}\backslash\{\tilde{x}\colon \epsilon \triangleq \epsilon;\ \tilde{X}\colon \circ \triangleq \circ\}$,
$\sigma_n = \sigma_{n-1}\{\tilde{x} \mapsto \epsilon, \tilde{X} \mapsto \circ\}$.
By Definition 2.32 it holds that

$$\begin{aligned}
&\sigma_{n-1}\sigma_{\mathrm{L}}(P_{n-1} \cup S_{n-1})\\
=\ &\sigma_{n-1}\{\tilde{x} \mapsto \epsilon, \tilde{X} \mapsto \circ\}\sigma_{\mathrm{L}}(P_{n-1} \cup S_{n-1}\backslash\{\tilde{x}\colon \epsilon \triangleq \epsilon;\ \tilde{X}\colon \circ \triangleq \circ\})\\
=\ &\sigma_{n-1}\{\tilde{x} \mapsto \epsilon, \tilde{X} \mapsto \circ\}\sigma_{\mathrm{L}}(P_{n-1} \cup S_n)\\
=\ &\sigma_n\sigma_{\mathrm{L}}(P_{n-1} \cup S_n)\\
=\ &\sigma_n\sigma_{\mathrm{L}}(P_n \cup S_n).
\end{aligned}$$

□

This lemma has a corollary which states that for the invariant, the initial substitution is irrelevant:

Corollary 2.19. *If* P_0; S_0; $\vartheta_0 \Longrightarrow^* P_n$; S_n; $\vartheta_0\vartheta_1 \dots \vartheta_n$ *is a derivation in* $\mathfrak{G}_{\mathfrak{a}}^{2\mathcal{V}}$ *then for all* $\tilde{x}_0 \colon \tilde{s}_0 \triangleq \tilde{q}_0$; $\tilde{X}_0 \colon \tilde{c}_0 \triangleq \tilde{d}_0$; $\mathfrak{a}_0 \in P_0 \cup S_0$ *holds*

- $\tilde{X}_0(\tilde{x}_0)\sigma_{L}(P_0 \cup S_0) = \tilde{X}_0(\tilde{x}_0)\vartheta_1 \dots \vartheta_n\sigma_{L}(P_n \cup S_n)$,
- $\tilde{X}_0(\tilde{x}_0)\sigma_{R}(P_0 \cup S_0) = \tilde{X}_0(\tilde{x}_0)\vartheta_1 \dots \vartheta_n\sigma_{R}(P_n \cup S_n)$.

Using this results, we state now the main theorems which together give us correctness and uniqueness of the generalizations that are computed by $\mathfrak{G}_{\mathfrak{a}}^{2\mathcal{V}}$. From Theorem 2.21 (Soundness) follows coherence of $\mathfrak{G}_{\mathfrak{a}}^{2\mathcal{V}}$.

Theorem 2.20 (Termination)**.** *The algorithm* $\mathfrak{G}_{\mathfrak{a}}^{2\mathcal{V}}$ *terminates on any input.*

Proof. We define the complexity measure of the triple P; S; σ as a tuple of multisets (M_P, M_S), where

$$M_P = \{\|\tilde{s}\| + \|\tilde{q}\| \mid \tilde{x} \colon \tilde{s} \triangleq \tilde{q};\ \tilde{X} \colon \tilde{c} \triangleq \tilde{d};\ \mathfrak{a} \in P\},$$
$$M_S = \{\|\tilde{s}\| + \|\tilde{q}\| + \|\tilde{c}\| + \|\tilde{d}\| \mid \tilde{x} \colon \tilde{s} \triangleq \tilde{q};\ \tilde{X} \colon \tilde{c} \triangleq \tilde{d};\ \mathfrak{a} \in S\}.$$

The measures are compared by the well-founded lexicographic ordering. Each rule strictly reduces the complexity of the triple P; S; σ. □

The Soundness Theorem shows that $\mathfrak{G}_{\mathfrak{a}}^{2\mathcal{V}}$ indeed computes rigid generalizations. Besides, the store keeps the information which indicates how to obtain the initial hedges from the generalization:

Theorem 2.21 (Soundness)**.** *Let* P *be a set of AUEs of the form* $\{\tilde{x} \colon \tilde{s} \triangleq \tilde{q};\ \tilde{X} \colon \circ \triangleq \circ;\ \mathfrak{a}\}$. *Every exhaustive rule application in* $\mathfrak{G}_{\mathfrak{a}}^{2\mathcal{V}}$ *yields a derivation* P; $\varnothing$; $Id \Longrightarrow^+ \varnothing$; S; σ *where* $\tilde{g} = \tilde{X}(\tilde{x})\sigma$ *is a rigid generalization of* $\tilde{s}$ *and* $\tilde{q}$ *with respect to* $\mathfrak{a}$ *and the store* S *records all the differences such that* $\tilde{g}\sigma_{L}(S) = \tilde{s}$ *and* $\tilde{g}\sigma_{R}(S) = \tilde{q}$.

Proof. We will proceed in the following way:

1. For any arbitrary fixed AUE in P, there is a rule in $\mathfrak{G}_{\mathfrak{a}}^{2\mathcal{V}}$ which is applicable.
2. $\tilde{g}$ is a supporting generalization of $\tilde{s}$ and $\tilde{q}$ with respect to $\mathfrak{a}$.
   $\mathfrak{G}_{\mathfrak{a}}^{2\mathcal{V}}$ maintains the invariant that $P \cup S$ is a set of AUEs.
   S records all the differences such that $\tilde{g}\sigma_{\mathrm{L}}(S) = \tilde{s}$ and $\tilde{g}\sigma_{\mathrm{R}}(S) = \tilde{q}$.
3. $\tilde{g}$ is a rigid generalization of $\tilde{s}$ and $\tilde{q}$ with respect to $\mathfrak{a}$.

First we introduce some auxiliary notations used in the proof. The notation ϑ_i^k is used for a substitution composition $\vartheta_i\vartheta_{i+1} \dots \vartheta_k$. Given an AUE $\tilde{x} \colon \tilde{s} \triangleq \tilde{q}$; $\tilde{X} \colon \tilde{c} \triangleq \tilde{d}$; $\mathfrak{a} \in P$ we denote by $\mathring{\mathfrak{a}}$ the admissible alignment of $\tilde{c}[\tilde{s}]$ and $\tilde{d}[\tilde{q}]$. It is obtained by extending $\mathfrak{a} = a_1\langle i_1 \cdot I_1, j_1 \cdot J_1\rangle \dots a_m\langle i_m \cdot I_m, j_m \cdot J_m\rangle$ with the positions of the holes in $\tilde{c}$, $\tilde{d}$ donated $I_\circ \cdot i_\circ$, $J_\circ \cdot j_\circ$, respectively, in the way $\mathring{\mathfrak{a}} = a_1\langle I_\circ \cdot (i_\circ + i_1) \cdot I_1, J_\circ \cdot (j_\circ + j_1) \cdot J_1\rangle \dots$ $a_m\langle I_\circ \cdot (i_\circ + i_m) \cdot I_m, J_\circ \cdot (j_\circ + j_m) \cdot J_m\rangle$.

Ad 1. Show that for any $\tilde{x}\colon \tilde{s} \triangleq \tilde{q}$; $\tilde{X}\colon \tilde{c} \triangleq \tilde{d}$; $\mathfrak{a} \in P$ with $\mathfrak{a}$ being an admissible alignment of $\tilde{s}$ and $\tilde{q}$ there is a rule which can be applied. If $\mathfrak{a} = \mathfrak{e}$ then the rule Sol-H is applicable by Lemma 2.15. Let $\mathfrak{a} = a_1\langle i_1{\cdot}I_1, j_1{\cdot}J_1\rangle \dots a_m\langle i_m{\cdot}I_m, j_m{\cdot}J_m\rangle \neq \mathfrak{e}$. From the condition of Spl-H it follows that the rule Spl-H is applicable iff $i_1 \neq i_m$ and $j_1 \neq j_m$. Otherwise either $i_1 = \dots = i_m$ or $j_1 = \dots = j_m$. W.l.o.g. we assume $i_1 = \dots = i_m$. If $I_1 \neq \epsilon$ then Abs-L is applicable, therefore we furthermore assume $I_1 = \epsilon$, which gives us an alignment of the form $a_1\langle i, j_1{\cdot}J_1\rangle \dots a_m\langle i{\cdot}I_m, j_m{\cdot}J_m\rangle$. If we also have $j_1 = \dots = j_m$, then either we can apply Abs-R or App-A. (Note that, if $a_1\langle i, j\rangle a_2\langle I_2, J_2\rangle \dots a_m\langle I_m, J_m\rangle$ is an admissible alignment of $\tilde{s}$ and $\tilde{q}$, then a_1 is the symbol at position i in $\tilde{s}$ and at position j in $\tilde{q}$.) This leaves us with the case where $j_1 \neq j_m$, leading to $j_1{\cdot}J_1 \sqcupdot j_m{\cdot}J_m$ but $i \sqsubset i{\cdot}I_m$ which is a collision and cannot appear in an admissible alignment.

Ad 2. We use well-founded induction on the length of derivations. Our base case is the derivation of length zero. We know that the initial problem set P is a set of AUEs and for any AUE in P a rule is applicable. It follows that for the base case holds $P = \varnothing$. Furthermore all the AUEs in S have an empty alignment. Every generalization of two hedges is supporting with respect to an empty alignment. Therefore Corollary 2.19 covers the base case. Let P_0; S_0; $\vartheta_0 \Longrightarrow P_1$; S_1; $\vartheta_0\vartheta_1 \Longrightarrow^* \varnothing$; S_n; ϑ_0^n be a derivation in $\mathfrak{G}_{\mathfrak{a}}^{2\mathcal{V}}$. As induction hypothesis (IH) we assume that

- $\tilde{X}_1(\tilde{x}_1)\vartheta_2^n$ is a supporting generalization of $\tilde{c}_1[\tilde{s}_1]$ and $\tilde{d}_1[\tilde{q}_1]$ with respect to $\mathring{\mathfrak{a}}_1$ for all $\tilde{x}_1\colon \tilde{s}_1 \triangleq \tilde{q}_1$; $\tilde{X}_1\colon \tilde{c}_1 \triangleq \tilde{d}_1$; $\mathfrak{a}_1 \in P_1 \cup S_1$,

By case analysis on the applied rule, we will show that this implies

- $\tilde{X}_0(\tilde{x}_0)\vartheta_1^n$ is a supporting generalization of $\tilde{c}_0[\tilde{s}_0]$ and $\tilde{d}_0[\tilde{q}_0]$ with respect to $\mathring{\mathfrak{a}}_0$ for any arbitrary but fixed $\tilde{x}_0\colon \tilde{s}_0 \triangleq \tilde{q}_0$; $\tilde{X}_0\colon \tilde{c}_0 \triangleq \tilde{d}_0$; $\mathfrak{a}_0 \in P_0 \cup S_0$,

Spl-H. $\begin{aligned}\mathfrak{a}_0 &= a_1\langle i_1{\cdot}I_1, j_1{\cdot}J_1\rangle \dots a_k\langle i_k{\cdot}I_k, j_k{\cdot}J_k\rangle \\ &\quad\ a_{k+1}\langle i_{k+1}{\cdot}I_{k+1}, j_{k+1}{\cdot}J_{k+1}\rangle \dots a_m\langle i_m{\cdot}I_m, j_m{\cdot}J_m\rangle, \\ \vartheta_1 &= \{\tilde{x}_0 \mapsto (\tilde{Y}(\tilde{y}), \tilde{Z}(\tilde{z}))\}.\end{aligned}$

By the IH we know that $\tilde{Y}(\tilde{y})\vartheta_2^n$ is a supporting generalization of $\tilde{s}_0|_{i_1}^{i_k}$ and $\tilde{q}_0|_{j_1}^{j_k}$ with respect to $a_1\langle (i_1 - i_1^{--}){\cdot}I_1, (j_1 - j_1^{--}){\cdot}J_1\rangle \dots a_k\langle (i_k - i_1^{--}){\cdot}I_k, (j_k - j_1^{--}){\cdot}J_k\rangle$, and $\tilde{Z}(\tilde{z})\vartheta_2^n$ is a supporting generalization of $\tilde{s}_0|_{i_k^{++}}^{i_m}$ and $\tilde{q}_0|_{j_k^{++}}^{j_m}$ with respect to $a_{k+1}\langle (i_{k+1} - i_k){\cdot}I_{k+1}, (j_{k+1} - j_k){\cdot}J_{k+1}\rangle \dots a_m\langle (i_m - i_k){\cdot}I_m, (j_m - j_k){\cdot}J_m\rangle$. It follows that $(\tilde{Y}(\tilde{y}), \tilde{Z}(\tilde{z}))\vartheta_2^n = \tilde{x}_0\vartheta_1^n$ is a supporting generalization of $\tilde{s}_0|_{i_1}^{i_m}$ and $\tilde{q}_0|_{j_1}^{j_m}$ with respect to $a_1\langle (i_1 - i_1^{--}){\cdot}I_1, (j_1 - j_1^{--}){\cdot}J_1\rangle \dots a_m\langle (i_m - i_1^{--}){\cdot}I_m, (j_m - j_1^{--}){\cdot}J_m\rangle$. Notice that our contexts are hedges. Hence $\tilde{X}_0(\tilde{x}_0)\vartheta_1^n$ is a supporting generalization of $(\tilde{r}_1, \tilde{s}_0|_{i_1}^{i_m}, \tilde{r}_2)$ and $(\tilde{r}_3, \tilde{q}_0|_{j_1}^{j_m}, \tilde{r}_4)$ with respect to $a_1\langle (i_1 - i_1^{--} + |\tilde{r}_1|){\cdot}I_1, (j_1 - j_1^{--} + |\tilde{r}_3|){\cdot}J_1\rangle \dots a_m\langle (i_m - i_1^{--} + |\tilde{r}_1|){\cdot}I_m, (j_m - j_1^{--} + |\tilde{r}_3|){\cdot}J_m\rangle$, where $\tilde{r}_1, \tilde{r}_2, \tilde{r}_3, \tilde{r}_4$ are arbitrary hedges. Finally we set $\tilde{r}_1 = \tilde{s}_0|_1^{i_1^{--}}$, $\tilde{r}_2 = \tilde{s}_0|_{i_m^{++}}^{|\tilde{s}_0|}$, $\tilde{r}_3 = \tilde{q}_0|_1^{j_1^{--}}$, $\tilde{r}_4 = \tilde{q}_0|_{j_m^{++}}^{|\tilde{q}_0|}$ to get $\tilde{X}_0(\tilde{x}_0)\vartheta_1^n$ is a supporting generalization of $\tilde{s}_0$ and $\tilde{q}_0$ with respect to $a_1\langle i_1{\cdot}I_1, j_1{\cdot}J_1\rangle \dots a_m\langle i_m{\cdot}I_m, j_m{\cdot}J_m\rangle = \mathfrak{a}_0$ and it also is a supporting generalization of $\tilde{c}_0[\tilde{s}_0]$ and $\tilde{d}_0[\tilde{q}_0]$ with respect to $\mathring{\mathfrak{a}}_0$.

App-A. $\begin{aligned}\tilde{s}_0 &= (\tilde{s}_l, a_1(\tilde{s}_1), \tilde{s}_r), \quad \tilde{q}_0 = (\tilde{q}_l, a_1(\tilde{q}_1), \tilde{q}_r), \\ \mathfrak{a}_0 &= a_1\langle i, j\rangle a_2\langle i{\cdot}I_2, j{\cdot}J_2\rangle \ \dots\ a_m\langle i{\cdot}I_m, j{\cdot}J_m\rangle, \\ \vartheta_1 &= \{\tilde{x}_0 \mapsto a_1(\tilde{Y}(\tilde{y}))\}.\end{aligned}$

By IH $\tilde{Y}(\tilde{y})\vartheta_2^n$ is a supporting generalization of $\tilde{s}_1$ and $\tilde{q}_1$ with respect to $a_2\langle I_2, J_2\rangle \ldots a_m\langle I_m, J_m\rangle$. It follows that $a_1(\tilde{Y}(\tilde{y})\vartheta_2^n) = \tilde{x}_0\vartheta_1^n$ is a supporting generalization of the terms $a_1(\tilde{s}_1)$ and $a_1(\tilde{q}_1)$ with respect to $a_1\langle 1, 1\rangle a_2\langle 1{\cdot}I_2, 1{\cdot}J_2\rangle \ldots a_m\langle 1{\cdot}I_m, 1{\cdot}J_m\rangle$. Similarly as for Spl-H we conclude that $\tilde{X}_0(\tilde{x}_0)\vartheta_1^n$ is a supporting generalization of $(\tilde{s}_l, a_1(\tilde{s}_1), \tilde{s}_r)$ and $(\tilde{q}_l, a_1(\tilde{q}_1), \tilde{q}_r)$ with respect to $\mathfrak{a}_0$ where $i = |\tilde{s}_l| + 1$ and $j = |\tilde{q}_l| + 1$. It also is a supporting generalization of $\tilde{c}_0[\tilde{s}_0]$ and $\tilde{d}_0[\tilde{q}_0]$ with respect to $\mathring{\mathfrak{a}}_0$.

Abs-L. $\tilde{x}_0 = \tilde{x}_1$, $\tilde{X}_0 = \tilde{X}_1$,
$\tilde{s}_0 = (\tilde{s}_l, \hbar(\tilde{s}_1), \tilde{s}_r)$, $\tilde{q}_0 = \tilde{q}_1$,
$\tilde{c}_1 = \tilde{c}_0[\tilde{s}_l, \hbar(\circ), \tilde{s}_r]$, $\tilde{d}_0 = \tilde{d}_1$,
$\mathfrak{a}_0 = a_1\langle i{\cdot}I_1, J_1\rangle \ldots a_m\langle i{\cdot}I_m, J_m\rangle$,
$\vartheta_1 = Id$.

It directly follows $\tilde{c}_1[\tilde{s}_1] = \tilde{c}_0[\tilde{s}_l, \hbar(\circ), \tilde{s}_r][\tilde{s}_1] = \tilde{c}_0[\tilde{s}_l, \hbar(\tilde{s}_1), \tilde{s}_r] = \tilde{c}_0[\tilde{s}_0]$. By IH $\tilde{X}_1(\tilde{x}_1)\vartheta_2^n = \tilde{X}_0(\tilde{x}_0)\vartheta_1^n$ is a supporting generalization of $\tilde{c}_1[\tilde{s}_1]$ and $\tilde{d}_1[\tilde{q}_0]$ with respect to $a_1\langle I_1, J_1\rangle \mathring{\ldots} a_m\langle I_m, J_m\rangle$. From Corollary 2.19 we know that $\tilde{X}_0(\tilde{x}_0)\vartheta_1^n$ is a generalization of $\tilde{c}_0[\tilde{s}_0]$ and $\tilde{d}_0[\tilde{q}_0]$. It follows that $\tilde{X}_0(\tilde{x}_0)\vartheta_1^n$ is a supporting generalization of $\tilde{c}_0[\tilde{s}_l, \hbar(\tilde{s}_1), \tilde{s}_r]$ and $\tilde{d}_0[\tilde{q}_0]$ with respect to $\mathring{\mathfrak{a}}_0$, were $i = |\tilde{s}_l| + 1$.

Abs-R. The reasoning is the same as for Abs-L.

Sol-H, Res-C, Mer-S, Clr-S. Those remaining rules operate only on AUEs with empty alignments. Every generalization of two hedges is supporting with respect to an empty alignment. Therefore Corollary 2.19 covers those cases.

Summary. It follows that $\mathfrak{G}_{\mathfrak{a}}^{2\mathcal{V}}$ maintains the invariant that $P \cup S$ is a set of AUEs. Let P be a set of AUEs of the form $\{\tilde{x}\colon \tilde{s} \triangleq \tilde{q};\ \tilde{X}\colon \circ \triangleq \circ;\ \mathfrak{a}\}$ and P; $\varnothing$; $Id \Longrightarrow^+ \varnothing$; S; σ be a derivation in $\mathfrak{G}_{\mathfrak{a}}^{2\mathcal{V}}$. By Definition 2.32 we have $\tilde{X}(\tilde{x})\sigma_{\mathrm{L}}(\{\tilde{x}\colon \tilde{s} \triangleq \tilde{q};\ \tilde{X}\colon \circ \triangleq \circ;\ \mathfrak{a}\}) = \tilde{s}$ and $\tilde{X}(\tilde{x})\sigma_{\mathrm{R}}(\{\tilde{x}\colon \tilde{s} \triangleq \tilde{q};\ \tilde{X}\colon \circ \triangleq \circ;\ \mathfrak{a}\}) = \tilde{q}$. From Lemma 2.18 follows that $\tilde{X}(\tilde{x})\sigma\sigma_{\mathrm{L}}(S) = \tilde{s}$ and $\tilde{X}(\tilde{x})\sigma\sigma_{\mathrm{R}}(S) = \tilde{q}$.

Ad 3. Let P; $\varnothing$; $Id \Longrightarrow^+ \varnothing$; S; σ be a derivation in $\mathfrak{G}_{\mathfrak{a}}^{2\mathcal{V}}$ and $\{\tilde{x}\colon \tilde{s} \triangleq \tilde{q};\ \tilde{X}\colon \circ \triangleq \circ;\ \mathfrak{a}\} \in P$ arbitrary. From above we know that $\tilde{g} = \tilde{X}(\tilde{x})\sigma$ is a supporting generalization of $\tilde{s}$ and $\tilde{q}$ with respect to $\mathfrak{a}$. First we discuss the following property of a rigid generalization:

- There are substitutions σ, ϑ with $\tilde{g}\sigma = \tilde{s}$ and $\tilde{g}\vartheta = \tilde{q}$ such that all the contexts in σ and ϑ are singleton contexts.

The rule Res-C eliminates all those contexts which are hedges from the store S and we already showed that $\tilde{g}\sigma_{\mathrm{L}}(S) = \tilde{s}$ and $\tilde{g}\sigma_{\mathrm{R}}(S) = \tilde{q}$. Therefore we just set $\sigma = \sigma_{\mathrm{L}}(S)$ and $\vartheta = \sigma_{\mathrm{R}}(S)$.

Let $\tilde{r}$ be an arbitrary hedge. We define the predicate $\mathfrak{R}(\tilde{r})$ to be true iff $\tilde{r}$ is a rigid hedge, i.e., the following properties hold:

1. No context variable in $\tilde{s}$ applies to the empty hedge.
2. $\tilde{s}$ doesn't contain consecutive hedge variables.

3. $\tilde{s}$ doesn't contain vertical chains of (context) variables.
4. $\tilde{s}$ doesn't contain context variables with a hedge variable as the first or the last argument (i.e., no subterms of the form $\tilde{X}(\tilde{x}, \dots)$ and $\tilde{X}(\dots, \tilde{x})$).

Let P_0; S_0; $\vartheta_0 \Longrightarrow P_1$; S_1; $\vartheta_0\vartheta_1 \Longrightarrow^* \varnothing$; S_n; ϑ_0^n be an exhaustive derivation in $\mathfrak{G}_{\mathfrak{a}}^{2\mathcal{V}}$. As IH we assume that for all $\tilde{x}_1 \colon \tilde{s}_1 \triangleq \tilde{q}_1$; $\tilde{X}_1 \colon \tilde{c}_1 \triangleq \tilde{d}_1$; $\mathfrak{a}_1 \in P_1$ holds $\mathfrak{R}(\tilde{X}_1(\tilde{x}_1)\vartheta_2^n)$, and furthermore for all $\tilde{y}_1 \colon \tilde{s}_1 \triangleq \tilde{q}_1$; $\tilde{Y}_1 \colon \tilde{c}_1 \triangleq \tilde{d}_1 \in S_1$ holds $\mathfrak{R}(\tilde{y}_1\vartheta_2^n)$ and $\mathfrak{R}(\tilde{Y}_1(\circ)\vartheta_2^n)$. (Note that from $\mathfrak{R}(\tilde{X}_1(\tilde{x}_1)\vartheta_2^n)$ follows $\mathfrak{R}(\tilde{x}_1\vartheta_2^n)$ and $\mathfrak{R}(\tilde{X}_1(\circ)\vartheta_2^n)$.) The trivial base case is the derivation of length zero where $P = \varnothing$ and $\vartheta_1 = Id$.

By case analysis on the applied rule, we will show that this implies for any arbitrary but fixed $\tilde{x}_0 \colon \tilde{s}_0 \triangleq \tilde{q}_0$; $\tilde{X}_0 \colon \tilde{c}_0 \triangleq \tilde{d}_0$; $\mathfrak{a}_0 \in P_0$ holds $\mathfrak{R}(\tilde{X}_0(\tilde{x}_0)\vartheta_1^n)$, and for any $\tilde{y}_0 \colon \tilde{s}_0 \triangleq \tilde{q}_0$; $\tilde{Y}_0 \colon \tilde{c}_0 \triangleq \tilde{d}_0 \in S_0$ holds $\mathfrak{R}(\tilde{y}_0\vartheta_1^n)$ and $\mathfrak{R}(\tilde{Y}_0(\circ)\vartheta_1^n)$.

Abs-L, Abs-R. $P_0 = P \cup \{\tilde{x} \colon \tilde{s}_0 \triangleq \tilde{q}_0;\ \tilde{X} \colon \tilde{c}_0 \triangleq \tilde{d}_0;\ \mathfrak{a}_0\}$,
$P_1 = P \cup \{\tilde{x} \colon \tilde{s}_1 \triangleq \tilde{q}_1;\ \tilde{X} \colon \tilde{c}_1 \triangleq \tilde{d}_1;\ \mathfrak{a}_1\}$,
$S_0 = S_1$, $\vartheta_1 = Id$.

Trivial because $\tilde{X}(\tilde{x})\vartheta_1^n = \tilde{X}(\tilde{x})\vartheta_2^n$ and $S_0 = S_1$.

App-A. $P_0 = P \cup \{\tilde{x} \colon \tilde{s}_0 \triangleq \tilde{q}_0;\ \tilde{X} \colon \tilde{c}_0 \triangleq \tilde{d}_0;\ \mathfrak{a}_0\}$,
$P_1 = P \cup \{\tilde{y} \colon \tilde{s}_1 \triangleq \tilde{q}_1;\ \tilde{Y} \colon \circ \triangleq \circ;\ \mathfrak{a}_1\}$,
$S_0 = S_1 \backslash \{\tilde{x} \colon \epsilon \triangleq \epsilon;\ \tilde{X} \colon \tilde{c}_1 \triangleq \tilde{d}_1\}$,
$\vartheta_1 = \{\tilde{x} \mapsto a_1(\tilde{Y}(\tilde{y}))\}$.

We have $\tilde{X}(\tilde{x})\vartheta_1^n = \tilde{X}(a_1(\tilde{Y}(\tilde{y})))\vartheta_2^n$. By IH it holds $\mathfrak{R}(\tilde{Y}(\tilde{y})\vartheta_2^n)$ and $\mathfrak{R}(\tilde{X}(\circ)\vartheta_2^n)$. It follows that all the properties also hold for $\tilde{X}(a_1(\tilde{Y}(\tilde{y})))\vartheta_2^n$.

Sol-H. $P_0 = P_1 \cup \{\tilde{x} \colon \tilde{s} \triangleq \tilde{q};\ \tilde{X} \colon \circ \triangleq \circ;\ \mathfrak{e}\}$,
$S_0 = S_1 \backslash \{\tilde{x} \colon \tilde{s} \triangleq \tilde{q};\ \tilde{X} \colon \circ \triangleq \circ\}$,
$\vartheta_1 = \{\tilde{X} \mapsto \circ\}$.

Substitution composition gives $\tilde{X}(\tilde{x})\vartheta_1^n = \tilde{x}\vartheta_2^n$ and by IH we have $\mathfrak{R}(\tilde{x}\vartheta_2^n)$.

Mer-S. $P_0 = P_1$,
$S_0 = S \cup \{\tilde{x}_1 \colon \tilde{s} \triangleq \tilde{q};\ \tilde{X}_1 \colon \tilde{c} \triangleq \tilde{d}\} \cup \{\tilde{x}_2 \colon \tilde{s} \triangleq \tilde{q};\ \tilde{X}_2 \colon \tilde{c} \triangleq \tilde{d}\}$,
$S_1 = S \cup \{\tilde{x}_1 \colon \tilde{s} \triangleq \tilde{q};\ \tilde{X}_1 \colon \tilde{c} \triangleq \tilde{d}\}$,
$\vartheta_1 = \{\tilde{x}_2 \mapsto \tilde{x}_1, \tilde{X}_2 \mapsto \tilde{X}_1(\circ)\}$.

Hence $\tilde{x}_2\vartheta_1^n = \tilde{x}_1\vartheta_2^n$ and $\tilde{X}_2(\circ)\vartheta_1^n = \tilde{X}_1(\circ)\vartheta_2^n$. By IH holds that $\mathfrak{R}(\tilde{x}_1\vartheta_2^n)$ and $\mathfrak{R}(\tilde{X}_1(\circ)\vartheta_2^n)$.

Clr-S. $P_0 = P_1$,
$S_0 = S_1 \cup \{\tilde{x} \colon \epsilon \triangleq \epsilon;\ \tilde{X} \colon \circ \triangleq \circ\}$,
$\vartheta_1 = \{\tilde{x} \mapsto \epsilon, \tilde{X} \mapsto \circ\}$.

We have $\tilde{x}\vartheta_1^n = Id$ and $\tilde{X}(\circ)\vartheta_1^n = \circ$, which is trivial.

Res-C. $P_0 = P_1,$
$S_0 = S \cup \{\tilde{x}\colon \epsilon \triangleq \epsilon;\ \tilde{X}\colon (\tilde{s}_l, \dot{c}, \tilde{s}_r) \triangleq (\tilde{q}_l, \dot{d}, \tilde{q}_r)\}$
$S_1 = S \cup \{\tilde{x}\colon \epsilon \triangleq \epsilon; \tilde{X}\colon \dot{c} \triangleq \dot{d},$
$\tilde{y}\colon \tilde{s}_l \triangleq \tilde{q}_l; \tilde{Y}\colon \circ \triangleq \circ,\ \tilde{z}\colon \tilde{s}_r \triangleq \tilde{q}_r; \tilde{Z}\colon \circ \triangleq \circ\},$
$\tilde{s}_l \neq \epsilon$ or $\tilde{s}_r \neq \epsilon$ or $\tilde{q}_l \neq \epsilon$ or $\tilde{q}_r \neq \epsilon,$
$\vartheta_1 = \{\tilde{X} \mapsto (\tilde{y}, \tilde{X}(\circ), \tilde{z})\}.$

Therefore $\tilde{x}\vartheta_1^n = \tilde{x}\vartheta_2^n$ but $\tilde{X}(\circ)\vartheta_1^n = (\tilde{y}, \tilde{X}(\circ), \tilde{z})\vartheta_2^n$. By IH we know that $\mathfrak{R}(\tilde{y}\vartheta_2^n)$, $\mathfrak{R}(\tilde{X}(\circ)\vartheta_2^n)$ and $\mathfrak{R}(\tilde{z}\vartheta_2^n)$ holds. Res-C is the only rule which maps a context variable to a hedge. The rule itself produces AUEs where all the contexts are terms (not hedges). Together with the condition $\tilde{s}_l \neq \epsilon$ or $\tilde{s}_r \neq \epsilon$ or $\tilde{q}_l \neq \epsilon$ or $\tilde{q}_r \neq \epsilon$ it follows that Res-C never applies twice for the same context variable $\tilde{X}$. This considerations guarantee that $\tilde{X}(\circ)\vartheta_2^n$ is a term (not a hedge) which implies that $\mathfrak{R}((\tilde{y}, \tilde{X}(\circ), \tilde{z})\vartheta_2^n)$ holds.

Spl-H. $P_0 = P \cup \{\tilde{x}\colon \tilde{s} \triangleq \tilde{q};\ \tilde{X}\colon \tilde{c} \triangleq \tilde{d};\ \mathfrak{a}_0\}$
$P_1 = P \cup \{\tilde{y}\colon \tilde{s}|_{i_1}^{i_k} \triangleq \tilde{q}|_{j_1}^{j_k};\ \tilde{Y}\colon \circ \triangleq \circ;\ \mathfrak{a}_1,$
$\tilde{z}\colon \tilde{s}|_{i_k^{++}}^{i_m} \triangleq \tilde{q}|_{j_k^{++}}^{j_m};\ \tilde{Z}\colon \circ \triangleq \circ;\ \mathfrak{a}_2\},$
$\mathfrak{a}_0 = a_1\langle i_1{\cdot}I_1, j_1{\cdot}J_1\rangle \ldots a_{k+1}\langle i_{k+1}{\cdot}I_{k+1}, j_{k+1}{\cdot}J_{k+1}\rangle$
$\ldots a_m\langle i_m{\cdot}I_m, j_m{\cdot}J_m\rangle,$
$\mathfrak{a}_1 = a_1\langle (i_1 - i_1^{--}){\cdot}I_1, (j_1 - j_1^{--}){\cdot}J_1\rangle$
$\ldots a_k\langle (i_k - i_1^{--}){\cdot}I_k, (j_k - j_1^{--}){\cdot}J_k\rangle,$
$\mathfrak{a}_2 = a_{k+1}\langle (i_{k+1}{-}i_k){\cdot}I_{k+1}, (j_{k+1}{-}j_k){\cdot}J_{k+1}\rangle$
$\ldots a_m\langle (i_m{-}i_k){\cdot}I_m, (j_m{-}j_k){\cdot}J_m\rangle,$
$i_1 = i_k$ or $j_1 = j_k,$
$i_1 \neq i_{k+1}$ and $j_1 \neq j_{k+1},$
$S_0 = S_1 \backslash \{\tilde{x}\colon \epsilon \triangleq \epsilon;\ \tilde{X}\colon \tilde{c}[\tilde{s}|_1^{i_1^{--}}, \circ, \tilde{s}|_{i_m^{++}}^{|\tilde{s}|}] \triangleq \tilde{d}[\tilde{q}|_1^{j_1^{--}}, \circ, \tilde{q}|_{j_m^{++}}^{|\tilde{q}|}]\},$
$\vartheta_1 = \{\tilde{x} \mapsto (\tilde{Y}(\tilde{y}), \tilde{Z}(\tilde{z}))\}.$

We have $\tilde{X}(\tilde{x})\vartheta_1^n = \tilde{X}(\tilde{Y}(\tilde{y}), \tilde{Z}(\tilde{z}))\vartheta_2^n$. By IH it holds $\mathfrak{R}(\tilde{Y}(\tilde{y})\vartheta_2^n)$, $\mathfrak{R}(\tilde{Z}(\tilde{z})\vartheta_2^n)$ and $\mathfrak{R}(\tilde{X}(\circ)\vartheta_2^n)$. The condition $i_1 \neq i_{k+1}$ implicitly demands that there are at least two elements $a_1\langle i_1{\cdot}I_1, j_1{\cdot}J_1\rangle$ and $a_{k+1}\langle i_{k+1}{\cdot}I_{k+1}, j_{k+1}{\cdot}J_{k+1}\rangle$ in $\mathfrak{a}_0$ and it follows that both alignments $\mathfrak{a}_1$ and $\mathfrak{a}_2$ are nonempty. We already know from above that $\tilde{Y}(\tilde{y})\vartheta_2^n$ and $\tilde{Z}(\tilde{z})\vartheta_2^n$ are supporting generalizations which contain a_1 and a_{k+1} respectively. Now we will show that

- $\tilde{Y}(\tilde{y})\vartheta_2^n$ is a term t_1 (not an arbitrary hedge),
- $\tilde{Z}(\tilde{z})\vartheta_2^n$ is a hedge of the form $\tilde{r}, t_2$ where t_2 is not a hedge variable.

Then it follows that $\mathfrak{R}(t_1, \tilde{r}, t_2)$ holds. Furthermore we have $\tilde{X}(\tilde{Y}(\tilde{y}), \tilde{Z}(\tilde{z}))\vartheta_2^n = \tilde{X}(\circ)\vartheta_2^n[t_1, \tilde{r}, t_2]$ such that also $\mathfrak{R}(\tilde{X}(\circ)\vartheta_2^n[t_1, \tilde{r}, t_2])$ holds because t_1 and t_2 are nonempty terms different from a hedge variable.

Let $\tilde{s}|_{i_1}^{i_k} = (t_{i_1}, \ldots, t_{i_k})$ and $\tilde{q}|_{j_1}^{j_k} = (t_{j_1}, \ldots, t_{j_k})$ where $t_{i_1} = \tilde{s}|_{i_1}$, $t_{i_k} = \tilde{s}|_{i_k}$, $t_{j_1} = \tilde{s}|_{j_1}$, $t_{j_k} = \tilde{s}|_{j_k}$, then all the t's are terms (allowing $t_{i_1} = t_{i_k}$, $t_{j_1} = t_{j_k}$). Furthermore t_{i_1} and t_{j_1} contain the function symbol corresponding to $a_1\langle i_1{\cdot}I_1, j_1{\cdot}J_1\rangle$, and similarly for t_{i_k}, t_{j_k} and $a_k\langle i_k{\cdot}I_k, j_k{\cdot}J_k\rangle$. We know that $\tilde{Y}(\tilde{y})\vartheta_2^n$ is a supporting generalization of $(t_{i_1}, \ldots, t_{i_k})$ and $(t_{j_1}, \ldots, t_{j_k})$ with respect to $\mathfrak{a}_1 = a_1\langle \ldots\rangle \ldots a_k\langle \ldots\rangle$ and therefore $\tilde{Y}(\tilde{y})\vartheta_2^n$ is of the form $(\tilde{r}_1, t_{a_1}, \ldots, t_{a_k}, \tilde{r}_2)$ where t_{a_1}, t_{a_k} denote those terms which contain a_1 and a_k

respectively (allowing $t_{a_1} = t_{a_k}$) and $\tilde{r}_1, \tilde{r}_2$ are arbitrary hedges. To finish this part we have to show that $\tilde{r}_1 = \tilde{r}_2 = \epsilon$. From above we also know that $\tilde{Y}(\tilde{y})\vartheta_2^n\sigma_{\text{L}}(S_n) = (t_{i_1}, \ldots, t_{i_k})$ and therefore $\tilde{r}_1, \tilde{r}_2$ can only contain variables which are mapped to ϵ by $\sigma_{\text{L}}(S_n)$. Because of $\mathfrak{R}(\tilde{Y}(\tilde{y})\vartheta_2^n)$ the hedges $\tilde{r}_1, \tilde{r}_2$ do not contain vertical chains of variables and therefore no context variable can appear there also there are no consecutive hedge variables. The only remaining possibility is that $\tilde{r}_1, \tilde{r}_2$ are either empty or a hedge variable. If both are empty then we are done thus we assume w.l.o.g. that $\tilde{r}_1 = \tilde{x}_1$ to get a contradiction. Then there is an AUE $\{\tilde{x}_1 \colon \epsilon \triangleq \epsilon;\ \tilde{X}_1 \colon \tilde{c}_1 \triangleq \tilde{d}_1\} \in S_n$. As $\tilde{X}_1$ does not appear below $\tilde{x}_1$ it must have been terminated by some mapping $\vartheta_k = \{\tilde{X}_1 \mapsto \circ\}$, $2 \leqslant k \leqslant n$. This implies that also $\tilde{c}_1 = \tilde{d}_1 = \circ$. By assumption the derivation is exhaustive but for $\{\tilde{x}_1 \colon \epsilon \triangleq \epsilon;\ \tilde{X}_1 \colon \circ \triangleq \circ\}$ the rule Clr-S is applicable which is a contradiction.

Let $\tilde{s}|_{i_k^{++}}^{i_m} = (\ldots, t_{i_m})$ and $\tilde{q}|_{j_k^{++}}^{j_m} = (\ldots, t_{j_m})$ where $t_{i_m} = \tilde{s}|_{i_m}$, $t_{j_m} = \tilde{s}|_{j_m}$, then t_{i_m} and t_{j_m} are terms. Furthermore t_{i_m} and t_{j_m} contain the function symbol corresponding to $a_m\langle i_m{\cdot}I_m, j_m{\cdot}J_m\rangle$. The further reasoning is the same as above. It follows that all the properties are preserved when composing $\tilde{X}(\tilde{Y}(\tilde{y}), \tilde{Z}(\tilde{z}))\vartheta_2^n$. □

The next theorem is the Completeness Theorem. It, essentially, says that for a given alignment of two input hedges, a rigid generalization which is computed by $\mathfrak{G}_{\mathfrak{a}}^{2\mathcal{V}}$ is least general among all rigid generalizations of the same input.

Theorem 2.22 (Completeness)**.** *Let $\tilde{g}$ be a rigid generalization of $\tilde{s}$ and $\tilde{q}$ with respect to $\mathfrak{a}$. Then there exists a derivation $\{\tilde{x} \colon \tilde{s} \triangleq \tilde{q};\ \tilde{X} \colon \circ \triangleq \circ;\ \mathfrak{a}\};\ \varnothing;\ Id \Longrightarrow^{+} \varnothing;\ S;\ \sigma$ obtained by $\mathfrak{G}_{\mathfrak{a}}^{2\mathcal{V}}$ such that $\tilde{g} \preceq \tilde{X}(\tilde{x})\sigma$.*

Proof. Let ϑ_1, ϑ_2, σ be substitutions, $\tilde{x} \in \mathcal{V}_{\mathsf{H}}$, $\tilde{X} \in \mathcal{V}_{\mathsf{C}}$ fresh variables and S a set of AUEs such that:

- $\text{Dom}(\vartheta_1) = \text{Dom}(\vartheta_2)$,
- $\tilde{g}\vartheta_1 = \tilde{s}$,
- $\tilde{g}\vartheta_2 = \tilde{q}$,
- $\sigma = \{\tilde{x} \mapsto \tilde{g}, \tilde{X} \mapsto \circ\}$,
- $S = \{\tilde{y} \colon \tilde{y}\vartheta_1 \triangleq \tilde{y}\vartheta_2;\ \tilde{Y} \colon \circ \triangleq \circ \mid \tilde{y} \in \text{Dom}(\vartheta_1), \tilde{Y} \text{ is fresh}\} \cup$
  $\{\tilde{y} \colon \epsilon \triangleq \epsilon;\ \tilde{Y} \colon \tilde{Y}\vartheta_1 \triangleq \tilde{Y}\vartheta_2 \mid \tilde{Y} \in \text{Dom}(\vartheta_1), \tilde{y} \text{ is fresh}\}$.

We construct the final state $P;\ S;\ \sigma$ where $P = \varnothing$, $\tilde{X}(\tilde{x})\sigma = \tilde{g}$ and furthermore we have $\tilde{s} = \tilde{X}(\tilde{x})\sigma\vartheta_1 = \tilde{X}(\tilde{x})\sigma\sigma_{\text{L}}(S)$ and $\tilde{q} = \tilde{X}(\tilde{x})\sigma\vartheta_2 = \tilde{X}(\tilde{x})\sigma\sigma_{\text{R}}(S)$. We start from this final state and we show that a rule R is applicable in reverse direction $P_0;\ S_0;\ \sigma_0 \ \mathsf{R}\Longleftarrow P_1; S_1; \sigma_1$ until we reach the state $\{\tilde{x}_0 \colon \tilde{s}_0 \triangleq \tilde{q}_0;\ \tilde{X}_0 \colon \circ \triangleq \circ;\ \mathfrak{a}_0\} \in P_0$ and $\tilde{g}_0 = \tilde{X}_0(\tilde{x}_0)\sigma_0 = \tilde{X}_0(\tilde{x}_0)$ such that, by the invariant Lemma 2.18, we will get the desired result.

First, all the variables in $\tilde{g}_1$ are made distinct by reversely applying the rule Mer-S. We also keep the store sound like described in the Mer-S rule. Now we introduce some additional fresh variables, denoted by $\tilde{Y}$ and $\tilde{y}$, to ensure that above every function symbol there is a context variable and that every leaf is a hedge variable such that $\tilde{g}_0$ is still a rigid generalization of $\tilde{s}_1$ and $\tilde{q}_1$ with respect to $\mathfrak{a}_1$. This can be done by

applying the rule Clr-S in reverse and we use the following transformation rules to build $\tilde{g}_0 = \phi(\tilde{g}_1)$:

$$\begin{array}{lll} \phi(\tilde{s}) = \tilde{Y}(\psi(\tilde{s})), \text{ if } \mathrm{Top}(\tilde{s}) \in \mathcal{F}, & \phi(\tilde{s}) = \psi(\tilde{s}), \text{ if } \mathrm{Top}(\tilde{s}) \notin \mathcal{F}. & \\ \hline \psi(\tilde{x}) = \tilde{x}, & \psi(a) = a(\tilde{y}), & \psi(f(t_1, \ldots, t_n))_{n>0} = f(\phi(t_1), \ldots, \phi(t_n)), \\ \psi(\tilde{X}(\tilde{s})) = \tilde{X}(\psi(\tilde{s})), & & \psi(t_1, \ldots, t_n)_{n>1} = \phi(t_1), \ldots, \phi(t_n). \end{array}$$

The next step is to apply the rule Res-C in reverse direction as long as possible, leaving only those hedge variables which occur as singleton terms under a function symbol, e.g., those which occur in subterms of the form $f(\tilde{x})$. We still have $\tilde{g}_0$ being a rigid generalization of $\tilde{s}_1$ and $\tilde{q}_1$ with respect to $\mathfrak{a}_1$.

Now we move all the AUEs $\tilde{x}_1 \colon \tilde{s}_1 \triangleq \tilde{q}_1;\ \tilde{X}_1 \colon \circ \triangleq \circ$ with $\tilde{x}_1 \in \mathcal{V}_{\mathsf{H}}(\tilde{g})$ from S_1 to P_0 by reversely applying the rule Sol-H. After this step, all the leafs are hedge variables and together with their two predecessors they are subterms of the form $f(\tilde{X}_1(\tilde{x}_1))$. There is one AUE in P_0 for every subterm (leaf) of this form.

We will show that one of the rules App-A, Abs-L/Abs-R, Spl-H is applicable in reverse direction until we get $P_0 = \{\tilde{x}_0 \colon \tilde{s}_0 \triangleq \tilde{q}_0;\ \tilde{X}_0 \colon \circ \triangleq \circ;\ \mathfrak{a}_0\}$ and $\tilde{X}_0(\tilde{x}_0)\sigma_0 = \tilde{X}_0(\tilde{x}_0)$.

First, the rule App-A is applicable in reverse to every $\{\tilde{x}_1 \colon \tilde{s}_1 \triangleq \tilde{q}_1;\ \tilde{X}_1 \colon \circ \triangleq \circ;\ \mathfrak{e}\} \in P$. It adds one function symbol to the alignment $\mathfrak{a}_0$ and removes it from $\tilde{g}_1$. Furthermore, the hole $\circ$ in $\tilde{c}_0, \tilde{d}_0$ has no siblings for all the new AUEs $\{\tilde{x}_0 \colon \tilde{s}_0 \triangleq \tilde{q}_0;\ \tilde{X}_0 \colon \tilde{c}_0 \triangleq \tilde{d}_0;\ \mathfrak{a}_0\} \in P_0$. Every reverse application of App-A strictly decreases the size of $\tilde{g}_0$. Afterwards all the leafs are of the form $\tilde{X}_0(\tilde{x}_0)$ and there is an AUE $\{\tilde{x}_0 \colon \tilde{s}_0 \triangleq \tilde{q}_0;\ \tilde{X}_0 \colon \tilde{c}_0 \triangleq \tilde{d}_0;\ \mathfrak{a}_0\} \in P_0$, where the hole $\circ$ in $\tilde{c}_0, \tilde{d}_0$ has no siblings.

- *Case $\tilde{c}_1 \neq \circ$ or $\tilde{d}_1 \neq \circ$:* By looking at the rules App-A, Abs-L/Abs-R, Spl-H it is trivial to see that, if $\tilde{c}_1 \neq \circ$ or $\tilde{d}_1 \neq \circ$ then the only possible reverse application is Abs-L/Abs-R. Furthermore the rule is always applicable in this situation because the property that the hole $\circ$ in $\tilde{c}_1, \tilde{d}_1$ has no siblings is maintained by every reverse rule application and from that it follows that a function is applied to either $\tilde{c}_1$ or $\tilde{d}_1$. Abs-L/Abs-R maintains the property that the hole $\circ$ in $\tilde{c}_0, \tilde{d}_0$ has no siblings such that we have the same situation with the possible rule applications App-A, Abs-L/Abs-R, Spl-H as above. Obviously Abs-L/Abs-R can only be applied finitely many times because it strictly decreases the size of $\tilde{c}_0$ or $\tilde{d}_0$.
- *Case $\tilde{c}_1 = \tilde{d}_1 = \circ$ and $\tilde{X}_1$ has no siblings:* Then there are two possibilities: Either $\tilde{g}_1 = \tilde{X}_1(\tilde{x}_1)$ and we are done, or a function is applied to $\tilde{X}_1(\tilde{x}_1)$ like $f(\tilde{X}_1(\tilde{x}_1))$ so that the rule App-A is applicable again (see above).
- *Case $\tilde{c}_1 = \tilde{d}_1 = \circ$ and $\tilde{X}_1$ has siblings:* Let $\tilde{X}_1, \ldots, \tilde{X}_n, n > 1$ be the siblings. If $n = 2$ and $\tilde{g}$ is of the form $\tilde{X}_0(\tilde{X}_1(\tilde{x}_1), \tilde{X}_2(\tilde{x}_2))$, then Spl-H can directly be applied in reverse order. Otherwise we use Clr-S to introduce a fresh context variable $\tilde{Y}$ above $\tilde{X}_{n-1}$ and $\tilde{X}_n$ leading to the form $\tilde{X}_1, \ldots, \tilde{X}_{n-2}, \tilde{Y}(\tilde{X}_{n-1}(\tilde{x}_{n-1}), \tilde{X}_n(\tilde{x}_n))$ such that Spl-H is applicable in reverse order to $\tilde{Y}(\tilde{X}_{n-1}(\tilde{x}_{n-1}), \tilde{X}_n(\tilde{x}_n))$. This strictly decreases the size of $\tilde{g}_0$.

Eventually we get $\tilde{g}_0$ being of the form $\tilde{X}_0(\tilde{x}_0)$ when applying this strategy exhaustively. And we know that $\{\tilde{x}_0 \colon \tilde{s}_0 \triangleq \tilde{q}_0;\ \tilde{X}_0 \colon \circ \triangleq \circ;\ \mathfrak{a}_0\} \in P_0$. By the invariant *Lemma* 2.18 we also know that $\tilde{s} = \tilde{X}(\tilde{x})\sigma_0\sigma_{\mathrm{L}}(P_0 \cup S_0) = \tilde{X}_0(\tilde{x}_0)\sigma_{\mathrm{L}}(P_0 \cup S_0)$ and $\tilde{q} = \tilde{X}(\tilde{x})\sigma_0\sigma_{\mathrm{R}}(P_0 \cup S_0) = \tilde{X}_0(\tilde{x}_0)\sigma_{\mathrm{R}}(P_0 \cup S_0)$. This gives us the result $\tilde{s}_0 = \tilde{s}$ and $\tilde{q}_0 = \tilde{q}$ which proves the existence of a derivation in $\mathfrak{G}^{2\mathcal{V}}_{\mathfrak{a}}$. □

The algorithm is nondeterministic. The Uniqueness Theorem says that different transformations compute generalizations which are equivalent modulo $\simeq$, i.e., differ from each other only by variable renaming:

Theorem 2.23 (Uniqueness modulo $\simeq$)**.** *Let $\mathfrak{a}$ be an admissible alignment of $\tilde{s}$ and $\tilde{q}$. If $\{\tilde{x}_1 \colon \tilde{s} \triangleq \tilde{q};\ \tilde{X}_1 \colon \circ \triangleq \circ;\ \mathfrak{a}\};\ \varnothing;\ Id \Longrightarrow^{+} \varnothing;\ S_1;\ \sigma_1$ and $\{\tilde{x}_2 \colon \tilde{s} \triangleq \tilde{q};\ \tilde{X}_2 \colon \circ \triangleq \circ;\ \mathfrak{a}\};\ \varnothing;\ Id \Longrightarrow^{+} \varnothing;\ S_2;\ \sigma_2$ are two exhaustive derivations in $\mathfrak{G}_{\mathfrak{a}}^{2\mathcal{V}}$, then $\tilde{X}_1(\tilde{x}_1)\sigma_1 \simeq \tilde{X}_2(\tilde{x}_2)\sigma_2$.*

Proof. By Newman's lemma [64] and Theorem 2.20, it suffices to show local confluence. Let $P_0;\ S_0;\ \sigma_0$ be an arbitrary state in $\mathfrak{G}_{\mathfrak{a}}^{2\mathcal{V}}$. We show that for any two rule applications $P_0;\ S_0;\ \sigma_0 \Longrightarrow_{\mathsf{R}} P_1; S_1; \sigma_0\vartheta_1$ and $P_0;\ S_0;\ \sigma_0 \Longrightarrow_{\mathsf{R}'} P_1'; S_1'; \sigma_0\vartheta_1'$ there are derivations such that $P_1; S_1; \sigma_0\vartheta_1 \Longrightarrow^{*} P_i; S_i; \sigma_i \ {}^{*}\!\Longleftarrow P_1'; S_1'; \sigma_0\vartheta_1'$. It is easy to see that if the rules R and R' operate on different AUEs then it holds, for $P_1; S_1; \sigma_0\vartheta_1 \Longrightarrow_{\mathsf{R}'} P_2; S_2; \sigma_0\vartheta_1\vartheta_2$ and $P_1'; S_1'; \sigma_0\vartheta_1' \Longrightarrow_{\mathsf{R}} P_2'; S_2'; \sigma_0\vartheta_1'\vartheta_2'$, that $P_2 = P_2'$, $S_2 = S_2'$ and $\vartheta_1\vartheta_2 = \vartheta_1'\vartheta_2'$ (using the corresponding names for fresh variables), because the variables of the AUEs are disjoint (by Lemma 2.16). Therefore we assume that both, R and R', operate on an arbitrary but fixed AUE $\{\tilde{x}_0 \colon \tilde{s}_0 \triangleq \tilde{q}_0;\ \tilde{X}_0 \colon \tilde{c}_0 \triangleq \tilde{d}_0;\ \mathfrak{a}_0\} \in P_0 \cup S_0$.

If R is one of Spl-H, App-A, Sol-H, then no other rule is applicable to the selected AUE and it follows that $\mathsf{R}' = \mathsf{R}$. If $\mathsf{R} =$ Abs-L then either $\mathsf{R}' = \mathsf{R}$ or $\mathsf{R}' =$ Abs-R. The first case is the trivial one. In the latter case, the rules Abs-L and Abs-R operate on different sides of the equations such that it does not matter which one is applied first, if both are applicable at the same time. Therefore we obtain for $P_1; S_1; \sigma_0 \Longrightarrow_{\mathsf{R}'} P_2; S_2; \sigma_0$ and $P_1'; S_1'; \sigma_0 \Longrightarrow_{\mathsf{R}} P_2'; S_2'; \sigma_0$, that $P_2 = P_2'$ and $S_2 = S_2'$. Obviously the same reasoning holds for $\mathsf{R} =$ Abs-R. It remains to show local confluence for the cases $\mathsf{R} =$ Res-C, $\mathsf{R} =$ Clr-S and $\mathsf{R} =$ Mer-S.

$\mathsf{R} =$ Res-C. $S_0 = S \cup \{\tilde{x}_1 \colon \epsilon \triangleq \epsilon;\ \tilde{X}_1 \colon (\tilde{s}_l, \dot{c}, \tilde{s}_r) \triangleq (\tilde{q}_l, \dot{d}, \tilde{q}_r)\}$,
$S_1 = S \cup \{\tilde{x}_1 \colon \epsilon \triangleq \epsilon; \tilde{X}_1 \colon \dot{c} \triangleq \dot{d}\} \cup$
$\{\tilde{y}_1 \colon \tilde{s}_l \triangleq \tilde{q}_l; \tilde{Y}_1 \colon \circ \triangleq \circ, \tilde{z}_1 \colon \tilde{s}_r \triangleq \tilde{q}_r; \tilde{Z}_1 \colon \circ \triangleq \circ\}$,
$\tilde{s}_l \neq \epsilon$ or $\tilde{s}_r \neq \epsilon$ or $\tilde{q}_l \neq \epsilon$ or $\tilde{q}_r \neq \epsilon$,
$\vartheta_1 = \{\tilde{X}_1 \mapsto (\tilde{y}_1, \tilde{X}_1(\circ), \tilde{z}_1)\}$.

The rule Clr-S is not applicable for the selected AUE because of the condition $\tilde{s}_l \neq \epsilon$ or $\tilde{s}_r \neq \epsilon$ or $\tilde{q}_l \neq \epsilon$ or $\tilde{q}_r \neq \epsilon$. Therefore the only nontrivial case is $\mathsf{R}' =$ Mer-S, and from the condition of Mer-S we know that there is an AUE $\{\tilde{x}_2 \colon \epsilon \triangleq \epsilon;\ \tilde{X}_2 \colon (\tilde{s}_l, \dot{c}, \tilde{s}_r) \triangleq (\tilde{q}_l, \dot{d}, \tilde{q}_r)\} \in S$ which is the second one selected by Mer-S. To show local confluence we select this AUE and apply Res-C again $P_1; S_1; \sigma_0\vartheta_1 \Longrightarrow_{\text{Res-C}} P_2; S_2; \sigma_0\vartheta_1\vartheta_2$, with $\vartheta_2 = \{\tilde{X}_2 \mapsto (\tilde{y}_2, \tilde{X}_2(\circ), \tilde{z}_2)\}$ and $P_2 = P_1 = P_0$. Furthermore we get $\{\tilde{x}_2 \colon \epsilon \triangleq \epsilon; \tilde{X}_2 \colon \dot{c} \triangleq \dot{d}, \tilde{y}_2 \colon \tilde{s}_l \triangleq \tilde{q}_l; \tilde{Y}_2 \colon \circ \triangleq \circ, \tilde{z}_2 \colon \tilde{s}_r \triangleq \tilde{q}_r; \tilde{Z}_2 \colon \circ \triangleq \circ\} \subset S_2$. W.l.o.g. let $P_0;\ S_0;\ \sigma_0 \Longrightarrow_{\mathsf{R}'} P_1'; S_1'; \sigma_0\vartheta_1'$ such that $P_1' = P_0$, $S_1' = S$ and $\vartheta_1' = \{\tilde{x}_1 \mapsto \tilde{x}_2, \tilde{X}_1 \mapsto \tilde{X}_2(\circ)\}$. Now we continue with $P_2; S_2; \sigma_0\vartheta_1\vartheta_2$ and apply the rule Mer-S three times to obtain $P_2; S_2; \sigma_0\vartheta_1\vartheta_2 \Longrightarrow^{3}_{\text{Mer-S}} P_i; S_i; \sigma_0\vartheta_1\vartheta_2\{\tilde{x}_1 \mapsto \tilde{x}_2, \tilde{X}_1 \mapsto \tilde{X}_2(\circ)\}\{\tilde{y}_1 \mapsto \tilde{y}_2, \tilde{Y}_1 \mapsto \tilde{Y}_2(\circ)\}\{\tilde{z}_1 \mapsto \tilde{z}_2, \tilde{Z}_1 \mapsto \tilde{Z}_2(\circ)\}$. Finally we apply to $P_1'; S_1'; \sigma_0\vartheta_1'$ the rule Res-C such that we get $P_1'; S_1'; \sigma_0\vartheta_1' \Longrightarrow_{\text{Res-C}} P_i; S_i; \sigma_0\vartheta_1'\{\tilde{X}_2 \mapsto (\tilde{y}_2, \tilde{X}_2(\circ), \tilde{z}_2)\}$. It remains to compare the two obtained substitutions $\sigma_i = \{\tilde{X}_1 \mapsto (\tilde{y}_1, \tilde{X}_1(\circ), \tilde{z}_1)\}\{\tilde{X}_2 \mapsto (\tilde{y}_2, \tilde{X}_2(\circ), \tilde{z}_2)\}\{\tilde{x}_1 \mapsto \tilde{x}_2, \tilde{X}_1 \mapsto \tilde{X}_2(\circ)\}$ $\{\tilde{y}_1 \mapsto \tilde{y}_2, \tilde{Y}_1 \mapsto \tilde{Y}_2(\circ)\}\{\tilde{z}_1 \mapsto \tilde{z}_2, \tilde{Z}_1 \mapsto \tilde{Z}_2(\circ)\}$ and $\sigma_i' = \{\tilde{x}_1 \mapsto \tilde{x}_2, \tilde{X}_1 \mapsto \tilde{X}_2(\circ)\}\{\tilde{X}_2 \mapsto (\tilde{y}_2, \tilde{X}_2(\circ), \tilde{z}_2)\}$. Therefore we compose both substitutions exhaustively: $\sigma_i = \{\tilde{x}_1 \mapsto \tilde{x}_2,$ $\tilde{X}_1 \mapsto (\tilde{y}_2, \tilde{X}_2(\circ), \tilde{z}_2), \tilde{X}_2 \mapsto (\tilde{y}_2, \tilde{X}_2(\circ), \tilde{z}_2), \tilde{y}_1 \mapsto \tilde{y}_2, \tilde{Y}_1 \mapsto \tilde{Y}_2(\circ), \tilde{z}_1 \mapsto \tilde{z}_2, \tilde{Z}_1 \mapsto \tilde{Z}_2(\circ)\}$

and $\sigma_i' = \{\tilde{x}_1 \mapsto \tilde{x}_2, \tilde{X}_1 \mapsto (\tilde{y}_2, \tilde{X}_2(\circ), \tilde{z}_2), \tilde{X}_2 \mapsto (\tilde{y}_2, \tilde{X}_2(\circ), \tilde{z}_2)\}$. As the fresh variables $\tilde{y}_1, \tilde{Y}_1, \tilde{z}_1, \tilde{Z}_1$ have been introduced during this reasoning process and all the AUEs containing one of those variables have been eliminated by applications of the rule Mer-S, the extra mappings in σ_i do not have any effect and may also be omitted.

R = Clr-S. $S_1 = S_0 \backslash \{\tilde{x}_1 \colon \epsilon \triangleq \epsilon;\ \tilde{X}_1 \colon \circ \triangleq \circ\}$,
$\vartheta_1 = \{\tilde{x}_1 \mapsto \epsilon, \tilde{X}_1 \mapsto \circ\}$.

For the same reason as above, the rule Res-C is not applicable for the selected AUE such that the only nontrivial case is $\mathsf{R}' =$ Mer-S. The reasoning can be done similarly to the case $\mathsf{R} =$ Res-C. We leave this easy exercise to the reader.

R = Mer-S. $S_0 = S \cup \{\tilde{x}_1 \colon \tilde{s} \triangleq \tilde{q};\ \tilde{X}_1 \colon \tilde{c} \triangleq \tilde{d}\} \cup \{\tilde{x}_2 \colon \tilde{s} \triangleq \tilde{q};\ \tilde{X}_2 \colon \tilde{c} \triangleq \tilde{d}\}$,
$S_1 = S \cup \{\tilde{x}_2 \colon \tilde{s} \triangleq \tilde{q};\ \tilde{X}_2 \colon \tilde{c} \triangleq \tilde{d}\}$,
$\vartheta_1 = \{\tilde{x}_1 \mapsto \tilde{x}_2, \tilde{X}_1 \mapsto \tilde{X}_2(\circ)\}$.

If $\mathsf{R} =$ Mer-S then there are the three cases $\mathsf{R}' = \mathsf{R}$, $\mathsf{R}' =$ Res-C and $\mathsf{R}' =$ Clr-S. The first case is the trivial one and we already showed local confluence for the other two cases. □

2.2.7 Complexity Analysis of $\mathfrak{G}_{\mathfrak{a}}^{2\mathcal{V}}$

The Complexity Theorem gives upper bounds for the computational complexity and for the required space of the algorithm $\mathfrak{G}_{\mathfrak{a}}^{2\mathcal{V}}$.

Theorem 2.24 (Complexity)**.** *The anti-unification algorithm $\mathfrak{G}_{\mathfrak{a}}^{2\mathcal{V}}$ has $O(n^2)$ time complexity and $O(n)$ space complexity, where n is the number of symbols in the input.*

Proof. Let $P_0; S_0; \sigma_0 = \{\tilde{x} \colon \tilde{s} \triangleq \tilde{q};\ \tilde{X} \colon \circ \triangleq \circ; \mathfrak{a}\}; \varnothing; Id$ be the initial state of $\mathfrak{G}_{\mathfrak{a}}^{2\mathcal{V}}$ and $P_{i-1}; S_{i-1}; \sigma_{i-1} \Longrightarrow P_i; S_i; \sigma_i$ an arbitrary rule application. By Theorem 2.23 we can arrange the rule applications as we like to obtain a maximal derivation. First the rules Spl-H, Abs-L/Abs-R, App-A and Sol-H are applied exhaustively. This are the only rules that operate on P_{i-1} and furthermore they do not have conditions on S_{i-1} or σ_{i-1} such that $P_0; S_0; \sigma_0 \Longrightarrow^+ \varnothing; S_j; \sigma_j$, for some j. Afterwards they are not applicable again and Res-C is applied exhaustively $\varnothing; S_j; \sigma_j \Longrightarrow^*_{\mathsf{Res\text{-}C}} \varnothing; S_k; \sigma_k$. It transforms all the contexts in the store to terms. The rules Clr-S and Mer-S operate on S_k but they only remove AUEs from there such that Res-C will not be applicable again. Finally we postpone the application of Mer-S to the very end, leading to a partial derivation $\varnothing; S_k; \sigma_k \Longrightarrow^*_{\mathsf{Clr\text{-}S}} \varnothing; S_l; \sigma_l \Longrightarrow^*_{\mathsf{Mer\text{-}S}} \varnothing; S_n; \sigma_n$ where no more rule is applicable because Mer-S does not introduce any AUEs, to which another rule could apply.

Now we analyze the first phase $P_0; S_0; \sigma_0 \Longrightarrow^+ \varnothing; S_j; \sigma_j$. The rule Spl-H splits an AUE into two AUEs and moves some parts into the store. The space overhead for one application is constant because the two new AUEs in P_i and the one in S_i together exactly cover the original one from P_{i-1}, and four new variables are introduced. It can be applied $O(n)$ many times because both of the new AUEs are nonempty. It needs linear time (by the length of the alignment) to check for applicability and find the position for splitting the AUE. Also the context application needs linear time. The rules Abs-L/Abs-R are also applicable $O(n)$ many times. They strictly reduce the size

of a hedge in P_i. The space overhead is zero. The test for applicability, the context application as well as the operations on the alignment need linear time. App-A is applicable $O(n)$ many times as well and one application needs linear time and constant space. It strictly reduces the size of a hedge in P_i and one application needs linear time, for the same reasons as the above rules. As Spl-H is applicable at most $O(n)$ many times and doubles the elements of P_i at each application and all the other rules do not increase the length of P_i, Sol-H is applicable $O(n)$ many times too. It follows that the number of introduced variables is $O(n)$ and the size of S_j is also bound by $O(n)$.

We compose the substitution σ_i immediately, but we only keep the mappings for $\tilde{x}$ and $\tilde{X}$ in σ_i such that $\sigma_i = \{\tilde{x} \mapsto \tilde{r}_i, \tilde{X} \mapsto \tilde{c}_i\}$, for some $\tilde{r}_i, \tilde{c}_i$. As all the introduced variables in Spl-H and App-A are fresh, they only appear once in $\tilde{r}_i$ or $\tilde{c}_i$. This invariant of the first phase leads to $O(n)$ size of σ_i as well as $O(n)$ time for the substitution composition in Spl-H, App-A and Sol-H. All together we get $O(n^2)$ time complexity and $O(n)$ space complexity for the first phase.

The second phase is $\varnothing; S_j; \sigma_j \Longrightarrow^*_{\mathsf{Res\text{-}C}} \varnothing; S_k; \sigma_k$. The rule Res-C is applicable only once per AUE leading to $O(n)$ many applications. The space overhead is constant at each application, introducing four fresh variables. It needs linear time at each application. We again compose σ_i immediately and for similar reasons as above, the substitution composition in Res-C only needs $O(n)$ time, leading to an overall time complexity of $O(n^2)$ and space complexity $O(n)$.

From the $O(n)$ size of the store, it follows that also the store cleaning rule is applicable $O(n)$ many times and the overall time complexity of this phase is $O(n^2)$, as we compose substitutions immediately like before. The space overhead for Clr-S is zero.

It remains to show that $\varnothing; S_l; \sigma_l \Longrightarrow^*_{\mathsf{Mer\text{-}S}} \varnothing; S_n; \sigma_n$ only needs $O(n^2)$ time. Therefore we postpone substitution composition. Comparing $O(n) * O(n)$ AUEs in the store needs $O(n^2)$ time and removing an AUE from the store needs constant time using a linked list. As the size of the store is bound by $O(n)$ and Mer-S removes one AUE at each application, there are $O(n)$ postponed substitution compositions. Each of them of constant size as they all are just variable renamings. This leads to linear space overhead and we have to compose $O(n)$ substitutions where each composition needs $O(n)$ time.

This concludes our complexity analysis where we showed that the algorithm runs in $O(n^2)$ time using $O(n)$ space. □

2.2.8 Computing Admissible Alignments

The algorithm $\mathfrak{G}^{2\mathcal{V}}_{\mathfrak{a}}$ is independent from the alignment computation. However, from the application point of view it is interesting to experiment with various such functions and identify practically well-behaving ones. It is not our goal to give here a survey of possible alignment computation functions. We just mention that to have $O(n^2)$ runtime complexity for computing higher-order generalizations (the best we can hope for due to Theorem 2.24), we could use some known techniques to compute an admissible alignment in quadratic time, e.g., as a constrained longest common subforest [5, 87] or an agreement subhedge/subtree [46, 67].

In this section we just want to give an idea of alignment computation by discussing a certain example, which is not very efficient, but it is simple and is implemented in our

Java library. The function computes admissible alignments of *longest length* for two given input hedges.

Since an alignment can be seen as a common subsequence of the word representation of two input hedges (see Definition 2.22), computing the complete set of all admissible alignments can be done by a generate and test method: generate all the common subsequences of the word representation of the input hedges and test them for admissibility (see Definition 2.24). However, this approach is not tractable because the number of subsequences of one sequence is already exponential in its length. Therefore it makes sense to reduce the number of generated common subsequences. One approach might be to compute some admissible alignments of longest length for two input hedges. (Notice, that this approach will not lead to a complete set of rigid lggs because there might be incomparable rigid lggs which contain less common function symbols.)

Definition 2.33 (Longest admissible alignment)**.** *A* longest admissible alignment *(laa) of two hedges is their admissible alignment with a longest length. I.e., an admissible alignment* $\mathfrak{a}$ *of two hedges* $\tilde{s}$ *and* $\tilde{q}$ *is an laa iff* $|\mathfrak{a}| \geqslant |\mathfrak{a}'|$ *for all admissible alignments* $\mathfrak{a}'$ *of the hedges* $\tilde{s}$ *and* $\tilde{q}$*.*

A complete set of longest admissible alignments can be computed with the aid of a function that for two hedges, $\tilde{s}$ and $\tilde{q}$, returns a set of alignments of length $k \in \mathbb{N}$. We denote this function by $\mathcal{Cs}(\tilde{s}, \tilde{q}, k)$ because it computes common subsequences of length k for the word representations of the input hedges $\tilde{s}$ and $\tilde{q}$. Given a set of alignments A, the subset of all admissible alignments is denoted by admissible(A). We formulate a simple generate and test algorithm by setting k initially to the length of the longest alignment (see, e.g., [50]) of two input hedges $\tilde{s}$ and $\tilde{q}$, and successively reducing k until we get a *nonempty* subset of *admissible* alignments from $\mathcal{Cs}(\tilde{s}, \tilde{q}, k-i)$, for some $0 \leqslant i \leqslant k$. The algorithm can be described by four simple steps:

1. $k :=$ Length of longest alignment of $\tilde{s}$ and $\tilde{q}$.
2. $A := \mathcal{Cs}(\tilde{s}, \tilde{q}, k)$.
3. If admissible(A) $= \varnothing$ then

   $k := k - 1$ and goto step 2.
4. return admissible(A).

This approach can still lead to iterating exponentially many alignments for two given input hedges. Consider a longest alignment of length k which is not admissible. By successively reducing the length to $k-i$, there are $\binom{k}{k-i}$ possible alignments which have to be tested for admissibility. However, our implementation (see section 4.3) offers this approach as an example of computing a set of admissible alignments for two input hedges.

2.2.9 Minimization by Anti-Unification using $\mathfrak{G}_{\mathfrak{a}}^{2\mathcal{V}}$

The algorithm $\mathfrak{G}_{\mathfrak{a}}^{2\mathcal{V}}$ computes a rigid lgg, which corresponds to a certain admissible alignment for two given hedges. An alignment computation function, like the one described in subsection 2.2.8, may return a finite set A of admissible alignments for the given input hedges. It may even happen that one alignment in A is a subsequence of

another one. To minimize the set of generalizations which is computed for two input hedges and a given set of alignments, we need to solve a higher-order matching problem for hedges, instantiating context and hedge variables.

Again we use the idea of solving that matching problem by $\mathfrak{G}_{\mathfrak{a}}^{2\mathcal{V}}$ itself. Lemma 2.4 says that for two hedges that are equigeneral $\tilde{s} \simeq \tilde{q}$ holds that $|\mathrm{Pos}_f(\tilde{s})| = |\mathrm{Pos}_f(\tilde{q})|$, for all $f \in \mathcal{F}$. It is stated for first-order unranked terms but trivially generalizes to the higher-order case. From Lemma 1.8 we know that deciding $\tilde{s} \stackrel{?}{\simeq} \tilde{q}$ (without using a universal matching algorithm) is crucial in order to solve a matching problem by $\mathfrak{G}_{\mathfrak{a}}^{2\mathcal{V}}$. We introduce some kind of "normal form", namely the compressed form, that enables us to decide $\tilde{s} \stackrel{?}{\simeq} \tilde{q}$ in the same manner as in subsection 2.1.4.

Definition 2.34. *A hedge $\tilde{s}$ is in* compressed form[†] *if $\tilde{s} \prec \tilde{s}\{\tilde{x} \mapsto \epsilon\}$ for all $\tilde{x} \in \mathcal{V}_{\mathsf{H}}(\tilde{s})$.*

Example 2.32. *The hedges $\tilde{s} = (a, \tilde{x}, \tilde{X}(a))$ and $\tilde{s}' = (a, \tilde{x}, \tilde{y})$ are not in compressed form because $\tilde{s} \simeq \tilde{s}\{\tilde{x} \mapsto \epsilon\}$ and $\tilde{s}' \simeq \tilde{s}'\{\tilde{x} \mapsto \epsilon\}$, while $\tilde{q} = (\tilde{x}, a, \tilde{X}(a))$ and $\tilde{q}' = (\tilde{x}, \tilde{x}, a, \tilde{y})$ are in compressed form because $\tilde{q} \prec \tilde{q}\{\tilde{x} \mapsto \epsilon\}$ and $\tilde{q}' \prec \tilde{q}'\{\tilde{x} \mapsto \epsilon\}$ and $\tilde{q}' \prec \tilde{q}'\{\tilde{y} \mapsto \epsilon\}$.*

Theorem 2.25. *Let $\tilde{s}$ and $\tilde{q}$ be compressed forms of rigid hedges and let $\tilde{s} \simeq \tilde{q}$. There exists a renaming σ such that $\tilde{s}\sigma = \tilde{q}$ (and vice versa).*

Proof. From $\tilde{s} \simeq \tilde{q}$ and Lemma 2.4 follows that there is a substitution σ so that $\tilde{s}\sigma = \tilde{q}$ and $\mathcal{F}(\mathrm{Ran}(\sigma)) = \varnothing$. We can assume that $\mathrm{Dom}(\sigma) \subseteq \mathcal{V}(\tilde{s})$. Since $\tilde{s}$ and $\tilde{q}$ are compressed forms of rigid hedges, we can assume that for all hedges and contexts $\tilde{r} \in \mathrm{Ran}(\sigma)$ holds (see Definition 2.27):

1. *No context variable in $\tilde{r}$ applies to the empty hedge.*
2. *$\tilde{r}$ doesn't contain consecutive hedge variables.*
3. *$\tilde{r}$ doesn't contain vertical chains of variables.*
4. *$\tilde{r}$ doesn't contain context variables with a hedge variable as the first or the last argument (i.e., no subterms of the form $\tilde{X}(\tilde{x}, \dots)$ and $\tilde{X}(\dots, \tilde{x})$).*

First, we assume that $\mathrm{Ran}(\sigma) \nsubseteq \mathcal{V}_{\mathsf{H}} \cup \{\tilde{X}(\circ) \mid \tilde{X} \in \mathcal{V}_{\mathsf{C}}\}$ which will lead to a contradiction. Afterwards, it remains to show that σ is a bijection from $\mathrm{Dom}(\sigma)$ to $\mathrm{Ran}(\sigma)$. (See Definition 2.21.)

If $\mathrm{Ran}(\sigma) \nsubseteq \mathcal{V}_{\mathsf{H}} \cup \{\tilde{X}(\circ) \mid \tilde{X} \in \mathcal{V}_{\mathsf{C}}\}$, then there is some hedge or context $\tilde{r} \in \mathrm{Ran}(\sigma)$ such that either $\tilde{r} = \epsilon$ or $\tilde{r} = \circ$ or $|\tilde{r}| > 1$ or it is an application of the form $\tilde{Y}(\tilde{r}')$, where $\tilde{Y} \in \mathcal{V}_{\mathsf{C}}$ and $\tilde{r}' \in \mathcal{T}\backslash\{\epsilon, \circ\}$ (see item 1) so that $\mathcal{F}(\tilde{r}') = \varnothing$.

Case 1: Assume $\tilde{r} = \tilde{Y}(\tilde{r}')$ and $\tilde{r}' \in \mathcal{T}\backslash\{\epsilon, \circ\}$. Since, by item 1, no context variable in $\tilde{r}'$ applies to the empty hedge, every leaf in the tree representation is either a hedge variable or the hole. By item 4 we can exclude the case that a leaf is a hedge variable so that each leaf is the hole because $\mathcal{F}(\tilde{r}') = \varnothing$. By item 3, vertical chains are forbidden, hence $|\tilde{r}'| > 1$. This leads to more than one hole which is a contradiction to our grammar.

[†]Notice that the compressed form of a rigid generalization might violate the property of Definition 2.28, namely the existence of substitutions to obtain the input hedges from the generalization, where all the contexts are *singletons*.

Case 2: Assume $|\tilde{r}| > 1$. By the considerations of *Case 1* we know that no element of $\tilde{r}$ can be of the form $\tilde{Y}(\tilde{r}')$ where $\tilde{r}' \in \mathcal{T}\backslash\{\epsilon, \circ\}$. For this reason, each element from $\tilde{r}$ is either the hole $\circ$, or a hedge variable $\tilde{y}$, or a singleton context of the form $\tilde{Y}(\circ)$. Together with item 2, follows that $\tilde{r}$ is a context. The case where an element of $\tilde{r}$ is the hole $\circ$ can be ruled out for the following reason: We know that there is also a substitution σ' so that $\tilde{s} = \tilde{q}\sigma'$ and $\mathcal{F}(\mathrm{Ran}(\sigma')) = \varnothing$ because $\tilde{s} \simeq \tilde{q}$. That substitution σ' would have to introduce the eliminated context variable and this leads to a contradiction by our previous considerations. The only case that remains is that $\tilde{r}$ is of the form $(\tilde{y}, \tilde{Y}(\circ))$ or $(\tilde{Y}(\circ), \tilde{y})$ or $(\tilde{y}, \tilde{Y}(\circ), \tilde{z})$ but this contradicts to $\tilde{q}$ being in compressed form.

Case 3: Assume $\tilde{r} = \epsilon$ or $\tilde{r} = \circ$. Since $\tilde{s} \simeq \tilde{q}$, there is also a substitution σ' so that $\tilde{s} = \tilde{q}\sigma'$ and $\mathcal{F}(\mathrm{Ran}(\sigma')) = \varnothing$. That substitution σ' would have to introduce some extra variables leading to *Case 1* or *Case 2* and its contradiction.

It remains to show that σ is a bijection from $\mathrm{Dom}(\sigma)$ to $\mathrm{Ran}(\sigma)$. Let $\sigma = \{\tilde{x}_1 \mapsto \tilde{y}_1, \ldots, \tilde{x}_n \mapsto \tilde{y}_n, \tilde{X}_1 \mapsto \tilde{Y}_1(\circ), \ldots, \tilde{X}_m \mapsto \tilde{Y}_m(\circ)\}$. Since $\tilde{s} \simeq \tilde{q}$, there is also a substitution ϑ so that $\tilde{q}\vartheta = \tilde{s}$. Therefore, $\tilde{s}\sigma\vartheta = \tilde{s}$ and it follows that σ is a renaming, i.e., a bijection from $\mathrm{Dom}(\sigma)$ to $\mathrm{Ran}(\sigma)$. □

Corollary 2.26. *Let $\tilde{s}$ and $\tilde{q}$ be compressed forms of rigid hedges such that $\tilde{s} \simeq \tilde{q}$. Then $\|\tilde{s}\| = \|\tilde{q}\|$.*

From the proof of Theorem 2.25 we can extract an algorithm to compute a compressed form of a rigid hedge. To get a compressed form of a rigid hedge, we need to eliminate certain hedge variables that appear in subhedges of the forms $(\tilde{y}, \tilde{Y}(\tilde{r}))$ or $(\tilde{Y}(\tilde{r}), \tilde{y})$ or $(\tilde{y}, \tilde{Y}(\tilde{r}), \tilde{z})$ such that, after elimination, we get a hedge that is equigeneral to the original one. The latter case is subsumed by the other two cases. This elimination has to be done exhaustively so that the result is an equigeneral rigid hedge in compressed form. The following corollary describes the generate and test algorithm:

Corollary 2.27. *Let $\tilde{s} \in \mathcal{T}$ be a rigid hedge, then the compressed form of $\tilde{s}$ is the hedge $\tilde{s}\sigma$ and σ is defined by the mapping that is identity everywhere except for $\{\tilde{x} \mapsto \epsilon \mid \tilde{x} \in V_1\}$, where $V_1 \subseteq \mathcal{V}_{\mathsf{H}}(\tilde{s})$ are the following hedge variables:*

$\tilde{x} \in V_1$ if there is a substitution ϑ so that $\tilde{s} = \tilde{s}\{\tilde{x} \mapsto \epsilon\}\vartheta$ and ϑ is identity everywhere except for $\{\tilde{X} \mapsto (\tilde{x}, \tilde{X}(\circ)) \mid V_2\}\{\tilde{X} \mapsto (\tilde{X}(\circ), \tilde{x}) \mid V_3\}$ with $V_2 \subseteq \mathcal{V}_{\mathsf{C}}(\tilde{s})$ and $V_3 \subseteq \mathcal{V}_{\mathsf{C}}(\tilde{s})$.

In Corollary 2.27, searching for the substitution ϑ is exponential in the number of context variables that appear in the hedge $\tilde{s}$. By the powerset of all possibilities for ϑ, the complexity of the generate and test method is $O(2^{2n})$ where $n = |\mathcal{V}_{\mathsf{C}}(\tilde{s})|$. The search space can be pruned by analyzing the neighborhood of the appearances of context variables in $\tilde{s}$ but it remains nondeterministic due to the inherent complexity of higher-order terms.

Our goal is to minimize a set of rigid higher-order generalizations. Since, to the best of our knowledge, there is no higher-order hedge-matching algorithm and first-order hedge-matching algorithms are already NP-complete (see, e.g., [51, 52]), we aim at performing the minimization step by $\mathfrak{G}^{2\mathcal{V}}_{\mathfrak{a}}$ itself. To do this, we need to decide $\tilde{s} \stackrel{?}{\simeq} \tilde{q}$ for two rigid hedges $\tilde{s}$ and $\tilde{q}$. From Theorem 2.25 we know that for the compressed forms of $\tilde{s}$ and $\tilde{q}$ this problem reduces to finding a simple renaming. Therefore, we translate $\tilde{s}$ and $\tilde{q}$ into their compressed forms and search for such a renaming. We can do this similarly to subsection 2.1.4 in the following way.

Deciding the Existence of a Renaming. Again, we use the idea of sharing variables by representing unranked terms as directed, acyclic graphs (dags). Recall Definition 2.15 and Definition 2.16 from subsection 2.1.4:

Definition 2.15 (Unranked term dag)**.** *An* unranked term dag *is a directed, acyclic graph that is weakly connected and whose nodes are labeled with function symbols or variables. Function symbols may have incoming and outgoing edges while variables do not have outgoing edges. There is one function symbol that does not have incoming edges, called the* root *of the term dag. The outgoing edges from any node are ordered.*

Definition 2.16 (Variable sharing term dag)**.** *A* variable sharing term dag *is a term dag where all occurrences of the same variable share the same node of the graph.*

Let $\tilde{s}$ and $\tilde{q}$ be compressed forms of rigid hedges. To decide $\tilde{s} \stackrel{?}{\simeq} \tilde{q}$, we construct the terms $f(\tilde{s})$ and $f(\tilde{q})$, where $f \in \mathcal{F}$ is arbitrary. Then those terms can be represented as variable sharing dags where applications of context variables are in *curried form.* We use the unranked function symbol @ to write any application $\tilde{X}(\tilde{r})$ in its curried form $@(\tilde{X}(), \tilde{r})$, where $\tilde{X} \in \mathcal{V}_\mathsf{C}$ and $\tilde{r} \in \mathcal{T}$. To solve the decision problem, the dags are traversed synchronously like described in subsection 2.1.4 so that we have an algorithm that decides $\tilde{s} \stackrel{?}{\simeq} \tilde{q}$ in linear time. Figure 2.23 shows an unranked term and its translation into a variable sharing dag in curried form.

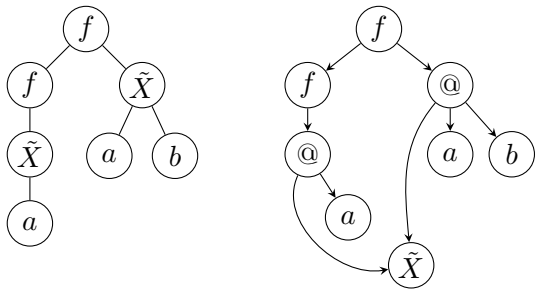

Figure 2.23: The term $f(f(\tilde{X}(a)), \tilde{X}(a, b))$ and its curried variable sharing dag.

Matching with $\mathfrak{G}_\mathfrak{a}^{2\mathcal{V}}$. Now we turn to discussing the matching problem. Our goal is to minimize a set of rigid generalizations of the same hedges but arbitrary alignments by using the algorithm $\mathfrak{G}_\mathfrak{a}^{2\mathcal{V}}$ itself. In fact, the following theorem says that $\mathfrak{G}_\mathfrak{a}^{2\mathcal{V}}$ decides $\tilde{s} \stackrel{?}{\preceq} \tilde{q}$ for arbitrary rigid hedges $\tilde{s}$ and $\tilde{q}$.

Theorem 2.28. *Let $\tilde{s}$ and $\tilde{q}$ be rigid hedges and let $\tilde{Y} \in \mathcal{V}_\mathsf{C}, \tilde{y} \in \mathcal{V}_\mathsf{H}$ be variables that occur neither in $\tilde{s}$ nor in $\tilde{q}$.*

- *Then $\tilde{s} \preceq \tilde{q}$ iff there is $\mathfrak{a}$ so that $\varnothing$; S; $\sigma = \mathfrak{G}_\mathfrak{a}^{2\mathcal{V}}(\tilde{Y}(\tilde{y})\colon \tilde{s} \triangleq \tilde{q})$ and $\tilde{Y}(\tilde{y})\sigma \simeq \tilde{s}$.*
- *Then $\tilde{q} \preceq \tilde{s}$ iff there is $\mathfrak{a}$ so that $\varnothing$; S; $\sigma = \mathfrak{G}_\mathfrak{a}^{2\mathcal{V}}(\tilde{Y}(\tilde{y})\colon \tilde{s} \triangleq \tilde{q})$ and $\tilde{Y}(\tilde{y})\sigma \simeq \tilde{q}$.*

Proof. Let $\tilde{s}$ and $\tilde{q}$ be rigid hedges and let $\tilde{Y} \in \mathcal{V}_\mathsf{C}, \tilde{y} \in \mathcal{V}_\mathsf{H}$ be variables that occur neither in $\tilde{s}$ nor in $\tilde{q}$.

We show $\tilde{s} \preceq \tilde{q}$ iff there is $\mathfrak{a}$ so that $\varnothing$; S; $\sigma = \mathfrak{G}_\mathfrak{a}^{2\mathcal{V}}(\tilde{Y}(\tilde{y})\colon \tilde{s} \triangleq \tilde{q})$ and $\tilde{Y}(\tilde{y})\sigma \simeq \tilde{s}$.

($\Rightarrow$) Assume $\tilde{s} \preceq \tilde{q}$. Since $\tilde{s}$ is a generalization of $\tilde{s}$ and $\tilde{q}$, by Theorem 2.12 there is an admissible alignments $\mathfrak{a}$ that corresponds to a supporting generalization of $\tilde{s}$ and $\tilde{q}$. By correctness and uniqueness of the final state $\varnothing;\, S;\, \sigma = \mathfrak{G}^{2\mathcal{V}}_{\mathfrak{a}}(\tilde{Y}(\tilde{y})\colon \tilde{s} \triangleq \tilde{q})$ follows that $\tilde{s} \simeq \tilde{Y}(\tilde{y})\sigma$. (Notice that $\tilde{Y}(\tilde{y})\sigma$ and $\tilde{s}$ are both rigid hedges.)

($\Leftarrow$) Assume there is $\mathfrak{a}$ so that $\varnothing;\, S;\, \sigma = \mathfrak{G}^{2\mathcal{V}}_{\mathfrak{a}}(\tilde{Y}(\tilde{y})\colon \tilde{s} \triangleq \tilde{q})$ and $\tilde{Y}(\tilde{y})\sigma \simeq \tilde{s}$. By coherence of $\mathfrak{G}^{2\mathcal{V}}_{\mathfrak{a}}$ follows that $\tilde{Y}(\tilde{y})\sigma\sigma_{\mathrm{R}}(S) = \tilde{q}$ By the assumption $\tilde{Y}(\tilde{y})\sigma \simeq \tilde{s}$ there is a renaming ϑ such that $\tilde{Y}(\tilde{y})\sigma = \tilde{s}\vartheta$. For that reasons $\tilde{s}\vartheta\sigma_{\mathrm{R}}(S) = \tilde{q}$. $\square$

By Theorem 2.28, to decide $\tilde{s} \overset{?}{\preceq} \tilde{q}$ for two rigid hedges $\tilde{s}$ and $\tilde{q}$ it suffices to search for an admissible alignment $\mathfrak{a}$ so that $\varnothing;\, S;\, \sigma = \mathfrak{G}^{2\mathcal{V}}_{\mathfrak{a}}(\tilde{Y}(\tilde{y})\colon \tilde{s} \triangleq \tilde{q})$ and $\tilde{Y}(\tilde{y})\sigma \simeq \tilde{s}$. By Lemma 2.4 the search space can be reduced drastically because $\mathfrak{a}$ must contain all the occurrences of all the function symbols from $\tilde{s}$. (Formally, from $|\mathfrak{a}| \neq |\{\mathrm{Pos}_f(\tilde{s}) : f \in \mathcal{F}\}|$ follows that $\tilde{Y}(\tilde{y})\sigma \prec \tilde{s}$, by Lemma 2.4.) Obviously, testing $\tilde{Y}(\tilde{y})\sigma \overset{?}{\simeq} \tilde{s}$ is done by translating $\tilde{Y}(\tilde{y})\sigma$ and $\tilde{s}$ into their, respective, compressed form. Then we perform the test on the curried variable sharing dags of the compressed forms. The most expensive operation is the translation into compressed form, therefore the search space pruning for possible alignments $\mathfrak{a}$ is of great practical value.

Example 2.33. *We demonstrate our results on the hedges* $\tilde{s} = (g(a, g(b), c))$ *and* $\tilde{q} = (a, f(a, c))$. *There are two admissible alignments of longest length, namely* $a\langle 1{\cdot}1, 1\rangle c\langle 1{\cdot}3, 2{\cdot}2\rangle$ *and* $a\langle 1{\cdot}1, 2{\cdot}1\rangle c\langle 1{\cdot}3, 2{\cdot}2\rangle$. *(It is the complete set of laas of* $\tilde{s}$ *and* $\tilde{q}$*.) We compute the two rigid generalizations of* $\tilde{s}$ *and* $\tilde{q}$ *with respect to those alignments by the algorithm* $\mathfrak{G}^{2\mathcal{V}}_{\mathfrak{a}}$*:*

$$\tilde{g}_1 = \tilde{X}_1(a, \tilde{x}_2, \tilde{X}_3(c)) \quad \textit{and} \quad \tilde{g}_2 = (\tilde{y}_1, \tilde{Y}_2(a, \tilde{y}_3, c)).$$

For $\tilde{g}_1$ *and* $\tilde{g}_2$*, there exists only one alignment* $\mathfrak{a}$ *that fulfills the condition* $|\mathfrak{a}| = |\{\mathrm{Pos}_f(\tilde{g}_1) : f \in \mathcal{F}\}|$*, namely* $\mathfrak{a} = a\langle 1{\cdot}1, 2{\cdot}1\rangle c\langle 1{\cdot}3{\cdot}1, 2{\cdot}3\rangle$*. Therefore, it is the sole candidate that can lead to a rigid generalization* $\tilde{g}'$ *of* $\tilde{g}_1$ *and* $\tilde{g}_2$ *so that* $\tilde{g}' \simeq \tilde{g}_1$*.* $\mathfrak{G}^{2\mathcal{V}}_{\mathfrak{a}}$ *computes the following generalization* $\tilde{g}'$ *for* $\tilde{g}_1$ *and* $\tilde{g}_2$ *w.r.t.* $\mathfrak{a}$*:*

$$\tilde{g}' = (\tilde{z}_1, \tilde{Z}_2(a, \tilde{z}_3, \tilde{Z}_4(c))).$$

The compressed form of $\tilde{g}_1$ *is* $\tilde{X}_1(a, \tilde{X}_3(c))$ *and the compressed form of* $\tilde{g}'$ *is* $\tilde{Z}_2(a, \tilde{Z}_4(c))$*. Therefore* $\tilde{g}_1 \preceq \tilde{g}_2$ *and the minimized set is* $\{\tilde{g}_2\}$*.*

2.3 Higher-Order Unranked Anti-Unification $4\mathcal{V}$

We consider generalizations for two input hedges and their admissible alignment. Our goal is to enhance the precision of the computed generalizations. For instance, in the clone detection example from Figure 2.22 different function names are generalized by a context variable. We loose the information that the similarities are located at the same level (under different heads) in both pieces of code. In order to capture such information in the generalization, we need variables that are more narrow than our context variables or our hedge variables. Here we introduce two additional types of variables: term variables to increase the horizontal precision and function variables to

increase the vertical precision of a generalization. The $4\mathcal{V}$ in the title of this section indicates the number of different variable types we consider.

Furthermore, we lift the restriction that a rigid generalization does not contain chains of variables, allowing function variables in vertical chain. Similarly, we allow consecutive term variables.

Example 2.34. *The hedge* $\tilde{g} = (F(a, F(a, \tilde{x}, a)), \tilde{x}, \tilde{X}(g(x, x)))$ *is a generalization of two hedges* $\tilde{s} = (f(a, f(a, c, a)), c, f(g(b, b)))$ *and* $\tilde{q} = (g(a, g(a, a)), g(d, d))$. *Dotted and dashed nodes indicate differences, while the solid ones form the admissible alignment.* $\tilde{s} = \tilde{g}\{F \mapsto f(\circ), \tilde{x} \mapsto c, \tilde{X} \mapsto f(\circ), x \mapsto b\}$ *and* $\tilde{q} = \tilde{g}\{F \mapsto g(\circ), \tilde{x} \mapsto \epsilon, \tilde{X} \mapsto \circ, x \mapsto d\}$.

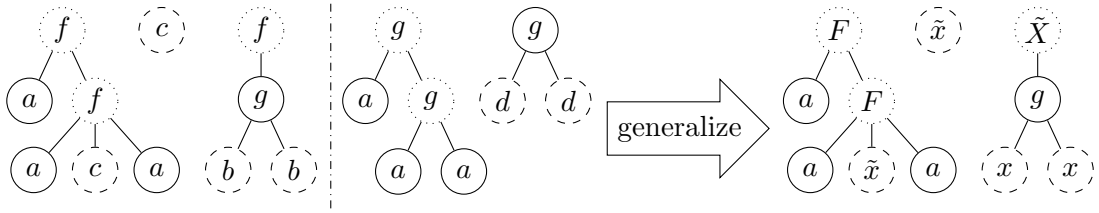

Figure 2.24: The hedges $\tilde{s}$ and $\tilde{q}$ and their higher-order lgg $\tilde{g}$ from Example 2.34.

In Example 2.34 we use all four kinds of variables in the generalization $\tilde{g}$ of the hedges $\tilde{s}$ and $\tilde{q}$. The function variable F in $\tilde{g}$ indicates that all the a's occur at the same level in both $\tilde{s}$ and $\tilde{q}$. The symbol g occurs at different levels because there is a context variable above. Furthermore, it illustrates the consecutive term variables (x, x). Notice that $\mathfrak{G}_{\mathfrak{a}}^{2\mathcal{V}}$ computes for the same hedges and the same admissible alignment the generalization $(\tilde{X}(a, \tilde{X}(a, \tilde{x}, a)), \tilde{x}, \tilde{Y}(g(\tilde{y})))$.

2.3.1 Higher-Order Unranked Terms and Hedges $4\mathcal{V}$ (Preliminaries)

Many definitions from subsection 2.1.1 and subsection 2.2.1 are still valid for the richer term language that we consider here. We overload only those definitions that do not trivially generalize to the following terms, hedges, and contexts:

Definition 2.35 (Terms, hedges, contexts)**.** *Given pairwise disjoint countable sets of unranked function symbols* $\mathcal{F}$ *(symbols without fixed arity), term variables* $\mathcal{V}_{\mathsf{T}}$*, function variables* $\mathcal{V}_{\mathsf{F}}$*, hedge variables* $\mathcal{V}_{\mathsf{H}}$*, context variables* $\mathcal{V}_{\mathsf{C}}$*, and a special symbol* $\circ$ *(the hole), we define* terms, hedges, *and* contexts *by the following grammar:*

$$
\begin{array}{lll}
t & ::= & x \mid f(\tilde{s}) \mid F(\tilde{s}) \qquad \textit{(term)} \\
s & ::= & t \mid \tilde{x} \mid \tilde{X}(\tilde{s}) \qquad \textit{(hedge element)} \\
\tilde{s} & ::= & s_1, \ldots, s_n \qquad \textit{(hedge)} \\
\mathring{c} & ::= & f(\tilde{s}_1, \circ, \tilde{s}_2) \mid F(\tilde{s}_1, \circ, \tilde{s}_2) \qquad \textit{(bounded context)} \\
\tilde{c} & ::= & \tilde{s}_1, \circ, \tilde{s}_2 \mid \tilde{s}_1, f(\tilde{c}), \tilde{s}_2 \mid \tilde{s}_1, F(\tilde{c}), \tilde{s}_2 \mid \tilde{s}_1, \tilde{X}(\tilde{c}), \tilde{s}_2 \qquad \textit{(context)}
\end{array}
$$

where $f \in \mathcal{F}$, $x \in \mathcal{V}_{\mathsf{T}}$, $F \in \mathcal{V}_{\mathsf{F}}$, $\tilde{x} \in \mathcal{V}_{\mathsf{H}}$, $\tilde{X} \in \mathcal{V}_{\mathsf{C}}$, *and* $n \geqslant 0$.

This definition can be seen as an extension of the terms, hedges, and contexts from subsection 2.2.1. The grammar considers two additional variable types: term variables

and function variables. The set of all variables $\mathcal{V}_\mathsf{T} \cup \mathcal{V}_\mathsf{F} \cup \mathcal{V}_\mathsf{H} \cup \mathcal{V}_\mathsf{C}$ is written as $\mathcal{V}$. Term variables are first-order variables and they can be instantiated by a term. Intuitively, a function variable stand for an arbitrary function that is applied to a hedge of arguments where some of the arguments may be fixed and others can be provided at instantiation time. Function variables can be instantiated by a *bounded context*, that is a term over $\mathcal{F} \cup \{\circ\}$ and $\mathcal{V}$, where the hole occurs exactly once at level 1. We denote bounded contexts by $\mathring{c}, \mathring{d}$, function variables by F, G, H, and term variables by x, y, z. We use $\mathcal{H}, \mathcal{G}$ for a function symbol or a function variable. Application of a bounded context $\mathring{c}$ to a hedge $\tilde{s}$, denoted by $\mathring{c}[\tilde{s}]$, amounts to replacing the hole in $\mathring{c}$ by the hedge $\tilde{s}$. It is the same as in the general case for arbitrary contexts and, like in the general case, a bounded context can also be applied to another context.

Variables from $\mathcal{V}_\mathsf{H} \cup \mathcal{V}_\mathsf{T}$ are called *first-order* variables and the others are called *higher-order* variables. The set of hedges and contexts constructed over $\mathcal{F} \cup \{\circ\}$ and $\mathcal{V}$ is denoted by $\mathcal{T}(\mathcal{F}, \mathcal{V})$, or simply by $\mathcal{T}$ if the concrete instances of $\mathcal{F}$ and $\mathcal{V}$ do not matter. Similarly to $\mathcal{V}_\mathsf{H}(\cdot)$ and $\mathcal{V}_\mathsf{C}(\cdot)$, we denote by $\mathcal{V}_\mathsf{T}(\cdot)$ and $\mathcal{V}_\mathsf{F}(\cdot)$, respectively, the set of all the occurring term variables and the set of all the occurring function variables in an element from $\mathcal{T}$ or in any subset of $\mathcal{T}$. The notions of the set of positions $\mathrm{Pos}(\tilde{s})$, the top symbols $\mathrm{Top}(\tilde{s})$, the length $|\tilde{s}|$, the size $\|\tilde{s}\|$, and the subterm $\tilde{s}|_I$ of a hedge or context $\tilde{s}$ can be extended naturally to take into account the new variable types.

Definition 2.36 (Substitution)**.** *A* substitution *is a mapping $\sigma : \mathcal{V} \to \mathcal{T}(\mathcal{F}, \mathcal{V})$ from hedge variables to hedges, from context variables to contexts, from term variables to terms, and from function variables to bounded contexts, which is identity almost everywhere. When substituting a higher-order variable by a context, the context will be applied to the argument hedge of that higher-order variable.*

Context application is defined by induction on the structure of contexts as in subsection 2.2.1. Any substitution σ can be extended to a mapping $\hat{\sigma} : \mathcal{T} \to \mathcal{T}$ that can be applied to hedges and contexts. Similarly to subsection 2.2.1, the application is defined by induction on the structure of hedges and contexts:

$$\tilde{s}\hat{\sigma} ::= \begin{cases} f(\tilde{q}\hat{\sigma}) & \text{if } \tilde{s} = f(\tilde{q}), \\ \sigma(\tilde{x}) & \text{if } \tilde{s} = \tilde{x}, \\ \sigma(x) & \text{if } \tilde{s} = x, \\ \sigma(\tilde{X})[\tilde{q}\hat{\sigma}] & \text{if } \tilde{s} = \tilde{X}(\tilde{q}), \\ \sigma(F)[\tilde{q}\hat{\sigma}] & \text{if } \tilde{s} = F(\tilde{q}), \\ \circ & \text{if } \tilde{s} = \circ, \\ s_1\hat{\sigma}, \ldots, s_n\hat{\sigma} & \text{if } |\tilde{s}| \neq 1 \text{ and } \tilde{s} = (s_1, \ldots, s_n). \end{cases}$$

For a substitution σ the *domain* is the set of variables

$$\mathrm{Dom}(\sigma) ::= \{\xi \in \mathcal{V}_\mathsf{H} \cup \mathcal{V}_\mathsf{T} \mid \sigma(\xi) \neq \xi\} \cup \{\Xi \in \mathcal{V}_\mathsf{C} \cup \mathcal{V}_\mathsf{F} \mid \sigma(\Xi) \neq \Xi(\circ)\}$$

and the *range* is the set of hedges and contexts

$$\mathrm{Ran}(\sigma) ::= \{\sigma(\xi) \mid \xi \in \mathrm{Dom}(\sigma)\} \cup \{\sigma(\Xi) \mid \Xi \in \mathrm{Dom}(\sigma)\}.$$

To simplify the notation, we do not distinguish between a substitution σ and its extension $\hat{\sigma}$.

Example 2.35. *Let* $\sigma = \{\tilde{x} \mapsto \epsilon,\ x \mapsto f(\tilde{X}(a)),\ F \mapsto g(\circ),\ \tilde{X} \mapsto F(F(\circ))\}$ *be a substitution, then* $(F(\tilde{x}), x, f(F(a)), \tilde{X}(f(a), a))\sigma = (g, f(\tilde{X}(a)), f(g(a)), F(F(f(a), a)))$.

The definition of a renaming is extended in the following way:

Definition 2.37 (Renaming)**.** *A substitution is called* renaming *if* $Ran(\sigma) \subseteq \mathcal{V}_{\mathsf{H}} \cup \mathcal{V}_{\mathsf{T}} \cup \{\tilde{X}(\circ) \mid \tilde{X} \in \mathcal{V}_{\mathsf{C}}\} \cup \{F(\circ) \mid F \in \mathcal{V}_{\mathsf{F}}\}$ *and* $|Dom(\sigma)| = |Ran(\sigma)|$.

A renaming is a bijection form the domain to the range of a substitution. The notions of instantiation and generalization are the same as in Definition 2.7 and Definition 2.8. They are defined by the existence of substitutions to compare hedges with respect to the instantiation quasi ordering. Lemma 2.1 and Corollary 2.2 are also directly applicable to this richer term language.

Theorem 2.29. *Higher-order unranked anti-unification is finitary: For any hedges* $\tilde{s}$ *and* $\tilde{q}$ *there exists their minimal complete set of generalizations. This set is finite and unique modulo* $\simeq$.

Proof. Like in the proof of Theorem 2.11, we use context variables as substitute for hedge variables. It suffices to consider only context variables, term variables and function variables.

Since $\tilde{X}()$ acts as generalization for every pair of hedges, we assume there is a nonempty set of hedges G for two arbitrary hedges $\tilde{s}$ and $\tilde{q}$ so that for all $\tilde{g} \in G$ holds that $\tilde{g} \preceq \tilde{s}$ and $\tilde{g} \preceq \tilde{q}$.

For all $\tilde{g} \in G$, the number of function symbols of $\tilde{g}$ is bound by the number of function symbols in $\tilde{s}$. Furthermore, from $\tilde{g}\sigma = \tilde{s}$ we know that a variable is either eliminated, or it stands for a certain part of the hedge $\tilde{s}$. There are only finitely many variables in $\tilde{g}$ that stand for some part of $\tilde{s}$ and all the others are eliminated from $\tilde{g}$ when applying σ. In fact the number is bound by $\|\tilde{s}\|$. Term variables and function variables cannot be eliminated, therefore the number of all occurrences of term variables and function variables together in $\tilde{g}$ is bounded by $\|\tilde{s}\|$. The same holds for $\tilde{q}$.

Putting this together, there are only finitely many variables in $\tilde{g}$ that indicate some differences at $\tilde{s}$ and $\tilde{q}$. Eliminating all those superfluous context variables we obtain $\tilde{g}' = \tilde{g}\{\tilde{X}_1 \mapsto \circ, \tilde{X}_2 \mapsto \circ, \dots\}$, such that the number of variables in $\tilde{g}'$ is bound by $\|\tilde{s}\| + \|\tilde{q}\|$ and $\tilde{g}'$ is a generalization of $\tilde{s}$ and $\tilde{q}$. Since $\tilde{g} \preceq \tilde{g}'$ it can be removed from G if $\tilde{g} \neq \tilde{g}'$. As the size of an lgg is bounded by $\|\tilde{s}\| + \|\tilde{q}\|$, also the number of possible lggs modulo $\simeq$ is bound. This leads to G being finite. □

2.3.2 Higher-Order Unranked Anti-Unification Algorithm $\mathfrak{G}_{\mathfrak{a}}^{4\mathcal{V}}$

Recall, our goal is to enhance the precision of the computed generalization. Therefore, we refine some of the concepts that have been introduced in subsection 2.2.3 and enrich the algorithm $\mathfrak{G}_{\mathfrak{a}}^{2\mathcal{V}}$ with two new rules for two new kinds of variables. Following the naming of the algorithm $\mathfrak{G}_{\mathfrak{a}}^{2\mathcal{V}}$, we denote its extended version by $\mathfrak{G}_{\mathfrak{a}}^{4\mathcal{V}}$. The first step towards our goal is to relax the notion of a rigid hedge:

Definition 2.38 (Rigid hedge)**.** *A hedge* $\tilde{s}$ *is* rigid *if the following conditions hold:*

1. *No higher-order variable in* $\tilde{s}$ *applies to the empty hedge.*

2. *If $\tilde{s}$ contains consecutive first-order variables, then both of them are term variables.*
3. *If $\tilde{s}$ contains a vertical chain of variables, then both of them are function variables.*
4. *$\tilde{s}$ doesn't contain higher-order variables with a first-order variable as the first or the last argument.*

Notice that consecutive variables are allowed if one of them is a higher-order variable or if both of them are term variables. This definition of rigid hedges relaxes Definition 2.27 because there we do not have consecutive first-order variables and all vertical chains of variables are forbidden. In order to use term variables and function variables, we also need to slightly widen the notion of an AUE from the previous section:

Definition 2.39 (Anti-unification equation)**.** *An* anti-unification equation*, AUE in short, is a triple of the form $\xi\colon \tilde{s} \triangleq \tilde{q}; \Xi\colon \tilde{c} \triangleq \tilde{d}; \mathfrak{a}$, where*

- *ξ is a* first-order *variable and $\tilde{s}, \tilde{q}$ are hedges,*
- *Ξ is a* higher-order *variable and $\tilde{c}, \tilde{d}$ are contexts,*
- *$\mathfrak{a}$ is an admissible alignment of $\tilde{s}$ and $\tilde{q}$.*

The definitions of a rigid generalization and a rigid lgg remain unchanged. We instantiate those definitions by the new notion of a rigid hedge. An example of a rigid lgg using Definition 2.38 can be found in the introduction of this section (Example 2.34).

Definition 2.40 (Rigid higher-order anti-unification algorithm)**.** *The algorithm $\mathfrak{G}_{\mathfrak{a}}^{4\mathcal{V}}$ is an extension of $\mathfrak{G}_{\mathfrak{a}}^{2\mathcal{V}}$. It works on tuples $P; S; \sigma$, called the states, like its predecessor. $\mathfrak{G}_{\mathfrak{a}}^{4\mathcal{V}}$ consists of ten transformation rules that operate on states. One of the eight rules that are inherited from $\mathfrak{G}_{\mathfrak{a}}^{2\mathcal{V}}$, namely the rule* Mer-S*, needs a marginal refinement. The two additional rules, called* narrowing*, are presented below.*

Similarly to the definition of an AUE, we need to redefine the rule Mer-S so that it works for both kinds of first-order variables and for both kinds of higher-order variables. We only need to override the declaration of the generalization variables.

Mer-S: **Merge Store**

$P; \{\xi_1\colon \tilde{s} \triangleq \tilde{q}; \Xi_1\colon \tilde{c} \triangleq \tilde{d},\ \xi_2\colon \tilde{s} \triangleq \tilde{q}; \Xi_2\colon \tilde{c} \triangleq \tilde{d}\} \cup S; \sigma \Longrightarrow$
$P; \{\xi_1\colon \tilde{s} \triangleq \tilde{q}; \Xi_1\colon \tilde{c} \triangleq \tilde{d}\} \cup S; \sigma\{\xi_2 \mapsto \xi_1, \Xi_2 \mapsto \Xi_1(\circ)\}$,

where ξ_1, ξ_2 are first-order variables of the same type and Ξ_1, Ξ_2 are higher-order variables of the same type.

The algorithm $\mathfrak{G}_{\mathfrak{a}}^{4\mathcal{V}}$ comprises two additional transformation rules. Both of them work on the store. The first one transforms hedge variables into term variables and the second one transforms context variables into function variables. The computational complexity introduced by adding those new rules is relatively low. In fact, we will show that $\mathfrak{G}_{\mathfrak{a}}^{4\mathcal{V}}$ still needs quadratic time and linear space on the size of the input.

Nar-FO: **Narrowing First-Order Variables**

$P; \{\tilde{x}\colon (t_1, \ldots, t_n) \triangleq (s_1, \ldots, s_n); \tilde{X}\colon \circ \triangleq \circ\} \uplus S; \sigma \Longrightarrow$
$P; \{y_i\colon t_i \triangleq s_i; \tilde{Y}_i\colon \circ \triangleq \circ \mid 1 \leqslant i \leqslant n\} \cup S; \sigma\{\tilde{x} \mapsto (y_1, \ldots, y_n)\}$,

where $n \geqslant 1$ and y_i's, $\tilde{Y}_i$'s are fresh. t_i's and s_i's are terms.

Nar-HO: **Narrowing Higher-Order Variables**

$P; \{\tilde{x}\colon \epsilon \triangleq \epsilon;\ \tilde{X}\colon \mathcal{H}_1(\tilde{s}_1, \ldots \mathcal{H}_n(\tilde{s}_n, \circ, \tilde{s}'_n)\ldots, \tilde{s}'_1) \triangleq \mathcal{G}_1(\tilde{q}_1, \ldots \mathcal{G}_n(\tilde{q}_n, \circ, \tilde{q}'_n)\ldots, \tilde{q}'_1)\} \cup S; \sigma \Longrightarrow$
$P; \{\tilde{y}_i\colon \epsilon \triangleq \epsilon;\ F_i\colon \mathcal{H}_i(\tilde{s}_i, \circ, \tilde{s}'_i) \triangleq \mathcal{G}_i(\tilde{q}_i, \circ, \tilde{q}'_i) \mid 1 \leqslant i \leqslant n\} \cup S; \sigma\{\tilde{X} \mapsto F_1(\ldots F_n(\circ)\ldots)\}$,

where $n \geqslant 1$ and $\tilde{y}_i$'s, F_i's are fresh. $\mathcal{H}_i$'s and $\mathcal{G}_i$'s are from $\mathcal{F} \cup \mathcal{V}_\mathsf{F}$.

The algorithm $\mathfrak{G}^{4\mathcal{V}}_{\mathfrak{a}}$ works in the same manner as $\mathfrak{G}^{2\mathcal{V}}_{\mathfrak{a}}$. We will prove that $\mathfrak{G}^{4\mathcal{V}}_{\mathfrak{a}}$ is coherent and computes a unique (modulo $\simeq$) final state, denoted by $\mathfrak{G}^{4\mathcal{V}}_{\mathfrak{a}}(\tilde{X}(\tilde{x})\colon \tilde{s} \triangleq \tilde{q})$ for two hedges $\tilde{s}$ and $\tilde{q}$ with respect to an admissible alignment $\mathfrak{a}$ and fresh variables $\tilde{x}$ and $\tilde{X}$. The rigid lgg that corresponds to a final state $\varnothing; S; \sigma = \mathfrak{G}^{4\mathcal{V}}_{\mathfrak{a}}(\tilde{X}(\tilde{x})\colon \tilde{s} \triangleq \tilde{q})$ can be obtained by $\tilde{X}(\tilde{x})\sigma$. Definition 2.32 can be easily generalized so that the substitutions $\sigma_{\mathrm{L}}(S)$ and $\sigma_{\mathrm{R}}(S)$ cover all the differences from the store. In fact, it can be redefined in the same manner as the Mer-S rule, overriding the declaration of the generalization variables $\tilde{x}, \tilde{X}$.

Before we turn to discussing the properties of the algorithm $\mathfrak{G}^{4\mathcal{V}}_{\mathfrak{a}}$ we illustrate its usage on some examples.

2.3.3 Illustration of the Algorithm $\mathfrak{G}^{4\mathcal{V}}_{\mathfrak{a}}$

Example 2.36. *We illustrate the transformation steps performed by $\mathfrak{G}^{4\mathcal{V}}_{\mathfrak{a}}$ on the hedges $\tilde{s} = (f(f(a)), f(g(b,b)))$ and $\tilde{q} = (g(g(a)), g(d,d))$ and their admissible alignment $\mathfrak{a} = a\langle 1{\cdot}1{\cdot}1, 1{\cdot}1{\cdot}1\rangle g\langle 2{\cdot}1, 2\rangle$. In the substitution, we only keep the mappings for the two generalization variables $\tilde{x}$ and $\tilde{X}$ of the initial AUE.*

$\{\tilde{x}\colon (f(f(a)), f(g(b,b))) \triangleq (g(g(a)), g(d,d));\ \tilde{X}\colon \circ \triangleq \circ;\ a\langle 1{\cdot}1{\cdot}1, 1{\cdot}1{\cdot}1\rangle g\langle 2{\cdot}1, 2\rangle\};\ \varnothing;\ Id$

$\Longrightarrow^{\text{Spl-H}}_{\text{Clr-S}}$ $\{\tilde{y}_1\colon f(f(a)) \triangleq g(g(a));\ \tilde{Y}_1\colon \circ \triangleq \circ;\ a\langle 1{\cdot}1{\cdot}1, 1{\cdot}1{\cdot}1\rangle,$
$\tilde{z}_1\colon f(g(b,b)) \triangleq g(d,d);\ \tilde{Z}_1\colon \circ \triangleq \circ;\ g\langle 1{\cdot}1, 1\rangle\};\ \varnothing;\ \{\tilde{x} \mapsto (\tilde{Y}_1(\tilde{y}_1), \tilde{Z}_1(\tilde{z}_1)), \tilde{X} \mapsto \circ\}$

$\Longrightarrow^{\text{Abs-L}}_{\text{Abs-L}}$ $\{\tilde{y}_1\colon a \triangleq g(g(a));\ \tilde{Y}_1\colon f(f(\circ)) \triangleq \circ;\ a\langle 1, 1{\cdot}1{\cdot}1\rangle,$
$\tilde{z}_1\colon f(g(b,b)) \triangleq g(d,d);\ \tilde{Z}_1\colon \circ \triangleq \circ;\ g\langle 1{\cdot}1, 1\rangle\};\ \varnothing;\ \{\tilde{x} \mapsto (\tilde{Y}_1(\tilde{y}_1), \tilde{Z}_1(\tilde{z}_1)), \tilde{X} \mapsto \circ\}$

$\Longrightarrow^{\text{Abs-R}}_{\text{Abs-R}}$ $\{\tilde{y}_1\colon a \triangleq a;\ \tilde{Y}_1\colon f(f(\circ)) \triangleq g(g(\circ));\ a\langle 1, 1\rangle,$
$\tilde{z}_1\colon f(g(b,b)) \triangleq g(d,d);\ \tilde{Z}_1\colon \circ \triangleq \circ;\ g\langle 1{\cdot}1, 1\rangle\};\ \varnothing;\ \{\tilde{x} \mapsto (\tilde{Y}_1(\tilde{y}_1), \tilde{Z}_1(\tilde{z}_1)), \tilde{X} \mapsto \circ\}$

$\Longrightarrow^{\text{App-A}}_{\text{Clr-S}}$ $\{\tilde{z}_1\colon f(g(b,b)) \triangleq g(d,d);\ \tilde{Z}_1\colon \circ \triangleq \circ;\ g\langle 1{\cdot}1, 1\rangle\};$
$\{\tilde{y}_1\colon \epsilon \triangleq \epsilon;\ \tilde{Y}_1\colon f(f(\circ)) \triangleq g(g(\circ))\};\ \{\tilde{x} \mapsto (\tilde{Y}_1(a), \tilde{Z}_1(\tilde{z}_1)), \tilde{X} \mapsto \circ\}$

$\Longrightarrow_{\text{Abs-L}}$ $\{\tilde{z}_1\colon g(b,b) \triangleq g(d,d);\ \tilde{Z}_1\colon f(\circ) \triangleq \circ;\ g\langle 1, 1\rangle\};$
$\{\tilde{y}_1\colon \epsilon \triangleq \epsilon;\ \tilde{Y}_1\colon f(f(\circ)) \triangleq g(g(\circ))\};\ \{\tilde{x} \mapsto (\tilde{Y}_1(a), \tilde{Z}_1(\tilde{z}_1)), \tilde{X} \mapsto \circ\}$

$\Longrightarrow^{\text{App-A}}_{\text{Sol-H}}$ $\varnothing;\ \{\tilde{y}_1\colon \epsilon \triangleq \epsilon;\ \tilde{Y}_1\colon f(f(\circ)) \triangleq g(g(\circ)),$
$\tilde{z}_1\colon \epsilon \triangleq \epsilon;\ \tilde{Z}_1\colon f(\circ) \triangleq \circ,\ \tilde{z}_2\colon (b,b) \triangleq (d,d);\ \tilde{Z}_2\colon \circ \triangleq \circ\};\ \{\tilde{x} \mapsto (\tilde{Y}_1(a), \tilde{Z}_1(g(\tilde{z}_2))), \tilde{X} \mapsto \circ\}$

$\Longrightarrow_{\text{Nar-HO}}$ $\varnothing;\ \{\tilde{y}_2\colon \epsilon \triangleq \epsilon;\ F_1\colon f(\circ) \triangleq g(\circ),\ \tilde{y}_3\colon \epsilon \triangleq \epsilon;\ F_2\colon f(\circ) \triangleq g(\circ),$
$\tilde{z}_1\colon \epsilon \triangleq \epsilon;\ \tilde{Z}_1\colon f(\circ) \triangleq \circ,\ \tilde{z}_2\colon (b,b) \triangleq (d,d);\ \tilde{Z}_2\colon \circ \triangleq \circ\};$
$\{\tilde{x} \mapsto (F_1(F_2(a)), \tilde{Z}_1(g(\tilde{z}_2))), \tilde{X} \mapsto \circ\}$

$\Longrightarrow_{\text{Mer-S}}$ $\varnothing;\ \{\tilde{y}_2\colon \epsilon \triangleq \epsilon;\ F_1\colon f(\circ) \triangleq g(\circ),\ \tilde{z}_1\colon \epsilon \triangleq \epsilon;\ \tilde{Z}_1\colon f(\circ) \triangleq \circ,\ \tilde{z}_2\colon (b,b) \triangleq (d,d);\ \tilde{Z}_2\colon \circ \triangleq \circ\};$
$\{\tilde{x} \mapsto (F_1(F_1(a)), \tilde{Z}_1(g(\tilde{z}_2))), \tilde{X} \mapsto \circ\}$

$\Longrightarrow_{\text{Nar-FO}}$ $\varnothing;\ \{\tilde{y}_2\colon \epsilon \triangleq \epsilon;\ F_1\colon f(\circ) \triangleq g(\circ),\ \tilde{z}_1\colon \epsilon \triangleq \epsilon;\ \tilde{Z}_1\colon f(\circ) \triangleq \circ,\ x_1\colon b \triangleq d;\ \tilde{Z}_3\colon \circ \triangleq \circ,$

$$
\begin{array}{ll}
 & x_2{:}\, b \triangleq d;\ \tilde{Z}_4{:}\, \circ \triangleq \circ\};\ \{\tilde{x} \mapsto (F_1(F_1(a)), \tilde{Z}_1(g(x_1, x_2))), \tilde{X} \mapsto \circ\} \\
\Longrightarrow_{\text{Mer-S}} & \varnothing;\ \{\tilde{y}_2{:}\, \epsilon \triangleq \epsilon;\ F_1{:}\, f(\circ) \triangleq g(\circ),\ \tilde{z}_1{:}\, \epsilon \triangleq \epsilon;\ \tilde{Z}_1{:}\, f(\circ) \triangleq \circ,\ x_1{:}\, b \triangleq d;\ \tilde{Z}_3{:}\, \circ \triangleq \circ\};\\
 & \{\tilde{x} \mapsto (F_1(F_1(a)), \tilde{Z}_1(g(x_1, x_1))), \tilde{X} \mapsto \circ\}.
\end{array}
$$

$\tilde{X}(\tilde{x})\sigma = (F_1(F_1(a)), \tilde{Z}_1(g(x_1, x_1)))$ *generalizes* $\tilde{s}$ *and* $\tilde{q}$ *with respect to* $\mathfrak{a}$. *From the store* S *we can read* $\sigma_{_L}(S) = \{F_1 \mapsto f(\circ),\ x_1 \mapsto b,\ \tilde{Z}_1 \mapsto f(\circ), \ldots\}$ *and* $\sigma_{_R}(S) = \{F_1 \mapsto g(\circ),\ x_1 \mapsto d,\ \tilde{Z}_1 \mapsto \circ, \ldots\}$. *Then we have* $\tilde{X}(\tilde{x})\sigma\sigma_{_L}(S) = \tilde{s}$ *and* $\tilde{X}(\tilde{x})\sigma\sigma_{_R}(S) = \tilde{q}$.

The algorithm $\mathfrak{G}^{2\mathcal{V}}_{\mathfrak{a}}$ from section 2.2 computes for the hedges from Example 2.36 with respect to the same alignment the generalization $(\tilde{X}(a), \tilde{Y}(g(\tilde{x})))$. It is strictly more general than the one $\mathfrak{G}^{4\mathcal{V}}_{\mathfrak{a}}$ computes.

Example 2.37. *Again we use the code sample from Roy et al. [75] to illustrate the algorithm* $\mathfrak{G}^{4\mathcal{V}}_{\mathfrak{a}}$ *on the application area of software clone detection. Recall the original code and its representation as an unranked term:*

```
1 void sumProd(int n) {
2   float sum = 0.0;
3   float prod = 1.0;
4   for(int i=1; i<=n; i++) {
5     sum = sum + i;
6     prod = prod * i;
7     foo(sum,prod); }}
```

$$
\begin{array}{l}
sumProd(input(type(int), n),\\
\quad returnType(void),\\
\quad =(type(float), sum, 0.0),\\
\quad =(type(float), prod, 1.0),\\
\quad for(=(type(int), i, 1), \leqslant(i, n), {+}{+}(i),\\
\quad\quad =(sum, +(sum, i)),\\
\quad\quad =(prod, *(prod, i)),\\
\quad\quad foo(sum, prod)))
\end{array}
$$

Figure 2.25: The original program used to illustrate clone detection by anti-unification.

The two considered clones are illustrated in Figure 2.26. In the first clone, the types of `sum` and `prod` have been changed from float to int. The second clone has already been discussed in subsection 2.2.5 (see Figure 2.22). It differs from the original program by applying the function `bar` instead of `foo` to the arguments `sum` and `prod` at line 7.

```
1 void sumProd(int n) {
2   int sum = 0;
3   int prod = 1;
4   for(int i=1; i<=n; i++) {
5     sum = sum + i;
6     prod = prod * i;
7     foo(sum, prod); }}
```

```
void sumProd(int n) {
  float sum = 0.0;
  float prod = 1.0;
  for(int i=1; i<=n; i++) {
    sum = sum + i;
    prod = prod * i;
    bar(sum,prod); }}
```

Figure 2.26: Two clones of the program from Figure 2.25.

We skip the translation from the code clones into hedges and directly show the results of running $\mathfrak{G}^{4\mathcal{V}}_{\mathfrak{a}}$ on the respective clone and the original program in Figure 2.27. We consider the longest admissible alignment which is unique in both cases.

$$
\begin{array}{l}
sumProd(input(type(int), n), \\
\quad returnType(void), \\
\quad =(type(\mathbf{x}), sum, \mathbf{y}), \\
\quad =(type(\mathbf{x}), prod, \mathbf{z}), \\
\quad for(=(type(int), i, 1), \leqslant(i, n), ++(i), \\
\qquad =(sum, +(sum, i)), \\
\qquad =(prod, *(prod, i)), \\
\qquad foo(sum, prod)))
\end{array}
\qquad
\begin{array}{l}
sumProd(input(type(int), n), \\
\quad returnType(void), \\
\quad =(type(float), sum, 0.0), \\
\quad =(type(float), prod, 1.0), \\
\quad for(=(type(int), i, 1), \leqslant(i, n), ++(i), \\
\qquad =(sum, +(sum, i)), \\
\qquad =(prod, *(prod, i)), \\
\qquad \mathbf{F}(sum, prod)))
\end{array}
$$

Figure 2.27: Result of running $\mathfrak{G}_{\mathfrak{a}}^{4\mathcal{V}}$ to detect the clones from Figure 2.26.

In the first case term variables are the only variables used in the generalization. That means it can be solved by basic first-order techniques. Nevertheless, without term variables our algorithm would compute results that are "worse" than those computed by syntactic first-order generalization. The second result contains a function variable F which stands for the application of either `bar` or `foo`. Remember, the algorithm $\mathfrak{G}_{\mathfrak{a}}^{2\mathcal{V}}$ introduced a context variable in order to abstract different function applications at two pieces of software code. $\mathfrak{G}_{\mathfrak{a}}^{4\mathcal{V}}$ uses function variables to preserve the information that the similarities are located at the same level of the input syntax trees.

2.3.4 Properties of the Algorithm $\mathfrak{G}_{\mathfrak{a}}^{4\mathcal{V}}$

Here we show that $\mathfrak{G}_{\mathfrak{a}}^{4\mathcal{V}}$ terminates and indeed computes a rigid lgg that is unique modulo $\simeq$ for two hedges and their admissible alignment. We also show that $\mathfrak{G}_{\mathfrak{a}}^{4\mathcal{V}}$ is coherent and its computational complexity is quadratic in time and linear in space on the size of the input. Before stating the main theorems, we formulate a couple of lemmas. We need them to prove the other properties of the algorithm.

Lemma 2.30. *Let P; S; $\vartheta \Longrightarrow_{\mathsf{R}_1} P_1$; S_1; $\vartheta\sigma_1 \Longrightarrow_{\mathsf{R}_2} P_2$; S_2; $\vartheta\sigma_1\sigma_2$ be a sequence of transformations where $\mathsf{R}_1 \in \{\mathsf{Nar\text{-}FO}, \mathsf{Nar\text{-}HO}\}$ and $\mathsf{R}_2 \in \mathfrak{G}_{\mathfrak{a}}^{4\mathcal{V}} \backslash \{\mathsf{Nar\text{-}FO}, \mathsf{Nar\text{-}HO}, \mathsf{Mer\text{-}S}\}$. Then there exists a transformation sequence P; S; $\vartheta \Longrightarrow_{\mathsf{R}_2} P_1'$; S_1'; $\vartheta\sigma_2 \Longrightarrow_{\mathsf{R}_1} P_2'$; S_2'; $\vartheta\sigma_2\sigma_1$ so that $P_2 = P_2', S_2 = S_2'$, and $\sigma_1\sigma_2 = \sigma_2\sigma_1$.*

Proof. The rule R_2 cannot operate on an AUE introduced by R_1 because the rules $\mathfrak{G}_{\mathfrak{a}}^{4\mathcal{V}} \backslash \{\mathsf{Nar\text{-}FO}, \mathsf{Nar\text{-}HO}, \mathsf{Mer\text{-}S}\}$ work only with generalization variables of the "$2\mathcal{V}$" types. Therefore we can assume that $\Longrightarrow_{\mathsf{R}_1}$ and $\Longrightarrow_{\mathsf{R}_2}$ are the exactly same transformation step in both transformation sequences, i.e., the same rule, the same AUE, and the same fresh variables. It is easy to see that $P_2 = P_2'$ because R_1 does not affect the first component of the state. As R_1 and R_2 operate on distinct AUEs and all the AUEs have pairwise disjoint generalization variables which are distinct from all the variable occurrences in hedges and contexts of $P \cup S$ we get that $\mathrm{Dom}(\sigma_1) \cap \mathrm{Dom}(\sigma_2) = \varnothing$, $\mathrm{Dom}(\sigma_1) \cap \mathcal{V}(\mathrm{Ran}(\sigma_2)) = \varnothing$, and $\mathrm{Dom}(\sigma_2) \cap \mathcal{V}(\mathrm{Ran}(\sigma_1)) = \varnothing$. Putting all this together we get that $P_2 = P_2'$, $S_2 = S_2'$, and $\sigma_1\sigma_2 = \sigma_2\sigma_1$. □

Lemma 2.31. *Let P; S; $\vartheta \Longrightarrow_{\mathsf{Mer\text{-}S}} P_1$; S_1; $\vartheta\sigma_1 \Longrightarrow_{\mathsf{R}} P_2$; S_2; $\vartheta\sigma_1\sigma_2$ be a sequence of transformations where $\mathsf{R} \in \mathfrak{G}_{\mathfrak{a}}^{4\mathcal{V}}$ is arbitrary but fixed. Then there exists a transformation sequence P; S; $\vartheta \Longrightarrow_{\mathsf{R}}^{*} P_1'$; S_1'; $\vartheta\sigma_1' \Longrightarrow_{\mathsf{Mer\text{-}S}}^{*} P_2'$; S_2'; $\vartheta\sigma_1'\sigma_2'$ so that $P_2 = P_2', S_2 = S_2'$, and $\tilde{X}(\tilde{x})\sigma_1\sigma_2 = \tilde{X}(\tilde{x})\sigma_1'\sigma_2'$ for all generalization variables in $P \cup S$.*

Proof. If Mer-S and R operate on different AUEs then we can choose the transformation P; S; $\vartheta \Longrightarrow_{\mathsf{R}} P_1'$; S_1'; $\vartheta\sigma_1' \Longrightarrow_{\mathsf{Mer\text{-}S}} P_2'$; S_2'; $\vartheta\sigma_1'\sigma_2'$ to get $P_2 = P_2', S_2 = S_2'$, and $\sigma_1\sigma_2 = \sigma_1'\sigma_2'$, by the same considerations as in the proof of Lemma 2.30.

Let us assume that R operates on the merged AUE from the first transformation step. Since R operates on the merged AUE, it operates on the store, hence $\mathsf{R} \in \{\mathsf{Res\text{-}C}, \mathsf{Clr\text{-}S}, \mathsf{Mer\text{-}S}, \mathsf{Nar\text{-}FO}, \mathsf{Nar\text{-}HO}\}$. If $\mathsf{R} \in \{\mathsf{Res\text{-}C}, \mathsf{Clr\text{-}S}\}$, then we can use confluence of $\mathfrak{G}_{\mathfrak{a}}^{2\mathcal{V}}$ because Res-C and Clr-S work only with generalization variables of the "$2\mathcal{V}$" types. Therefore, it suffices to analyze the cases $\mathsf{R} \in \{\mathsf{Mer\text{-}S}, \mathsf{Nar\text{-}FO}, \mathsf{Nar\text{-}HO}\}$. It follows that $P_2 = P_2'$ because Mer-S, Nar-FO, and Nar-HO do not touch the first component of the state. The case $\mathsf{R} = \mathsf{Mer\text{-}S}$ is trivial because we can choose the exactly same transformation sequence.

R=Nar-FO. Mer-S operates on two AUEs in S that consist of the same hedges and contexts, say $\{\tilde{x}_1 \colon \tilde{s} \triangleq \tilde{q};\ \tilde{X}_1 \colon \tilde{c} \triangleq \tilde{d},\ \tilde{x}_2 \colon \tilde{s} \triangleq \tilde{q};\ \tilde{X}_2 \colon \tilde{c} \triangleq \tilde{d}\} \subseteq S$. W.l.o.g. $S_1 = S\backslash\{\tilde{x}_2 \colon \tilde{s} \triangleq \tilde{q};\ \tilde{X}_2 \colon \tilde{c} \triangleq \tilde{d}\}$ and $\sigma_1 = \{\tilde{x}_2 \mapsto \tilde{x}_1, \tilde{X}_2 \mapsto \tilde{X}_1(\circ)\}$. By assumption, the rule Nar-FO operates on $\tilde{x}_1 \colon \tilde{s} \triangleq \tilde{q};\ \tilde{X}_1 \colon \tilde{c} \triangleq \tilde{d}$. Therefore, it is of the form $\tilde{x}_1 \colon (t_1, \ldots, t_n) \triangleq (s_1, \ldots, s_n);\ \tilde{X}_1 \colon \circ \triangleq \circ$ and $\tilde{X}_1()\vartheta = \circ = \tilde{X}_2()\vartheta$ and $\tilde{X}_1, \tilde{X}_2$ do not occur in the generalization for they are eliminated by Sol-H. Let S_2'' be $S_1\backslash\{\tilde{x}_1 \colon \tilde{s} \triangleq \tilde{q};\ \tilde{X}_1 \colon \tilde{c} \triangleq \tilde{d}\}$, then $S_2 = S_2'' \cup \{x_i \colon t_i \triangleq s_i;\ \tilde{Y}_i \colon \circ \triangleq \circ \mid 1 \leqslant i \leqslant n\}$ and $\sigma_2 = \{\tilde{x}_1 \mapsto (x_1, \ldots, x_n)\}$ where x_i's and $\tilde{Y}_i$'s are fresh.

Let S_0'' be $S\backslash\{\tilde{x}_1 \colon \tilde{s} \triangleq \tilde{q};\ \tilde{X}_1 \colon \tilde{c} \triangleq \tilde{d}\}$. We choose as first part of the transformation sequence P; S; $\vartheta \Longrightarrow_{\mathsf{Nar\text{-}FO}} P_0'$; S_0'; $\vartheta\sigma_0' \Longrightarrow_{\mathsf{Nar\text{-}FO}} P_1'$; S_1'; $\vartheta\sigma_1'$ so that $S_0' = S_0'' \cup \{x_i \colon t_i \triangleq s_i;\ \tilde{Y}_i \colon \circ \triangleq \circ \mid 1 \leqslant i \leqslant n\}$ and $\sigma_2 = \{\tilde{x}_1 \mapsto (x_1, \ldots, x_n)\}$ where x_i's and $\tilde{Y}_i$'s are the same fresh variables as above. Furthermore, $S_1' = S_0'\backslash\{\tilde{x}_2 \colon \tilde{s} \triangleq \tilde{q};\ \tilde{X}_2 \colon \tilde{c} \triangleq \tilde{d}\} \cup \{y_i \colon t_i \triangleq s_i;\ \tilde{Z}_i \colon \circ \triangleq \circ \mid 1 \leqslant i \leqslant n\}$ and $\sigma_2 = \{\tilde{x}_2 \mapsto (y_1, \ldots, y_n)\}$ where y_i's and $\tilde{Z}_i$'s are fresh. Now we apply Mer-S n times for merging $x_i \colon t_i \triangleq s_i;\ \tilde{Y}_i \colon \circ$ with $y_i \colon t_i \triangleq s_i;\ \tilde{Z}_i \colon \circ$ and we keep the variables $x_i, \tilde{Y}_i$ in the computed generalization. The mappings of the form $y_i \mapsto x_i$ remain in σ_2' but they are negligible because they have been introduced as being fresh and afterwards they have been eliminated, hence they can be seen as temporary. Putting our considerations together we get that $S_2 = S_2'$, and $\tilde{X}(\tilde{x})\sigma_1\sigma_2 = \tilde{X}(\tilde{x})\sigma_1'\sigma_2'$ for all generalization variables in $P \cup S$.

R=Nar-HO. The considerations are exactly the same as for Nar-FO. □

Theorem 2.32 (Termination)**.** *The algorithm $\mathfrak{G}_{\mathfrak{a}}^{4\mathcal{V}}$ terminates on any input.*

Proof. The termination proof for $\mathfrak{G}_{\mathfrak{a}}^{2\mathcal{V}}$ still works for $\mathfrak{G}_{\mathfrak{a}}^{4\mathcal{V}}$. (See Theorem 2.20.) □

Theorem 2.33 (Soundness)**.** *Let P be a set of AUEs of the form $\{\tilde{x} \colon \tilde{s} \triangleq \tilde{q};\ \tilde{X} \colon \circ \triangleq \circ;\ \mathfrak{a}\}$. Every exhaustive rule application in $\mathfrak{G}_{\mathfrak{a}}^{4\mathcal{V}}$ yields a derivation P; $\varnothing$; $Id \Longrightarrow^{+} \varnothing$; S; σ where $\tilde{g} = \tilde{X}(\tilde{x})\sigma$ is a rigid generalization of $\tilde{s}$ and $\tilde{q}$ with respect to $\mathfrak{a}$ and the store S records all the differences such that $\tilde{g}\sigma_{L}(S) = \tilde{s}$ and $\tilde{g}\sigma_{R}(S) = \tilde{q}$.*

Proof. A rigid generalization in the sense of Definition 2.28 is also a rigid hedge in the sense of Definition 2.38. By Lemma 2.30 and Lemma 2.31, we can assume that all the rules $\mathfrak{G}_{\mathfrak{a}}^{2\mathcal{V}}\backslash\{\mathsf{Mer\text{-}S}\}$ have been exhaustively applied and by soundness of $\mathfrak{G}_{\mathfrak{a}}^{2\mathcal{V}}$ we know that $\tilde{g}\sigma_{\mathrm{L}}(S) = \tilde{s}$ and $\tilde{g}\sigma_{\mathrm{R}}(S) = \tilde{q}$ holds and $\tilde{g}$ is a rigid generalization. Let P; $\varnothing$; $Id \Longrightarrow^{+}_{\mathfrak{G}_{\mathfrak{a}}^{2\mathcal{V}}\backslash\{\mathsf{Mer\text{-}S}\}} \varnothing$; S_0; ϑ_0 be such an exhaustive partial transformation and let $\varnothing$; S_0; $\vartheta_0 \Longrightarrow^{*}_{\mathsf{Nar\text{-}FO,Nar\text{-}HO}} \varnothing$; S_n; $\vartheta_0\vartheta_1^n$ be some transformation applications of the rules Nar-FO and Nar-HO. The empty transformation $n = 0$ is our base case. Assuming that for $n-1$ all the properties hold, we show by induction that $\varnothing$; S_{n-1}; $\vartheta_0\vartheta_1^{n-1} \Longrightarrow_{\mathsf{Nar\text{-}FO,Nar\text{-}HO}} \varnothing$; S_n; $\vartheta_0\vartheta_1^{n-1}\vartheta_n$ still maintains all the properties.

The rule Nar-FO transforms a hedge variable into consecutive term variables. Let $S' = S_{n-1}\backslash\{\tilde{x}\colon (t_1,\ldots,t_n) \triangleq (s_1,\ldots,s_n); \tilde{X}\colon \circ \triangleq \circ\}$ and $\vartheta_n = \{\tilde{x} \mapsto (x_1,\ldots,x_n)\}$ and $S_n = S' \cup \{x_i\colon t_i \triangleq s_i; \tilde{Y}_i\colon \circ \triangleq \circ \mid 1 \leqslant i \leqslant n\}$. From coherence and soundness, namely $\tilde{X}(\tilde{x})\vartheta_0\vartheta_1^{n-1}\sigma_{\mathrm{L}}(S_{n-1}) = \tilde{s}$, follows that $\tilde{X}(\tilde{x})\vartheta_0\vartheta_1^{n-1}\vartheta_n\sigma_{\mathrm{L}}(S_n) = \tilde{s}$ because $x_i\sigma_{\mathrm{L}}(S_n) = t_i$ for all $1 \leqslant i \leqslant n$. Furthermore, since $\tilde{X}(\tilde{x})\vartheta_0\vartheta_1^{n-1}$ is a rigid generalization, also $\tilde{X}(\tilde{x})\vartheta_0\vartheta_1^{n-1}\vartheta_n$ is rigid because consecutive term variables are allowed. The reasoning for Nar-HO is equivalent and to show that Mer-S maintains all the properties we can refer to the soundness proof of $\mathfrak{G}_{\mathfrak{a}}^{2\mathcal{V}}$ since all the considerations there also hold for the new types of variables. □

Theorem 2.34 (Completeness)**.** *Let $\tilde{g}$ be a rigid generalization of $\tilde{s}$ and $\tilde{q}$ with respect to $\mathfrak{a}$. Then there exists a derivation $\{\tilde{x}\colon \tilde{s} \triangleq \tilde{q}; \tilde{X}\colon \circ \triangleq \circ; \mathfrak{a}\}$; $\varnothing$; $Id \Longrightarrow^{+} \varnothing$; S; σ obtained by $\mathfrak{G}_{\mathfrak{a}}^{4\mathcal{V}}$ such that $\tilde{g} \preceq \tilde{X}(\tilde{x})\sigma$.*

Proof. We proceed in the same manner as in the proof of the completeness theorem (Theorem 2.22) of $\mathfrak{G}_{\mathfrak{a}}^{2\mathcal{V}}$, applying rules in reverse order to obtain from the generalization $\tilde{g}$ the initial state $\{\tilde{x}\colon \tilde{s} \triangleq \tilde{q}; \tilde{X}\colon \circ \triangleq \circ; \mathfrak{a}\}$; $\varnothing$; Id. Assuming that there are substitutions so that $\tilde{g}\vartheta = \tilde{s}$ and $\tilde{g}\vartheta' = \tilde{q}$, we initialize for the reverse transformation the store by those substitutions so that $\tilde{g}\sigma_{\mathrm{L}}(S) = \tilde{s}$ and $\tilde{g}\sigma_{\mathrm{R}}(S) = \tilde{q}$. The strategy from the proof of Theorem 2.22 is extended in the following way: First Mer-S is applied in reverse $\varnothing$; S'; $\sigma' \;{}^{*}_{\mathsf{Mer\text{-}S}}\!\!\Longleftarrow \varnothing$; S; σ until all the variable occurrences are disjoint in the range of σ'. Afterwards we transform all the (maximal) subsequences of term variables into fresh hedge variables and all the (maximal) application-sequences of function variables are transformed into fresh context variables by reverse application of Nar-FO and Nar-HO, respectively $\varnothing$; S''; $\sigma'' \;{}^{*}_{\mathsf{Nar\text{-}FO,Nar\text{-}HO}}\!\!\Longleftarrow \varnothing$; S'; σ'. Afterwards we use the strategy from the proof of Theorem 2.22. □

Theorem 2.35 (Uniqueness modulo $\simeq$)**.** *Let $\mathfrak{a}$ be an admissible alignment of $\tilde{s}$ and $\tilde{q}$. If $\{\tilde{x}_1\colon \tilde{s} \triangleq \tilde{q}; \tilde{X}_1\colon \circ \triangleq \circ; \mathfrak{a}\}$; $\varnothing$; $Id \Longrightarrow^{+} \varnothing$; S_1; σ_1 and $\{\tilde{x}_2\colon \tilde{s} \triangleq \tilde{q}; \tilde{X}_2\colon \circ \triangleq \circ; \mathfrak{a}\}$; $\varnothing$; $Id \Longrightarrow^{+} \varnothing$; S_2; σ_2 are two exhaustive derivations in $\mathfrak{G}_{\mathfrak{a}}^{4\mathcal{V}}$, then $\tilde{X}_1(\tilde{x}_1)\sigma_1 \simeq \tilde{X}_2(\tilde{x}_2)\sigma_2$.*

Proof. By Lemma 2.30 and Lemma 2.31 we can rearrange both derivations so that they correspond to an exhaustive transformation in $\mathfrak{G}_{\mathfrak{a}}^{2\mathcal{V}}$. Doing so we get partial derivations $\{\tilde{x}_1\colon \tilde{s} \triangleq \tilde{q}; \tilde{X}_1\colon \circ \triangleq \circ; \mathfrak{a}\}$; $\varnothing$; $Id \Longrightarrow^{+} \varnothing$; S_1'; σ_1' and $\{\tilde{x}_2\colon \tilde{s} \triangleq \tilde{q}; \tilde{X}_2\colon \circ \triangleq \circ; \mathfrak{a}\}$; $\varnothing$; $Id \Longrightarrow^{+} \varnothing$; S_2'; σ_2'. From the uniqueness theorem of $\mathfrak{G}_{\mathfrak{a}}^{2\mathcal{V}}$ we know that $\tilde{X}_1(\tilde{x}_1)\sigma_1' \simeq \tilde{X}_2(\tilde{x}_2)\sigma_2'$. Since Nar-FO operates on AUEs with empty contexts and Nar-HO works on AUEs with empty hedges, they operate on disjoint AUEs. Together with disjointness of the generalization variables we get confluence of Nar-FO and

Nar-HO applications. Mer-S transformations are postponed until the end. From confluence of Nar-FO and Nar-HO applications we get $\varnothing; S_1'; \sigma_1' \Longrightarrow^*_{\text{Nar-FO,Nar-HO}} \varnothing; S_1''; \sigma_1''$ and $\varnothing; S_2'; \sigma_2' \Longrightarrow^*_{\text{Nar-FO,Nar-HO}} \varnothing; S_2''; \sigma_2''$ with $\tilde{X}_1(\tilde{x}_1)\sigma_1'' \simeq \tilde{X}_2(\tilde{x}_2)\sigma_2''$. The postponed Mer-S transformations are confluent too (see uniqueness proof of $\mathfrak{G}_{\mathfrak{a}}^{2\mathcal{V}}$). □

2.3.5 Complexity Analysis of $\mathfrak{G}_{\mathfrak{a}}^{4\mathcal{V}}$

The complexity analysis shows that the two additional rules do neither increase the upper bound of the runtime complexity nor the upper bound of the required space that we gave in Theorem 2.24.

Theorem 2.36 (Complexity)**.** *The anti-unification algorithm $\mathfrak{G}_{\mathfrak{a}}^{4\mathcal{V}}$ has $O(n^2)$ time complexity and $O(n)$ space complexity, where n is the number of symbols in the input.*

Proof. Like in the uniqueness proof, we perform an exhaustive $\mathfrak{G}_{\mathfrak{a}}^{2\mathcal{V}}$ transformation that can be done in $O(n^2)$ time and $O(n)$ space, where n is the number of symbols in the input. Mer-S is postponed until the end and the reasoning is the same as in the case of $\mathfrak{G}_{\mathfrak{a}}^{2\mathcal{V}}$. Therefore, it suffices to show that exhaustive Nar-FO and Nar-HO transformations can be done in $O(n^2)$ time using $O(n)$ space. From the previous complexity analysis of $\mathfrak{G}_{\mathfrak{a}}^{2\mathcal{V}}$ we know that the size of the store is bounded by $O(n)$. Since the number of term variables and function variables together which are introduced by Nar-FO and Nar-HO is also bounded by $O(n)$ and each term variable and each function variable in the generalization stands for at least one symbol at both input hedges, the space complexity of $O(n)$ remains intact. For the same reason there are only linearly many applications of Nar-FO and Nar-HO together. Since the store is bounded by $O(n)$, all the applicability tests of Nar-FO and Nar-HO together can be done in linear time too. There are at most linearly many substitution compositions that can be done in linear time, hence the additional rules do neither increase the time complexity nor the space complexity of the anti-unification process. □

2.3.6 Minimization by Anti-Unification using $\mathfrak{G}_{\mathfrak{a}}^{4\mathcal{V}}$

To use the minimization technique from $\mathfrak{G}_{\mathfrak{a}}^{2\mathcal{V}}$ by the extended algorithm $\mathfrak{G}_{\mathfrak{a}}^{4\mathcal{V}}$ we need to prove Theorem 2.25 again for the redefined notion of a rigid hedge. We recall the theorem from subsection 2.2.9 and prove that it still holds:

Theorem 2.25. *Let $\tilde{s}$ and $\tilde{q}$ be compressed forms of rigid hedges and let $\tilde{s} \simeq \tilde{q}$. There exists a renaming σ such that $\tilde{s}\sigma = \tilde{q}$ (and vice versa).*

Proof. From $\tilde{s} \simeq \tilde{q}$ and Lemma 2.4 follows that there is a substitution σ so that $\tilde{s}\sigma = \tilde{q}$ and $\mathcal{F}(\mathrm{Ran}(\sigma)) = \varnothing$. We can assume that $\mathrm{Dom}(\sigma) \subseteq \mathcal{V}(\tilde{s})$. Since $\tilde{s}$ and $\tilde{q}$ are compressed forms of rigid hedges, we can assume that for all hedges and contexts $\tilde{r} \in \mathrm{Ran}(\sigma)$ holds (see Definition 2.38):

1. *No higher-order variable in $\tilde{r}$ applies to the empty hedge.*
2. *If $\tilde{r}$ contains consecutive first-order variables, then both of them are term variables.*
3. *If $\tilde{r}$ contains a vertical chain of variables, then both of them are function variables.*
4. *$\tilde{r}$ doesn't contain higher-order variables with a first-order variable as the first or the last argument.*

First, we assume that $\mathrm{Ran}(\sigma) \nsubseteq \mathcal{V}_\mathsf{H} \cup \mathcal{V}_\mathsf{T} \cup \{\tilde{X}(\circ) \mid \tilde{X} \in \mathcal{V}_\mathsf{C}\} \cup \{F(\circ) \mid F \in \mathcal{V}_\mathsf{F}\}$ which will lead to a contradiction. Afterwards, it remains to show that σ is a bijection from $\mathrm{Dom}(\sigma)$ to $\mathrm{Ran}(\sigma)$. (See Definition 2.37.)

If $\mathrm{Ran}(\sigma) \nsubseteq \mathcal{V}_\mathsf{H} \cup \mathcal{V}_\mathsf{T} \cup \{\tilde{X}(\circ) \mid \tilde{X} \in \mathcal{V}_\mathsf{C}\} \cup \{F(\circ) \mid F \in \mathcal{V}_\mathsf{F}\}$, then there is some hedge or context $\tilde{r} \in \mathrm{Ran}(\sigma)$ such that either $\tilde{r} = \epsilon$ or $\tilde{r} = \circ$ or $|\tilde{r}| > 1$ or it is an application of the form $\tilde{Y}(\tilde{r}')$, where $\tilde{Y} \in \mathcal{V}_\mathsf{C} \cup \mathcal{V}_\mathsf{F}$ and $\tilde{r}' \in \mathcal{T}\backslash\{\epsilon, \circ\}$ (see item 1) so that $\mathcal{F}(\tilde{r}') = \varnothing$.

Case 1: Assume $\tilde{r} = \tilde{Y}(\tilde{r}')$ and $\tilde{r}' \in \mathcal{T}\backslash\{\epsilon, \circ\}$. Since, by item 1, no higher-order variable in $\tilde{r}'$ applies to the empty hedge, every leaf in the tree representation is either a first-order variable or the hole. By item 4 we can exclude the case that a leaf is a first-order variable so that each leaf is the hole because $\mathcal{F}(\tilde{r}') = \varnothing$. Since there is only one hole in a context, $\mathrm{Top}(\tilde{r}')$ and $\tilde{Y}$ are two variables in vertical chains in the hedge $\tilde{r}$. (Notice that $|\tilde{r}'| > 1$ would lead to more than one hole.) By item 3, it follows that both of them are function variables. Therefore, the definition of σ contains a mapping of the form $\tilde{X} \mapsto F(G(\tilde{c}))$ for some $F, G \in \mathcal{V}_\mathsf{F}, \tilde{X} \in \mathcal{V}_\mathsf{C}$ and some context $\tilde{c}$. By assumption $\tilde{s}\sigma = \tilde{q}$ and with the mapping $\tilde{X} \mapsto F(G(\tilde{c}))$ follows that $\tilde{s} \prec \tilde{q}$. This is a contradiction to $\tilde{s} \simeq \tilde{q}$.

Case 2: Assume $|\tilde{r}| > 1$. By the considerations of *Case 1* we know that no element of $\tilde{r}$ can be of the form $\tilde{Y}(\tilde{r}')$ where $\tilde{r}' \in \mathcal{T}\backslash\{\epsilon, \circ\}$. For this reason and because $\mathcal{F}(\tilde{r}') = \varnothing$, each element from $\tilde{r}$ is either the hole, or a first-order variable, or a singleton context of the form $\tilde{Y}(\circ)$ where $\tilde{Y} \in \mathcal{V}_\mathsf{C} \cup \mathcal{V}_\mathsf{F}$.

Case 2.1: Assume that $\tilde{r}$ contains an element that is a term variable. Because $|\tilde{r}| > 1$ and by assumption $\tilde{s}\sigma = \tilde{q}$ it follows that σ contains a mapping of the form $\tilde{X} \mapsto \tilde{r}$ or $\tilde{x} \mapsto \tilde{r}$. Since at least one element of $\tilde{r}$ is a term variable it follows that $\tilde{s} \prec \tilde{q}$. This is a contradiction to $\tilde{s} \simeq \tilde{q}$.

Case 2.2: Assume that $\tilde{r}$ contains an element that is the hole. It follows that σ contains a mapping of the form $\tilde{X} \mapsto (\tilde{r}' \circ \tilde{r}'')$. Because $\tilde{r}'$ and $\tilde{r}''$ cannot be contexts, it follows that both of them are (possibly empty) sequences of first-order variables. We know that there is also a substitution σ' so that $\tilde{s} = \tilde{q}\sigma'$ and $\mathcal{F}(\mathrm{Ran}(\sigma')) = \varnothing$ because $\tilde{s} \simeq \tilde{q}$. That substitution σ' would have to introduce the eliminated context variable and this leads to a contradiction by our previous considerations.

Case 2.3: Assume that $\tilde{r}$ contains no element that is a term variable or the hole. The only possibility that remains is that $\tilde{r}$ is of the form $(\tilde{y}, \tilde{Y}(\circ))$ or $(\tilde{Y}(\circ), \tilde{y})$ or $(\tilde{y}, \tilde{Y}(\circ), \tilde{z})$ where $\tilde{Y} \in \mathcal{V}_\mathsf{C} \cup \mathcal{V}_\mathsf{F}$.

Case 2.3.1: Assume that $\tilde{Y} \in \mathcal{V}_\mathsf{F}$, say $\tilde{Y} = F$. Because $|\tilde{r}| > 1$ it follows that σ contains a mapping of the form $\tilde{X} \mapsto (\tilde{r}', F(\circ), \tilde{r}'')$ where $\tilde{X} \in \mathcal{V}_\mathsf{C}$. It follows that $\tilde{s} \prec \tilde{q}$ because $\tilde{s}\sigma = \tilde{q}$. This is a contradiction to $\tilde{s} \simeq \tilde{q}$.

Case 2.3.2: Assume that $\tilde{Y} \in \mathcal{V}_\mathsf{C}$. This contradicts to $\tilde{q}$ being in compressed form.

Case 3: Assume $\tilde{r} = \epsilon$ or $\tilde{r} = \circ$. Since $\tilde{s} \simeq \tilde{q}$, there is also a substitution σ' so that $\tilde{s} = \tilde{q}\sigma'$ and $\mathcal{F}(\mathrm{Ran}(\sigma')) = \varnothing$. That substitution σ' would have to introduce some extra variables leading to *Case 1* or *Case 2* and its contradiction.

It remains to show that σ is a bijection from $\mathrm{Dom}(\sigma)$ to $\mathrm{Ran}(\sigma)$. Since $\tilde{s} \simeq \tilde{q}$, there is a substitution ϑ so that $\tilde{q}\vartheta = \tilde{s}$. Therefore, $\tilde{s}\sigma\vartheta = \tilde{s}$. It follows that σ is a renaming. □

Notice that a term variable can only be instantiated by a term and *not* by a hedge variable. Furthermore, function variables can only be instantiated by a bonded context and *not* by a hedge variable. For that reason only eliminations of hedge variables that

appear next to a context variable need to be considered when computing the compressed form of a rigid hedge, like done in Corollary 2.27. Computing a compressed form of a rigid hedge can be done in exactly the same way as in subsection 2.2.9 because term variables and function variables cannot be instantiated by a hedge variable.

It follows that for two rigid hedges $\tilde{s}$ and $\tilde{q}$ we can decide $\tilde{s} \stackrel{?}{\simeq} \tilde{q}$ again by translating $\tilde{s}$ and $\tilde{q}$ into their compressed forms and then we search for a renaming on the compressed forms. To search for a renaming, we can again use the dag representation where higher-order variables are in curried form.

To solve the matching problem for two rigid hedges by $\mathfrak{G}_{\mathfrak{a}}^{4\mathcal{V}}$, one can proceed in exactly the same way as described in subsection 2.2.9 for the simpler case without term variables and function variables. Even the proof of Theorem 2.28 remains sound without any modification. Therefore we can just state the following corollary:

Corollary 2.37. *Let $\tilde{s}$ and $\tilde{q}$ be rigid hedges and let $\tilde{Y} \in \mathcal{V}_{\mathsf{C}}, \tilde{y} \in \mathcal{V}_{\mathsf{H}}$ be variables that occur neither in $\tilde{s}$ nor in $\tilde{q}$.*

- *Then $\tilde{s} \preceq \tilde{q}$ iff there is $\mathfrak{a}$ so that $\varnothing; S; \sigma = \mathfrak{G}_{\mathfrak{a}}^{4\mathcal{V}}(\tilde{Y}(\tilde{y}) \colon \tilde{s} \triangleq \tilde{q})$ and $\tilde{Y}(\tilde{y})\sigma \simeq \tilde{s}$.*
- *Then $\tilde{q} \preceq \tilde{s}$ iff there is $\mathfrak{a}$ so that $\varnothing; S; \sigma = \mathfrak{G}_{\mathfrak{a}}^{4\mathcal{V}}(\tilde{Y}(\tilde{y}) \colon \tilde{s} \triangleq \tilde{q})$ and $\tilde{Y}(\tilde{y})\sigma \simeq \tilde{q}$.*

Notice that the considerations about pruning the search space from subsection 2.2.9 also apply here. To decide $\tilde{s} \stackrel{?}{\preceq} \tilde{q}$ by the algorithm $\mathfrak{G}_{\mathfrak{a}}^{4\mathcal{V}}$, one only needs to consider those alignments $\mathfrak{a}$ that fulfill the condition $|\mathfrak{a}| = |\{\mathrm{Pos}_f(\tilde{s}) : f \in \mathcal{F}\}|$. Informally, all appearances of function symbols in $\tilde{s}$ should also occur in $\mathfrak{a}$. Otherwise $\mathfrak{a}$ is not a possible candidate.

Example 2.38. *We demonstrate our results on the hedges $\tilde{s} = (f(f(a)), f(a))$ and $\tilde{q} = (a, g(g(a)), a)$. There are three admissible alignments of longest length, namely $a\langle 1{\cdot}1{\cdot}1, 1\rangle a\langle 2{\cdot}1, 2{\cdot}1{\cdot}1\rangle$ and $a\langle 1{\cdot}1{\cdot}1, 1\rangle a\langle 2{\cdot}1, 3\rangle$ and $a\langle 1{\cdot}1{\cdot}1, 2{\cdot}1{\cdot}1\rangle a\langle 2{\cdot}1, 3\rangle$. (It is the complete set of laas of $\tilde{s}$ and $\tilde{q}$.) We compute the three rigid generalizations of $\tilde{s}$ and $\tilde{q}$ with respect to those alignments by the algorithm $\mathfrak{G}_{\mathfrak{a}}^{4\mathcal{V}}$:*

$$\tilde{g}_1 = (\tilde{X}_1(a), \tilde{X}_2(a), \tilde{x}_3) \quad \text{and} \quad \tilde{g}_2 = (\tilde{Y}_1(a), \tilde{y}_2, \tilde{Y}_3(a)) \quad \text{and} \quad \tilde{g}_3 = (\tilde{z}_1, F_2(F_3(a)), \tilde{Z}_4(a)).$$

For $\tilde{g}_1$ and $\tilde{g}_2$, there exists only one alignment $\mathfrak{a}$ that fulfills the condition $|\mathfrak{a}| = |\{\mathrm{Pos}_f(\tilde{g}_1) : f \in \mathcal{F}\}|$, namely $\mathfrak{a} = a\langle 1{\cdot}1, 1{\cdot}1\rangle a\langle 2{\cdot}1, 3{\cdot}1\rangle$. Therefore, it is the sole candidate that can lead to a rigid generalization $\tilde{g}'$ of $\tilde{g}_1$ and $\tilde{g}_2$ so that $\tilde{g}' \simeq \tilde{g}_1$. $\mathfrak{G}_{\mathfrak{a}}^{4\mathcal{V}}$ computes the following generalization $\tilde{g}'$ for $\tilde{g}_1$ and $\tilde{g}_2$ w.r.t. $\mathfrak{a}$:

$$\tilde{g}' = (\tilde{Z}_1(a), \tilde{z}_2, \tilde{Z}_3(a), \tilde{z}_4).$$

The compressed form of $\tilde{g}_1$ is $(\tilde{X}_1(a), \tilde{X}_2(a))$ and the compressed form of $\tilde{g}'$ is $(\tilde{Z}_1(a), \tilde{Z}_3(a))$. Therefore $\tilde{g}_1 \preceq \tilde{g}_2$ and $\tilde{g}_1$ can be removed so that $\{\tilde{g}_2, \tilde{g}_3\}$ remain in the complete set of generalizations of $\tilde{s}$ and $\tilde{q}$ with respect to the set of all laas. Next we test $\tilde{g}_2 \stackrel{?}{\preceq} \tilde{g}_3$. The only candidate alignment is $\mathfrak{a} = a\langle 1{\cdot}1, 2{\cdot}1{\cdot}1\rangle a\langle 3{\cdot}1, 3{\cdot}1\rangle$. $\mathfrak{G}_{\mathfrak{a}}^{4\mathcal{V}}$ computes the following generalization $\tilde{g}'$ for $\tilde{g}_2$ and $\tilde{g}_3$ w.r.t. $\mathfrak{a}$:

$$\tilde{g}' = (\tilde{x}_1, \tilde{X}_2(a), \tilde{x}_3, \tilde{X}_4(a)).$$

The compressed form of $\tilde{g}_2$ is $(\tilde{Y}_1(a), \tilde{Y}_3(a))$ and the compressed form of $\tilde{g}'$ is $(\tilde{X}_2(a), \tilde{X}_4(a))$. Therefore, $\{\tilde{g}_3\}$ is the mcg. Notice that the compressed form of $\tilde{g}_3$ is $\tilde{g}_3$ itself, hence $\tilde{g}' \prec \tilde{g}_3$ and it follows that also $\tilde{g}_2 \prec \tilde{g}_3$ and $\tilde{g}_1 \prec \tilde{g}_3$.

Chapter 3

Anti-Unification for Ranked Terms with Binders

A computer program can be seen as a function from input values to an output. In order to define the behavior of such a function we need to address the input values that are transformed during the evaluation process. Therefore, we give them some temporary names, like x, y, etc., that are of no interest for the user. Similarly, mathematical expressions, like $x^2 + y + 1$ where the variables x, y denote unknown subjects, may be described in a schematic way as functions:

$$f(x, y) \;=\; x^2 + y + 1$$

The variables x and y are bounded in the scope of the function definition. We can rename them by z_1 and z_2 without changing the semantics of f:

$$f(z_1, z_2) \;=\; z_1^2 + z_2 + 1$$

Church [27] developed the lambda calculus to represent computable functions in a purely syntactic manner. The notion of alpha equivalence defines the class of terms that are equal up to *bound variable* renaming. For instance, the above examples are two alpha equivalent lambda terms:

$$\lambda x.\lambda y.\, x^2 + y + 1 \;=_\alpha\; \lambda z_1.\lambda z_2.\, z_1^2 + z_2 + 1$$

Pitts and Gabbay [37, 38, 39] introduced nominal techniques to study first-order systems with bindings. They distinguish between *atoms* that can be bound, and *variables* that can be instantiated. In contrast to the lambda calculus where bound variables may appear in functional position, atoms do not appear in functional position. Furthermore, the name of a bound atom is used as a label for the binding. Alpha equivalence is essentially defined on swapping of atoms. The above examples can also be encoded by nominal terms, since no higher-order variables are involved. We use the atoms a, b, and c in the example:

$$\mathsf{a}.\mathsf{b}.\, \mathsf{a}^2 + \mathsf{b} + 1 \;=_\alpha\; \mathsf{c}_1.\mathsf{c}_2.\, \mathsf{c}_1^2 + \mathsf{c}_2 + 1$$

We can prove alpha equivalence by applying the swappings $(\mathsf{a}\,\mathsf{c}_1)$ and $(\mathsf{b}\,\mathsf{c}_2)$ to the left-hand side term or to the right-hand side term (the formal definitions can be found in subsection 3.1.1):

$$(\mathsf{b}\,\mathsf{c}_2)\bullet((\mathsf{a}\,\mathsf{c}_1)\bullet \mathsf{a.b.}\,\mathsf{a}^2+\mathsf{b}+1) = (\mathsf{b}\,\mathsf{c}_2)\bullet \mathsf{c}_1.\mathsf{b.}\,\mathsf{c}_1^2+\mathsf{b}+1 = \mathsf{c}_1.\mathsf{c}_2.\,\mathsf{c}_1^2+\mathsf{c}_2+1$$
$$(\mathsf{b}\,\mathsf{c}_2)\bullet((\mathsf{a}\,\mathsf{c}_1)\bullet \mathsf{c}_1.\mathsf{c}_2.\,\mathsf{c}_1^2+\mathsf{c}_2+1) = (\mathsf{b}\,\mathsf{c}_2)\bullet \mathsf{a.c}_2.\,\mathsf{a}^2+\mathsf{c}_2+1 = \mathsf{a.b.}\,\mathsf{a}^2+\mathsf{b}+1$$

When dealing with languages that involve binders together with substitutions, one needs to be careful to avoid unwanted capturing by the scope of a binder at instantiation time. The approach in lambda calculus is to avoid variable capturing by alpha equivalent renaming so that the variables in a substitution do not interfere with bound variables of the term to be instantiated. For instance, $(\lambda x.\lambda y.\, x^2+y+z)\{z \mapsto x\}$ becomes $\lambda x_1.\lambda y.\, x_1^2+y+x$. Nominal techniques use a so called *freshness context* in order to specify atoms which are forbidden in a substitution when instantiating a variable. Atoms that are not forbidden might be captured by a binder. For instance, $\mathsf{a.b.}\,\mathsf{a}^2+\mathsf{b}+z$ $\{z \mapsto \mathsf{a}\} = \mathsf{a.b.}\,\mathsf{a}^2+\mathsf{b}+\mathsf{a}$ if a is not forbidden in the instantiation of z. A pair of a nominal term and a freshness context is called a term-in-context.

In the present chapter, we develop anti-unification algorithms for nominal terms-in-context and for simply-typed lambda terms. Both of them rely on a subalgorithm that constructively decides an extended variant of alpha equivalence which is needed to compute *least general* generalizations for two input terms. We prove soundness and completeness properties of the main algorithms and their subalgorithms, and analyze their complexity. Several authors, e.g. [24, 32, 55], showed that there is a close relation between nominal terms and higher-order pattern. In particular, Levy and Villaret [55] showed how to translate nominal unification problems into higher-order pattern unification problems and how to obtain nominal unifiers back from higher-order pattern unifiers.

Our original intention was to "kill two birds with one stone" by developing an anti-unification algorithm that computes higher-order pattern generalizations of two λ-terms and introducing a similar translation approach to solve the anti-unification problem for nominal terms. However, it turned out that the nature of freshness contexts considered in nominal terms-in-context introduces some difficulties when searching for a meet. In general, a minimal complete set of generalizations does not exist for two nominal terms-in-context. This is in sharp contrast with the related problem of anti-unification for higher-order patterns, which always have a single lgg. It is not clear how such a translation approach would work. Therefore, we developed two algorithms that are independent of each other:

1. We discuss the problem of searching a meet for two nominal terms-in-context and illustrate the problem that arises. To tackle that problem, we restrict the set of considered atoms that are allowed in a generalization to be finite and fixed. Under this restriction the anti-unification problem for two terms-in-context becomes unitary. We develop an anti-unification algorithm that computes a unique lgg for two terms-in-context and a finite set of atoms. The upper bound of its runtime complexity is $O(n^5)$ and it needs $O(n^4)$ space, where n is the input size.
2. We consider the anti-unification problem for simply-typed lambda terms and develop a rule-based algorithm that computes a least general higher-order pattern generalization for two arbitrary simply-typed λ-terms. The upper bound for the runtime complexity of this algorithm is quadratic in the size of the two input terms and it needs linear space in the size of the input.

3.1 Anti-Unification for Nominal Terms

Here we study the nominal anti-unification problem, which is concerned with finding a least general generalization for two terms-in-context. In general, the problem is of type zero, but if the set of atoms permitted in generalizations is finite, then there exists a unique lgg (modulo $\simeq_{=_\alpha}$). We present an algorithm that computes it. The algorithm relies on a subalgorithm that constructively decides equivariance between two terms-in-context. Nominal anti-unification can be applied to problems were generalization of first-order terms is needed (inductive learning, clone detection, etc.), but bindings are involved. Languages that are based on nominal logic are, for instance, αProlog [26], FreshML [79], Nominal Isabelle [81], etc. We start with an illustrating example that shows two nominal terms and their generalization. Notice that in Example 3.1 the variable x can be reused three times, even though the binders $\mathsf{a}.$ and $\mathsf{b}.$ refer to atoms at different levels in the tree representation of the terms to be generalized.

Example 3.1. *The nominal term* $\mathsf{a}.f(x, g((\mathsf{a}\,\mathsf{c})(\mathsf{a}\,\mathsf{b})\cdot x, (\mathsf{a}\,\mathsf{d})(\mathsf{a}\,\mathsf{c})\cdot x), \mathsf{d})$ *is a generalization of the two nominal terms* $\mathsf{a}.f(\mathsf{a}, g(\mathsf{b}, \mathsf{c}), \mathsf{d})$ *and* $\mathsf{b}.f(\mathsf{c}, g(\mathsf{b}, \mathsf{d}), \mathsf{d})$*. The first term can be obtained from the generalization by instantiating the variable x with the atom* a *and the second one by replacing x with* c*. The swappings in front of x are applied immediately at instantiation time.*

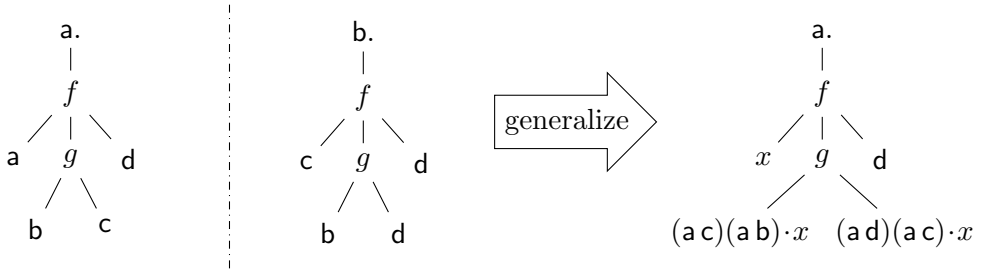

Figure 3.1: Two nominal terms and their lgg.

3.1.1 Nominal Terms (Preliminaries)

Characters that denote function symbols, terms, term variables, substitutions, etc. and standard notations, like the cardinality of a set, are the same as in the previous chapters.

Definition 3.1 (Nominal signature)**.** *In* nominal signatures *we have a set of* sorts of atoms*, a disjoint set of* sorts of data*, and a set of* function symbols*. Sorts of atoms are denoted by ν and sorts of data by δ. Function symbols have an arity of the form $\tau_1 \times \cdots \times \tau_n \to \delta$, where τ_i are sorts given by the grammar:*

$$\tau ::= \nu \mid \delta \mid \langle\nu\rangle\tau$$

Sorts of the form $\langle\nu\rangle\tau$ classify terms that are binding abstractions of atoms of sort ν over terms of sort τ.

Example 3.2. *In Figure 3.2, we illustrate an example of a nominal signature for expressions in a small fragment of ML. It is taken from Urban et al. [83].*

sort of atoms:	*vid*
sort of data:	*exp*
function symbols:	vr: $vid \to exp$
	app: $exp \times exp \to exp$
	fn: $\langle vid \rangle exp \to exp$
	lv: $exp \times \langle vid \rangle exp \to exp$
	lf: $\langle vid \rangle((\langle vid \rangle exp) \times exp) \to exp$.

Figure 3.2: Nominal signature for expressions in a small fragment of ML

Definition 3.2 (Nominal term)**.** *Given disjoint sets of countably infinite* term variables $\mathcal{V}^\dagger$*, countably infinite* atoms $\mathcal{A}^\ddagger$*, and a signature* Σ*.*

A swapping $(\mathsf{a}\,\mathsf{b})$ *is a pair of atoms* $\mathsf{a}, \mathsf{b} \in \mathcal{A}$ *of the same sort. A* permutation *is a (possibly empty) sequence of swappings.* Nominal terms *are given by the grammar:*

$$t ::= f(t_1, \ldots, t_n) \mid \mathsf{a} \mid \mathsf{a}.t \mid \pi{\cdot}x$$

where f *is an* n*-ary function symbol,* a *is an atom,* π *is a permutation, and* x *is a variable. They are called respectively* application, atom, abstraction, *and* suspension.

We denote atoms by upright letters $\mathsf{a}, \mathsf{b}, \mathsf{c}, \mathsf{d}, \mathsf{e}$. The letter $\hbar$ denotes an atom or a function symbol. Upright Greek letters π, ρ, μ, τ are used to denote permutations. The *inverse* of a permutation $\pi = (\mathsf{a}_1\,\mathsf{b}_1)\ldots(\mathsf{a}_n\,\mathsf{b}_n)$ is the permutation $(\mathsf{a}_n\,\mathsf{b}_n)\ldots(\mathsf{a}_1\,\mathsf{b}_1)$, denoted by π^{-1}. The identity permutation is denoted by Id and instead of $Id{\cdot}x$ we write just x. The set of nominal terms constructed over Σ, $\mathcal{A}$, and $\mathcal{V}$ is denoted by $\mathcal{T}(\Sigma, \mathcal{A}, \mathcal{V})$, or simply by $\mathcal{T}$ if the concrete instances of Σ, $\mathcal{A}$, and $\mathcal{V}$ are unimportant.

Definition 3.3 (Permutation)**.** *The action of a permutation* π *on a term* t *is defined by induction on the number of swappings in* π*:*

$$Id{\bullet}t = t; \qquad (\mathsf{a}\,\mathsf{b})\pi{\bullet}t = (\mathsf{a}\,\mathsf{b}){\bullet}(\pi{\bullet}t).$$

where the effect of a swapping is defined by induction on the structure of terms:

$$(\mathsf{a}\,\mathsf{b}){\bullet}f(t_1, \ldots, t_n) = f((\mathsf{a}\,\mathsf{b}){\bullet}t_1, \ldots, (\mathsf{a}\,\mathsf{b}){\bullet}t_n); \qquad (\mathsf{a}\,\mathsf{b}){\bullet}(\mathsf{c}.t) = ((\mathsf{a}\,\mathsf{b}){\bullet}\mathsf{c})\,.\,((\mathsf{a}\,\mathsf{b}){\bullet}t)\,;$$

$$(\mathsf{a}\,\mathsf{b}){\bullet}\mathsf{a} = \mathsf{b}; \qquad (\mathsf{a}\,\mathsf{b}){\bullet}\mathsf{b} = \mathsf{a}; \qquad (\mathsf{a}\,\mathsf{b}){\bullet}\mathsf{c} = \mathsf{c}, \text{ when } \mathsf{c} \notin \{\mathsf{a}, \mathsf{b}\}; \qquad (\mathsf{a}\,\mathsf{b}){\bullet}\pi{\cdot}x = (\mathsf{a}\,\mathsf{b})\pi{\cdot}x.$$

Like in the previous chapters, the set of variables of a nominal term t is denoted by $\mathcal{V}(t)$. A term t is called *ground* if $\mathcal{V}(t) = \varnothing$. We define the set of *atoms* of a permutation π as the set $\{\mathsf{a} \mid \pi{\bullet}\mathsf{a} \neq \mathsf{a}\}$, denoted by $\mathcal{A}(\pi)$. Similarly, the set of atoms of a term t is the set of all atoms which appear in it and is denoted by $\mathcal{A}(t)$. For instance, $\mathcal{A}(f(\mathsf{a}.g(\mathsf{a}), (\mathsf{b}\,\mathsf{c}){\cdot}x, \mathsf{d}) = \{\mathsf{a}, \mathsf{b}, \mathsf{c}, \mathsf{d}\}$. We extend the notion $\mathcal{A}(\cdot)$ for subsets of $\mathcal{T}$.

[†]We assume that $\mathcal{V}$ contains countably infinite variables of each sort of atoms and sort of data.

[‡]We assume that $\mathcal{A}$ contains countably infinite atoms of each sort of atoms.

The set of *positions* of a term t is denoted by $\mathrm{Pos}(t)$. Positions are defined with respect to the tree representation in the usual way, as strings of integers. However, suspensions are put in a single leaf node. For any term t, we denote by $t|_I$ the subterm of t at position I.

Example 3.3. *Figure 3.3 illustrates the tree form of the term* $f(\mathsf{a}.\mathsf{b}.g((\mathsf{a}\,\mathsf{b})\cdot x, \mathsf{a}), h(\mathsf{c}))$*, and the corresponding positions. The application of* f *stands in the position* ϵ *(the empty sequence). The suspension is put in one node of the tree, at the position* $1{\cdot}1{\cdot}1{\cdot}1$*. The abstraction operator and the corresponding bound atom together occupy one node as well. The subterm* $\mathsf{b}.g((\mathsf{a}\,\mathsf{b})\cdot x, \mathsf{a})$ *at position* $1{\cdot}1$ *can be denoted by* $f(\mathsf{a}.\mathsf{b}.g((\mathsf{a}\,\mathsf{b})\cdot x, \mathsf{a}), h(\mathsf{c}))|_{1\cdot 1} = \mathsf{b}.g((\mathsf{a}\,\mathsf{b})\cdot x, \mathsf{a})$.

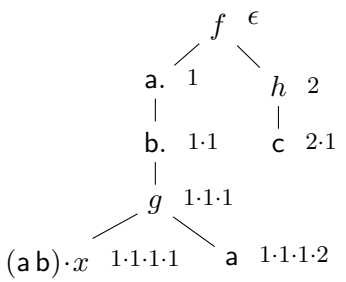

Figure 3.3: The tree form and positions of the nominal term $f(\mathsf{a}.\mathsf{b}.g((\mathsf{a}\,\mathsf{b})\cdot x, \mathsf{a}), h(\mathsf{c}))$.

As usual, we denote by $\|t\|$ the cardinality of the set of positions of a nominal term t. Similarly, $\|t\|_{\mathsf{Abs}}$ stand for the number of abstraction occurrences in t, i.e., the cardinality $|\{I \mid I \in \mathrm{Pos}(t) \text{ and } t|_I = \mathsf{a}.t' \text{ for some } \mathsf{a} \in \mathcal{A} \text{ and } t' \in \mathcal{T}\}|$.

Every permutation π naturally defines a bijective function from the set of atoms to the sets of atoms, that we will also represent as π. Suspensions are uses of variables with a permutation of atoms waiting to be applied once the variable is instantiated. Occurrences of an atom a are said to be bound if they are in the scope of an abstraction of a, otherwise are said to be free.

Definition 3.4 (Free atoms)**.** *We denote by* $\mathrm{FA}(t)$ *the set of all atoms which occur freely in* t *and by* $\mathrm{FA}^{\text{-}s}(t)$ *is the set of all atoms which occur freely in* t *ignoring suspensions:*

$$\begin{array}{ll} \mathrm{FA}(f(t_1,\ldots,t_n)) = \bigcup_{i=1}^{n} \mathrm{FA}(t_i), & \mathrm{FA}^{\text{-}s}(f(t_1,\ldots,t_n)) = \bigcup_{i=1}^{n} \mathrm{FA}^{\text{-}s}(t_i), \\ \mathrm{FA}(\mathsf{a}) = \{\mathsf{a}\}, \quad \mathrm{FA}(\mathsf{a}.t) = \mathrm{FA}(t)\backslash\{\mathsf{a}\}, & \mathrm{FA}^{\text{-}s}(\mathsf{a}) = \{\mathsf{a}\}, \quad \mathrm{FA}^{\text{-}s}(\mathsf{a}.t) = \mathrm{FA}^{\text{-}s}(t)\backslash\{\mathsf{a}\}, \\ \mathrm{FA}(\pi\cdot x) = \mathcal{A}(\pi), & \mathrm{FA}^{\text{-}s}(\pi\cdot x) = \varnothing. \end{array}$$

In Definition 3.5, we overload the notion of a top symbol (also called head of the term) from the previous chapter and Definition 3.6 overloads the definition of a substitution for nominal terms.

Definition 3.5 (Top symbol)**.** *The head (or top symbol) of a term* t*, denoted* $\mathrm{Top}(t)$*, is defined as:* $\mathrm{Top}(f(t_1,\ldots,t_n)) = f$*,* $\mathrm{Top}(\mathsf{a}) = \mathsf{a}$*,* $\mathrm{Top}(\mathsf{a}.t) = .$*, and* $\mathrm{Top}(\pi\cdot x) = x$*.*

Definition 3.6 (Substitution)**.** *A* substitution *is a mapping* $\sigma : \mathcal{V} \to \mathcal{T}(\Sigma, \mathcal{A}, \mathcal{V})$ *from variables to terms of the same sort, which is identity almost everywhere.*

Any substitution σ can be extended to a mapping $\hat{\sigma} : \mathcal{T} \to \mathcal{T}$ that can be applied to nominal terms. Application allows atom capture and forces the permutation effect. Similarly to subsection 2.1.1, the application is defined by induction on the structure of terms:

$$t\hat{\sigma} ::= \begin{cases} f(t_1\hat{\sigma},\ldots,t_n\hat{\sigma}) & \text{if } t = f(t_1,\ldots,t_n), \\ \mathsf{a} & \text{if } t = \mathsf{a}, \\ \mathsf{a}.(t\hat{\sigma}) & \text{if } t = \mathsf{a}.t, \\ \pi\bullet\sigma(x) & \text{if } t = \pi\cdot x \end{cases}$$

The notion of substitution *domain* and *range* are defined similarly to subsection 2.1.1. To simplify the notation, we do not distinguish between a substitution σ and its extension $\hat{\sigma}$.

Example 3.4. *For instance, in* $\mathsf{a}.x\{x \mapsto \mathsf{a}\} = \mathsf{a}.\mathsf{a}$ *the atom* a *is captured, and in* $\pi{\cdot}x\{x \mapsto t\} = \pi{\bullet}t$ *the permutation is immediately applied to* t*. A more concrete example is* $(\mathsf{a}\,\mathsf{b}){\cdot}x\{x \mapsto f(\mathsf{a}, (\mathsf{a}\,\mathsf{b}){\cdot}y)\} = f(\mathsf{b}, (\mathsf{a}\,\mathsf{b})(\mathsf{a}\,\mathsf{b}){\cdot}y)$.

Definition 3.7 (Freshness context)**.** *A* freshness constraint *is a pair of the form* $\mathsf{a}\#x$ *stating that the instantiation of* x *cannot contain free occurrences of* a*. A* freshness context *is a finite set of freshness constraints.*

We use ∇ and Γ to denote freshness contexts. $\mathcal{V}(\nabla)$ and $\mathcal{A}(\nabla)$ denote respectively the set of variables and atoms of ∇.

Definition 3.8. *We say that a substitution* σ respects *a freshness context* ∇*, if for all* x*,* $\mathrm{FA}^{\text{-}s}(x\sigma)$ *and* $\{\mathsf{a} \mid \mathsf{a}\#x \in \nabla\}$ *are disjoint, i.e.,* $\mathrm{FA}^{\text{-}s}(x\sigma) \cap \{\mathsf{a} \mid \mathsf{a}\#x \in \nabla\} = \varnothing$.

Definition 3.9 (Alpha equivalence and freshness predicate)**.** *The predicate* $=_\alpha$*, which stands for* α*-equivalence between terms, and the freshness predicate* $\#$ *were defined in [82, 83] by the following theory:*

$$\frac{}{\nabla \vdash \mathsf{a} =_\alpha \mathsf{a}}\,(=_\alpha\text{-atom}) \qquad \frac{\nabla \vdash t_1 =_\alpha t'_1 \quad \cdots \quad \nabla \vdash t_n =_\alpha t'_n}{\nabla \vdash f(t_1, \ldots t_n) =_\alpha f(t'_1, \ldots, t'_n)}\,(=_\alpha\text{-application})$$

$$\frac{\nabla \vdash t =_\alpha t'}{\nabla \vdash \mathsf{a}.t =_\alpha \mathsf{a}.t'}\,(=_\alpha\text{-abs-1}) \qquad \frac{\mathsf{a} \neq \mathsf{a}' \quad \nabla \vdash t =_\alpha (\mathsf{a}\,\mathsf{a}'){\bullet}t' \quad \nabla \vdash \mathsf{a}\#t'}{\nabla \vdash \mathsf{a}.t =_\alpha \mathsf{a}'.t'}\,(=_\alpha\text{-abs-2})$$

$$\frac{\mathsf{a}\#x \in \nabla \text{ for all } \mathsf{a} \text{ such that } \pi{\bullet}\mathsf{a} \neq \pi'{\bullet}\mathsf{a}}{\nabla \vdash \pi{\cdot}x =_\alpha \pi'{\cdot}x}\,(=_\alpha\text{-susp.})$$

where the freshness predicate $\#$ *is defined by*

$$\frac{\mathsf{a} \neq \mathsf{a}'}{\nabla \vdash \mathsf{a}\#\mathsf{a}'}\,(\#\text{-atom}) \qquad \frac{\nabla \vdash \mathsf{a}\#t_1 \quad \cdots \quad \nabla \vdash \mathsf{a}\#t_n}{\nabla \vdash \mathsf{a}\#f(t_1, \ldots t_n)}\,(\#\text{-application})$$

$$\frac{}{\nabla \vdash \mathsf{a}\#\mathsf{a}.t}\,(\#\text{-abst-1}) \qquad \frac{\mathsf{a} \neq \mathsf{a}' \quad \nabla \vdash \mathsf{a}\#t}{\nabla \vdash \mathsf{a}\#\mathsf{a}'.t}\,(\#\text{-abst-2})$$

$$\frac{(\pi^{-1}{\bullet}\mathsf{a}\#x) \in \nabla}{\nabla \vdash \mathsf{a}\#\pi{\cdot}x}\,(\#\text{-susp.})$$

The intended meanings of the two predicates from Definition 3.9 are:

1. $\nabla \vdash \mathsf{a}\#t$ holds, if for every substitution σ such that $t\sigma$ is a ground term and σ respects the freshness context ∇, we have a is not free in $t\sigma$;
2. $\nabla \vdash t =_\alpha u$ holds, if for every substitution σ such that $t\sigma$ and $u\sigma$ are ground terms and σ respects the freshness context ∇, $t\sigma$ and $u\sigma$ are α-equivalent.

Based on the definition of the freshness predicate #, we can design an algorithm which solves the following problem:

Given: A set of freshness formulas $\{\mathsf{a}_1\#t_1, \ldots, \mathsf{a}_n\#t_n\}$.

Compute: A *minimal* (with respect to $\subseteq$) freshness context ∇ such that $\nabla \vdash \mathsf{a}_1\#t_1, \ldots, \nabla \vdash \mathsf{a}_n\#t_n$.

Such a freshness context ∇ may or may not exist, and the algorithm should detect it. Essentially, the algorithm is a bottom-up application of the rules of the freshness predicate, starting from each of the $\nabla \vdash \mathsf{a}_1\#t_1, \ldots, \nabla \vdash \mathsf{a}_n\#t_n$. It succeeds if each branch of such a derivation tree is either closed (i.e., ends with the application of the #-atom or the #-abst-1 rule), or ends with an application of the #-susp. rule, producing a membership atom of the form $\mathsf{a}\#x \in \nabla$ for some a and x.

Definition 3.10 (Minimal freshness context algorithm)**.** *We give a rule-based description of the algorithm, which we call $\mathcal{C}t\mathcal{X}$ for it is supposed to compute a freshness context. The rules operate on pairs $F; \nabla$, where F is a set of atomic freshness formulas of the form $\mathsf{a}\#t$, and ∇ is the freshness context to be computed. The pair $F; \nabla$ is called the state. The algorithm $\mathcal{C}t\mathcal{X}$ consists of five transformation rules that operate on states. The rules are presented below.*

The following transformation rules of $\mathcal{C}t\mathcal{X}$ transform states into states. The character $\cup\!\!\!\cdot$ stands for disjoint union.

Del-$\mathcal{C}t\mathcal{X}$: **Delete (#-atom)**

$\{\mathsf{a}\#\mathsf{b}\} \cup\!\!\!\cdot\, F; \nabla \Longrightarrow F; \nabla, \qquad \text{if } \mathsf{a} \neq \mathsf{b}.$

Abs-$\mathcal{C}t\mathcal{X}$ 1: **Abstraction (#-abst-1)**

$\{\mathsf{a}\#\mathsf{a}.t\} \cup\!\!\!\cdot\, F; \nabla \Longrightarrow F; \nabla.$

Abs-$\mathcal{C}t\mathcal{X}$ 2: **Abstraction (#-abst-2)**

$\{\mathsf{a}\#\mathsf{b}.t\} \cup\!\!\!\cdot\, F; \nabla \Longrightarrow \{\mathsf{a}\#t\} \cup F; \nabla, \qquad \text{if } \mathsf{a} \neq \mathsf{b}.$

Dec-$\mathcal{C}t\mathcal{X}$: **Decomposition (#-application)**

$\{\mathsf{a}\#f(t_1, \ldots, t_n)\} \cup\!\!\!\cdot\, F; \nabla \Longrightarrow \{\mathsf{a}\#t_1, \ldots, \mathsf{a}\#t_n\} \cup F; \nabla.$

Sus-$\mathcal{C}t\mathcal{X}$: **Suspension (#-susp.)**

$\{\mathsf{a}\#\pi\cdot x\} \cup\!\!\!\cdot\, F; \nabla \Longrightarrow F; \{\pi^{-1}\bullet\mathsf{a}\#x\} \cup \nabla.$

To compute a minimal freshness context which justifies the atomic freshness formulas $\mathsf{a}_1\#t_1, \ldots, \mathsf{a}_n\#t_n$, we start with $\{\mathsf{a}_1\#t_1, \ldots, \mathsf{a}_n\#t_n\}; \varnothing$ and apply the rules of $\mathcal{C}t\mathcal{X}$ as long as possible. It is easy to see that the algorithm terminates. The state to which no rule applies has either the form $\varnothing; \nabla$ or $\{\mathsf{a}\#\mathsf{a}\} \cup F; \nabla$, where ∇ is a freshness context. In the former case we say that the algorithm succeeds and computes ∇, writing this fact as $\mathcal{C}t\mathcal{X}(\{\mathsf{a}_1\#t_1, \ldots, \mathsf{a}_n\#t_n\}) = \nabla$. In the latter case we say that $\mathcal{C}t\mathcal{X}$ fails and write $\mathcal{C}t\mathcal{X}(\{\mathsf{a}_1\#t_1, \ldots, \mathsf{a}_n\#t_n\}) = \bot$.

Theorem 3.1. *Let F be a set of freshness formulas and ∇ be a freshness context. Then $\mathcal{C}t\mathcal{X}(F) \subseteq \nabla$ iff $\nabla \vdash \mathsf{a}\#t$ for all $\mathsf{a}\#t \in F$.*

Proof. By the structural induction over t, exploiting the similarity between the rules of $\mathcal{C}t\mathcal{x}$ and the definition of the freshness predicate $\#$. □

Corollary 3.2. *$\mathcal{C}t\mathcal{x}(F) = \bot$ iff there is no freshness context that would justify all formulas in F.*

Definition 3.11 (Context substitution application)**.** *The application of a substitution σ to a freshness context ∇ is defined by $\nabla\sigma = \mathcal{C}t\mathcal{x}(\{\mathsf{a}\#x\sigma \mid \mathsf{a}\#x \in \nabla\})$. When $\nabla\sigma \neq \bot$, we call $\nabla\sigma$ the* instance *of ∇ under σ.*

Example 3.5. *We illustrate the application of the substitution $\sigma = \{x \mapsto \mathsf{b}.f(\mathsf{a},\mathsf{b}), y \mapsto (\mathsf{a}\,\mathsf{b})\cdot x\}$ to the freshness context $\nabla = \{\mathsf{c}\#x, \mathsf{a}\#y\}$. The algorithm $\mathcal{C}t\mathcal{x}$ is initialized by the state $\{\mathsf{c}\#\mathsf{b}.f(\mathsf{a},\mathsf{b}), \mathsf{a}\#(\mathsf{a}\,\mathsf{b})\cdot x\}; \varnothing$. Now we apply the rules as long as possible:*

$$\begin{aligned}
& \{\mathsf{c}\#\mathsf{b}.f(\mathsf{a},\mathsf{b}), \mathsf{a}\#(\mathsf{a}\,\mathsf{b})\cdot x\}; \varnothing \\
\Longrightarrow_{\mathsf{Abs}\text{-}\mathcal{C}t\mathcal{x}\,2} \quad & \{\mathsf{c}\#f(\mathsf{a},\mathsf{b}), \mathsf{a}\#(\mathsf{a}\,\mathsf{b})\cdot x\}; \varnothing \\
\Longrightarrow_{\mathsf{Dec}\text{-}\mathcal{C}t\mathcal{x}} \quad & \{\mathsf{c}\#\mathsf{a}, \mathsf{c}\#\mathsf{b}, \mathsf{a}\#(\mathsf{a}\,\mathsf{b})\cdot x\}; \varnothing \\
\Longrightarrow^{2}_{\mathsf{Del}\text{-}\mathcal{C}t\mathcal{x}} \quad & \{\mathsf{a}\#(\mathsf{a}\,\mathsf{b})\cdot x\}; \varnothing \\
\Longrightarrow_{\mathsf{Sus}\text{-}\mathcal{C}t\mathcal{x}} \quad & \varnothing; \{\mathsf{b}\#x\}
\end{aligned}$$

Therefore $\{\mathsf{b}\#x\}$ is the instance of ∇ under σ.

Lemma 3.3. *σ respects ∇ iff $\nabla\sigma \neq \bot$.*

Proof. ($\Rightarrow$) Assume that σ respects ∇, i.e., for all x, $\mathrm{FA}^{\text{-s}}(x\sigma) \cap \{\mathsf{a} \mid \mathsf{a}\#x \in \nabla\} = \varnothing$. It follows that, for all $\mathsf{a}\#x \in \nabla$, $x\sigma$ is so that a only appears in suspensions. Therefore, when applying $\mathcal{C}t\mathcal{x}(\{\mathsf{a}\#x\sigma \mid \mathsf{a}\#x \in \nabla\}) = \mathcal{C}t\mathcal{x}(\{\mathsf{a}_1\#t_1, \ldots, \mathsf{a}_n\#t_n\})$ we know that, for all $\mathsf{a}_i\#t_i$, $1 \leqslant i \leqslant n$ holds $\mathsf{a}_i \notin \mathrm{FA}^{\text{-s}}(t_i)$. For that reason, the application of $\mathcal{C}t\mathcal{x}$ will never lead to a freshness formula of the form $\mathsf{a}\#\mathsf{a}$, which would lead to $\bot$.

($\Leftarrow$) Assume that $\nabla\sigma = \mathcal{C}t\mathcal{x}(\{\mathsf{a}_1\#t_1, \ldots, \mathsf{a}_n\#t_n\}) = \Gamma$. Since a freshness formula of the form $\mathsf{a}\#\mathsf{a}$ would lead to $\bot$, we know that, for all $\mathsf{a}_i\#t_i$, $1 \leqslant i \leqslant n$, a_i does not appear freely in t_i, i.e., $\mathsf{a}_i \notin \mathrm{FA}^{\text{-s}}(t_i)$. □

It is not hard to see that (a) if σ respects ∇, then σ respects any $\nabla' \subseteq \nabla$, and (b) if σ respects ∇ and ϑ respects $\nabla\sigma$, then $\sigma\vartheta$ respects ∇ and $(\nabla\sigma)\vartheta = \nabla(\sigma\vartheta)$.

Definition 3.12 (Term-in-context)**.** *A* term-in-context *is a pair $\langle \nabla, t\rangle$ of a freshness context ∇ and a nominal term t.*

Definition 3.13 (Instantiation ordering)**.** *A term-in-context $\langle \nabla_1, t_1\rangle$ is* more general *than a term-in-context $\langle \nabla_2, t_2\rangle$, written $\langle \nabla_1, t_1\rangle \preceq \langle \nabla_2, t_2\rangle$[†], if there exists a substitution σ, which respects ∇_1, such that $\nabla_1\sigma \subseteq \nabla_2$ and $\nabla_2 \vdash t_1\sigma =_\alpha t_2$. Similarly, we write $\nabla \vdash t_1 \preceq t_2$[†] if there exists a substitution σ such that $\nabla \vdash t_1\sigma =_\alpha t_2$. Two terms-in-context p_1 and p_2 are* equigeneral*, written $p_1 \simeq p_2$, iff $p_1 \preceq p_2$ and $p_2 \preceq p_1$. The strict part of $\preceq$ is denoted by $\prec$, i.e., $p_1 \prec p_2$ iff $p_1 \preceq p_2$ and not $p_2 \preceq p_1$. We also write $\nabla \vdash t_1 \simeq t_2$ iff $\nabla \vdash t_1 \preceq t_2$ and $\nabla \vdash t_2 \preceq t_1$.*

[†]For the sake of simplicity, we write $\preceq$ instead of using the logically consistent notation $\preceq_{=_\alpha}$.

Example 3.6. *We demonstrate this relations on some examples:*

- $\langle \{\mathsf{a}\#x\}, f(\mathsf{a})\rangle \simeq \langle \varnothing, f(\mathsf{a})\rangle$. *We can use* $\{x \mapsto \mathsf{b}\}$ *for the substitution applied to the first pair.*
- $\langle \varnothing, f(x)\rangle \preceq \langle \{\mathsf{a}\#x\}, f(x)\rangle$ *(with* $\sigma = Id$*), but not* $\langle \{\mathsf{a}\#x\}, f(x)\rangle \preceq \langle \varnothing, f(x)\rangle$.
- $\langle \varnothing, f(x)\rangle \preceq \langle \{\mathsf{a}\#y\}, f(y)\rangle$ *with* $\sigma = \{x \mapsto y\}$.
- $\langle \{\mathsf{a}\#x\}, f(x)\rangle \not\preceq \langle \varnothing, f(y)\rangle$, *because in order to satisfy* $\{\mathsf{a}\#x\}\sigma \subseteq \varnothing$, *the substitution* σ *should map* x *to a term* t *which contains neither* a *(freely) nor variables. But then* $\varnothing \vdash f(t) =_\alpha f(y)$ *does not hold. Hence, together with the previous example, we get* $\langle \varnothing, f(y)\rangle \prec \langle \{\mathsf{a}\#x\}, f(x)\rangle$.
- $\langle \{\mathsf{a}\#x\}, f(x)\rangle \not\preceq \langle \{\mathsf{a}\#x\}, f(\mathsf{a})\rangle$. *Notice that* $\sigma = \{x \mapsto \mathsf{a}\}$ *does not respect* $\{\mathsf{a}\#x\}$.
- $\langle \{\mathsf{b}\#x\}, (\mathsf{a}\,\mathsf{b})\cdot x\rangle \preceq \langle \{\mathsf{c}\#x\}, (\mathsf{a}\,\mathsf{c})\cdot x\rangle$ *with the substitution* $\sigma = \{x \mapsto (\mathsf{a}\,\mathsf{b})(\mathsf{a}\,\mathsf{c})\cdot x\}$. *Hence, we get* $\langle \{\mathsf{b}\#x\}, (\mathsf{a}\,\mathsf{b})\cdot x\rangle \simeq \langle \{\mathsf{c}\#x\}, (\mathsf{a}\,\mathsf{c})\cdot x\rangle$, *because the* $\succeq$ *part can be shown with the help of the substitution* $\{x \mapsto (\mathsf{a}\,\mathsf{c})(\mathsf{a}\,\mathsf{b})\cdot x\}$.

Definition 3.14. *A term-in-context* $\langle \Gamma, r\rangle$ *is called a* generalization *of two terms-in-context* $\langle \nabla_1, t\rangle$ *and* $\langle \nabla_2, s\rangle$ *if* $\langle \Gamma, r\rangle \preceq \langle \nabla_1, t\rangle$ *and* $\langle \Gamma, r\rangle \preceq \langle \nabla_2, s\rangle$. *It is a* least general generalization, *(lgg in short) of* $\langle \nabla_1, t\rangle$ *and* $\langle \nabla_2, s\rangle$ *if there is no generalization* $\langle \Gamma', r'\rangle$ *of* $\langle \nabla_1, t\rangle$ *and* $\langle \nabla_2, s\rangle$ *which satisfies* $\langle \Gamma, r\rangle \prec \langle \Gamma', r'\rangle$.

3.1.2 Nominal Anti-Unification from Type Zero to Type Unitary

If we have an infinite number of atoms in the language, the relation $\prec$ is not well-founded: $\langle \varnothing, x\rangle \prec \langle \{\mathsf{a}\#x\}, x\rangle \prec \langle \{\mathsf{a}\#x, \mathsf{b}\#x\}, x\rangle \prec \cdots$. As a consequence, two terms-in-context may not have an lgg and not even a minimal complete set of generalizations:

Example 3.7. *Let* $p_1 = \langle \varnothing, \mathsf{a}_1\rangle$ *and* $p_2 = \langle \varnothing, \mathsf{a}_2\rangle$ *be two terms-in-context where* a_1 *and* a_2 *are atoms of the same sort and* $\mathsf{a}_1 \neq \mathsf{a}_2$. *Then in any complete set of generalizations of* p_1 *and* p_2 *there is an infinite chain* $\langle \varnothing, x\rangle \prec \langle \{\mathsf{a}_3\#x\}, x\rangle \prec \langle \{\mathsf{a}_3\#x, \mathsf{a}_4\#x\}, x\rangle \prec \cdots$, *where* $\{\mathsf{a}_1, \mathsf{a}_2, \mathsf{a}_3, \ldots\}$ *is the set of all atoms of the same sort of the language.*

The following theorem characterizes the generalization type of nominal anti-unification:

Theorem 3.4. *The problem of anti-unification for terms-in-context is of type zero.*

Proof. Recall Definition 1.3: A theory is of generalization type *zero* iff there exists a pair of terms within the considered theory that does not have an mcg. Given a partially ordered set, Baader [8] showed that, in the case of unification types, it suffices to prove that there is an element p_0 such that for all elements $p_0 \preceq p$ there exists an element p' so that $p \prec p'$. We use this result in our proof. More precisely, we consider a complete set of generalizations G of the two terms-in-context p_1 and p_2 from Example 3.7 with respect to a countably infinite set of atoms $\mathcal{A}$ of the corresponding sort. There will be a unique most general term-in-context in G which plays the role of p_0, namely $\langle \varnothing, x\rangle$. Then we show that for all $p \in G$ (with $\langle \varnothing, x\rangle \preceq p$) there exists $p' \in G$ so that $p \prec p'$.

Since the atoms a_1 and a_2 (from Example 3.7) are not equal, their meet is a suspension of the form $\pi{\cdot}x$. Every pair of suspensions $\pi{\cdot}x$ and $\rho{\cdot}y$ is equivalent with respect to the instantiation ordering: $\pi{\cdot}x\{x \mapsto \pi^{-1}\rho{\cdot}y\} = \rho{\cdot}y$, and vise versa. W.l.o.g., we choose $Id{\cdot}x$ as representative of their meet.

The set of all finite subsets of $\mathcal{A}\backslash\{\mathsf{a}_1, \mathsf{a}_2\}$ is denoted by $\mathcal{A}$. (Notice that a freshness context is finite by Definition 3.7.) It follows that $G = \{\langle\{\mathsf{a}'\#x \mid \mathsf{a}' \in \mathcal{A}'\}, x\rangle \mid \mathcal{A}' \in \mathcal{A}\}$ is a complete set of generalizations for the two terms-in-context p_1 and p_2.

Let $\langle\nabla, x\rangle \in G$ be arbitrary but fixed. We know that ∇ is finite. Since $\mathcal{A}$ is countably infinite, there exists $\langle\nabla \cup \{\mathsf{a}'\#x\}, x\rangle \in G$ so that $\langle\nabla, x\rangle \prec \langle\nabla \cup \{\mathsf{a}'\#x\}, x\rangle$.

Hence, p_1 and p_2 do not have a minimal complete set of generalizations. □

The reason is one can make terms-in-context less and less general by adding freshness constraints for the available (infinitely many) atoms. However, if we restrict the set of atoms which can be used in the generalizations to be fixed and finite, then the anti-unification problem becomes unitary. (We do not prove this property here, it will follow from Theorem 3.11 and Theorem 3.12 in subsection 3.1.8.)

Definition 3.15 ($\mathcal{A}$-based)**.** *We say that, respectively, a term t, a freshness context ∇, a permutation π is* based *on a set of atoms $\mathcal{A}$, iff $\mathcal{A}(t) \subseteq \mathcal{A}$, $\mathcal{A}(\nabla) \subseteq \mathcal{A}$, $\mathcal{A}(\pi) \subseteq \mathcal{A}$. A term-in-context $\langle\nabla, t\rangle$ is based on $\mathcal{A}$ if both t and ∇ are based on it. An $\mathcal{A}$-based permutation defines a bijection from $\mathcal{A}$ to $\mathcal{A}$.*

Definition 3.16 ($\mathcal{A}$-based lgg)**.** *If p_1 and p_2 are $\mathcal{A}$-based terms-in-context, then their* $\mathcal{A}$-based generalizations *are terms-in-context which are generalizations of p_1 and p_2 and are based on $\mathcal{A}$. An* $\mathcal{A}$-based lgg *of $\mathcal{A}$-based terms-in-context p_1 and p_2 is a term-in-context p, which is an $\mathcal{A}$-based generalization of p_1 and p_2 and there is no $\mathcal{A}$-based generalization p' of p_1 and p_2 which satisfies $p \prec p'$.*

The problem we would like to solve is the following:

Given: Two nominal terms t and s of the same sort, a freshness context ∇, and a *finite* set of atoms $\mathcal{A}$ such that t, s, and ∇ are based on $\mathcal{A}$.

Find: A term r and a freshness context Γ, such that the term-in-context $\langle\Gamma, r\rangle$ is an $\mathcal{A}$-based least general generalization of the terms-in-context $\langle\nabla, t\rangle$ and $\langle\nabla, s\rangle$.

Our anti-unification problem is parametric on the set of atoms we consider as the base, and finiteness of this set is crucial to ensure the existence of an lgg.

Discussion about the set $\mathcal{A}$. Before we give the algorithm which solves the stated problem, we want to discuss how an $\mathcal{A}$-based lgg of two terms-in-context depends on the set of atoms $\mathcal{A}$. We start the discussion with an example:

Example 3.8. *Let $t = \mathsf{a}.\mathsf{b}$, $s = \mathsf{b}.\mathsf{a}$, $\nabla = \varnothing$, $\mathcal{A}_1 = \{\mathsf{a}, \mathsf{b}\}$, and $\mathcal{A}_2 = \{\mathsf{a}, \mathsf{b}, \mathsf{c}\}$. Then $\langle\varnothing, x\rangle$ is an $\mathcal{A}_1$-based lgg of $\langle\nabla, t\rangle$ and $\langle\nabla, s\rangle$, and $\langle\{\mathsf{c}\#x\}, \mathsf{c}.x\rangle$ is an $\mathcal{A}_2$-based lgg of them. Obviously, $\{\mathsf{c}\#x\} \vdash x \preceq \mathsf{c}.x$ but not $\{\mathsf{c}\#x\} \vdash \mathsf{c}.x \preceq x$.*

The reason why $\langle\varnothing, x\rangle$ is an $\mathcal{A}_1$-based lgg of $\langle\nabla, t\rangle$ and $\langle\nabla, s\rangle$ in Example 3.8 is that the only possible $\mathcal{A}_1$-based permutations are $(\mathsf{a}\,\mathsf{b})$ and Id. Since the atom b occurs

freely in t and a occurs freely in s, we can neither use $(\mathsf{a}\,\mathsf{b})$ for α-equivalent renaming of atoms in t nor can we use it to α-equivalently rename atoms in s.

When considering the set $\mathcal{A}_2$ as the base, we can rename $t =_\alpha (\mathsf{a}\,\mathsf{c})\bullet t = \mathsf{c}.\mathsf{b}$ and similarly we can use $s =_\alpha (\mathsf{b}\,\mathsf{c})\bullet s = \mathsf{c}.\mathsf{a}$. This leads to the $\mathcal{A}_2$-based lgg $\langle\{\mathsf{c}\#x\}, \mathsf{c}.x\rangle$.

Our observation naturally leads to the following two questions:

- Given two finite sets of atoms $\mathcal{A}_1$ and $\mathcal{A}_2$ with $\mathcal{A}_1 \subseteq \mathcal{A}_2$ and two $\mathcal{A}_1$-based terms-in-context $\langle\nabla, t\rangle$ and $\langle\nabla, s\rangle$.
- Given an $\mathcal{A}_1$-based lgg $\langle\Gamma_1, r_1\rangle$ and an $\mathcal{A}_2$-based lgg $\langle\Gamma_2, r_2\rangle$ of $\langle\nabla, t\rangle$ and $\langle\nabla, s\rangle$.
- *Can we get* $\Gamma_2 \vdash r_1 \simeq r_2$ *if there are enough atoms in* $\mathcal{A}_1$*?*
- *If yes, how many atoms do we need in* $\mathcal{A}_1$*?*

To answer this questions we introduce the notations of a *saturated set of atoms*. Let t, s be nominal terms, ∇ be a freshness context, and $\mathcal{A}$ be a set of atoms. The maximal subset of $\mathcal{A}$, *fresh* for t, s, and ∇, denoted $\mathrm{Fresh}(\mathcal{A}, t, s, \nabla)$, is defined as $\mathcal{A}\backslash(\mathcal{A}(t) \cup \mathcal{A}(s) \cup \mathcal{A}(\nabla))$.

Definition 3.17. *We say that a set of atoms* $\mathcal{A}$ *is* saturated *for* $\mathcal{A}$*-based* t, s *and* ∇*, if* $|\mathrm{Fresh}(\mathcal{A}, t, s, \nabla)| \geqslant \min(\|t\|_{\mathsf{Abs}}, \|s\|_{\mathsf{Abs}})$.

We will prove later (see Theorem 3.14) that the following conjecture holds. It answers the questions posed above:

Conjecture 3.1. *Let* $\mathcal{A}_1$ *and* $\mathcal{A}_2$ *be two finite sets of atoms with* $\mathcal{A}_1 \subseteq \mathcal{A}_2$ *such that the* $\mathcal{A}_1$*-based terms-in-context* $\langle\nabla, t\rangle$ *and* $\langle\nabla, s\rangle$ *have an* $\mathcal{A}_1$*-based lgg* $\langle\Gamma_1, r_1\rangle$ *and an* $\mathcal{A}_2$*-based lgg* $\langle\Gamma_2, r_2\rangle$*. If* $\mathcal{A}_1$ *is saturated for* t, s, ∇*, then* $\Gamma_2 \vdash r_1 \simeq r_2$*.*

In other words, this result answers the following question:

Given: Nominal terms t, s and a freshness context ∇.

Question: How to choose a set of atoms $\mathcal{A}$ so that

(a) t, s, ∇ are $\mathcal{A}$-based and

(b) the term r in the $\mathcal{A}$-based lgg $\langle\Gamma, r\rangle$ of $\langle\nabla, t\rangle$ and $\langle\nabla, s\rangle$ generalizes s and t in the "best way", maximally preserving similarities and uniformly abstracting differences between s and t.

Answer: Besides all the atoms $\mathcal{A}(t) \cup \mathcal{A}(s) \cup \mathcal{A}(\nabla)$, the set $\mathcal{A}$ should contain at least m more atoms, where $m = \min\{\|t\|_{\mathsf{Abs}}, \|s\|_{\mathsf{Abs}}\}$.

In the following subsection we introduce a rule-based algorithm that computes an $\mathcal{A}$-based lgg for two $\mathcal{A}$-based nominal terms and an $\mathcal{A}$-based freshness context. The algorithm is parametric on the finite set of atoms $\mathcal{A}$.

3.1.3 Nominal Anti-Unification Algorithm $\mathfrak{G}_{\mathsf{N}}$

We need to redefine the data structure of an anti-unification equation. It is similar to the one from subsection 2.1.2:

Definition 3.18 (Anti-unification equation)**.** *An* anti-unification equation*, AUE in short, is a triple* $x\colon t \triangleq s$*, where* x, t, s *have the same sort. The variable* x *is called a* generalization variable*.*

We say that a set P of AUEs is based on a finite set of atoms $\mathcal{A}$, if for all $x : t \triangleq s \in P$, the terms t and s are based on $\mathcal{A}$. The anti-unification algorithm has two global parameters and consists of four transformation rules that transform quadruples by rule application into quadruples of the same form.

Definition 3.19 (Nominal anti-unification algorithm)**.** *The nominal anti-unification algorithm is formulated in a rule-based way working on tuples* P; S; Γ; σ *and two global parameters* $\mathcal{A}$ *and* ∇*, where*

- P *and* S *are sets of AUEs such that if* $x\colon t \triangleq s \in P \cup S$*, then this is the sole occurrence of* x *in* $P \cup S$*;*
- P *is the set of AUEs to be solved;*
- $\mathcal{A}$ *is a finite set of atoms;*
- *The freshness context* ∇ *does not constrain generalization variables;*
- S *is a set of already solved AUEs (the store);*
- Γ *is a freshness context (computed so far) which constrains generalization variables;*
- σ *is a substitution (computed so far) mapping generalization variables to nominal terms;*
- P*,* S*,* ∇*, and* Γ *are* $\mathcal{A}$*-based.*

We call such a tuple a state *and the algorithm is called* $\mathfrak{G}_{\mathsf{N}}$*, where* N *stands for nominal. The rules below operate on states.*

In the transformation rules, we use the symbol y for fresh variables of the corresponding sorts. The symbol $\uplus$ stands for disjoint union.

Dec: **Decomposition**

$\{x\colon h(t_1, \ldots, t_m) \triangleq h(s_1, \ldots, s_m)\} \uplus P;\ S;\ \Gamma;\ \sigma$
$\quad\Longrightarrow \{y_1\colon t_1 \triangleq s_1, \ldots, y_m\colon t_m \triangleq s_m\} \cup P;\ S;\ \Gamma;\ \sigma\{x \mapsto h(y_1, \ldots, y_m)\},$

where h is a function symbol or an atom and $m \geqslant 0$.

Abs: **Abstraction**

$\{x\colon \mathsf{a}.t \triangleq \mathsf{b}.s\} \uplus P;\ S;\ \Gamma;\ \sigma \Longrightarrow \{y\colon (\mathsf{c}\,\mathsf{a})\bullet t \triangleq (\mathsf{c}\,\mathsf{b})\bullet s\} \cup P;\ S;\ \Gamma;\ \sigma\{x \mapsto \mathsf{c}.y\},$

where $\mathsf{c} \in \mathcal{A}$ and $\nabla \vdash \mathsf{c}\#\mathsf{a}.t$ and $\nabla \vdash \mathsf{c}\#\mathsf{b}.s$.

Sol: **Solving**

$\{x\colon t \triangleq s\} \uplus P;\ S;\ \Gamma;\ \sigma \Longrightarrow P;\ \{x\colon t \triangleq s\} \cup S;\ \Gamma \cup \Gamma';\ \sigma,$

if none of the previous rules is applicable, i.e., one of the following conditions hold:

(a) both terms have distinct heads: $\mathrm{Top}(t) \neq \mathrm{Top}(s)$, or

(b) both terms are suspensions: $t = \pi_1 \cdot y_1$ and $s = \pi_2 \cdot y_2$, where π_1, π_2 and y_1, y_2 are not necessarily distinct, or

(c) both are abstractions and rule Abs is not applicable: $t = \mathsf{a}.t'$, $s = \mathsf{b}.s'$ and there is no atom $\mathsf{c} \in \mathcal{A}$ satisfying $\nabla \vdash \mathsf{c}\#\mathsf{a}.t'$ and $\nabla \vdash \mathsf{c}\#\mathsf{b}.s'$.

The set Γ' is defined as $\Gamma' := \{\mathsf{a}\#x \mid \mathsf{a} \in \mathcal{A} \ \wedge \ \nabla \vdash \mathsf{a}\#t \ \wedge \ \nabla \vdash \mathsf{a}\#s\}$.

Mer: **Merging**

$P;\ \{x\colon t_1 \triangleq s_1, z\colon t_2 \triangleq s_2\} \uplus S;\ \Gamma;\ \sigma \Longrightarrow$
$P;\ \{x\colon t_1 \triangleq s_1\} \cup S;\ \Gamma\{z \mapsto \pi\cdot x\};\ \sigma\{z \mapsto \pi\cdot x\}$,

where π is an $\mathcal{A}(\{t_1, s_1, t_2, s_2\})$-based permutation such that $\nabla \vdash \pi\bullet t_1 =_\alpha t_2$, and $\nabla \vdash \pi\bullet s_1 =_\alpha s_2$.

Given a finite set of atoms $\mathcal{A}$, two nominal $\mathcal{A}$-based terms t and s, and an $\mathcal{A}$-based freshness context ∇, to compute $\mathcal{A}$-based generalizations for $\langle \nabla, t\rangle$ and $\langle \nabla, s\rangle$, we start with $\{x\colon t \triangleq s\};\ \varnothing;\ \varnothing;\ Id$, where x is a fresh variable, and apply the rules as long as possible. A *Derivation* is a sequence of state transformations by the rules. The state to which no rule applies has the form $\varnothing;\ S;\ \Gamma;\ \sigma$, where Mer does not apply to S. We call it the *final state.* Since we prove uniqueness (modulo $\simeq$) of the final state of exhaustive transformations (see Theorem 3.12), we can denote it by $\mathfrak{G}_{\mathsf{N}}(\mathcal{A}, \nabla, x\colon t \triangleq s)$, i.e., $\varnothing;\ S; \Gamma; \sigma = \mathfrak{G}_{\mathsf{N}}(\mathcal{A}, \nabla, x\colon t \triangleq s)$, and we say that the *result computed* by $\mathfrak{G}_{\mathsf{N}}$ is $\langle \Gamma, x\sigma\rangle$. The store S contains all the differences of the input terms so that $x\sigma\sigma_{\text{L}}(S) =_\alpha t$ and $x\sigma\sigma_{\text{R}}(S) =_\alpha s$, where the substitutions $\sigma_{\text{L}}(S)$ and $\sigma_{\text{R}}(S)$ are defined by:

Definition 3.20. *We define two substitutions obtained from a set S of AUEs:*

$$\sigma_{L}(S) ::= \{y \mapsto t_1 \mid y\colon t_1 \triangleq t_2 \in S\}$$
$$\sigma_{R}(S) ::= \{y \mapsto t_2 \mid y\colon t_1 \triangleq t_2 \in S\}$$

Note that the Dec rule works also for the AUEs of the form $x\colon \mathsf{a} \triangleq \mathsf{a}$. In the Abs rule, it is important to have the corresponding c in $\mathcal{A}$. If we take $\mathcal{A} = \mathcal{A}_2$ in Example 3.8, then Abs can transform the AUE between t and s there, but if $\mathcal{A} = \mathcal{A}_1$ in the same example, then Abs is not applicable. In this case the Sol rule takes over, because the condition (c) of this rule is satisfied.

The condition (b) of Sol helps to compute, e,g, $\langle \varnothing, x\rangle$ for identical terms-in-context $\langle \varnothing, (\mathsf{a}\,\mathsf{b})\cdot y\rangle$ and $\langle \varnothing, (\mathsf{a}\,\mathsf{b})\cdot y\rangle$. Although one might expect that computing $\langle \varnothing, (\mathsf{a}\,\mathsf{b})\cdot y\rangle$ would be more natural, from the generalization point of view it does not matter, because $\langle \varnothing, x\rangle$ is as general as $\langle \varnothing, (\mathsf{a}\,\mathsf{b})\cdot y\rangle$.

Merging requires to solve the so called equivariance problem, that is, decide whether there is a permutation π so that $\nabla \vdash \pi\bullet t =_\alpha s$ for some t, s, and ∇. In subsection 3.1.5 we show an algorithm which solves this problem[†].

3.1.4 Illustration of the Algorithm $\mathfrak{G}_{\mathsf{N}}$

Before introducing the algorithm that constructively solves the equivariance decision problem, we illustrate its usage with the help of some examples.

[†]Notice that we can compose $t = f(t_1, s_1)$ and $s = f(t_2, s_2)$ to satisfy the condition in Mer.

Example 3.9. *Let* $t = f(\mathsf{a},\mathsf{b})$, $s = f(\mathsf{b},\mathsf{c})$, $\nabla = \varnothing$, *and* $\mathcal{A} = \{\mathsf{a},\mathsf{b},\mathsf{c},\mathsf{d}\}$. *Then* $\mathfrak{G}_{\mathsf{N}}$ *performs the following transformations:*

$$\begin{aligned}
& \{x\colon f(\mathsf{a},\mathsf{b}) \triangleq f(\mathsf{b},\mathsf{c})\};\ \varnothing;\ \varnothing;\ \mathit{Id}\\
\Longrightarrow_{\mathsf{Dec}}\quad & \{y\colon \mathsf{a} \triangleq \mathsf{b}, z\colon \mathsf{b} \triangleq \mathsf{c}\};\ \varnothing;\ \varnothing;\ \{x \mapsto f(y,z)\}\\
\Longrightarrow^{2}_{\mathsf{Sol}}\quad & \varnothing;\ \{y\colon \mathsf{a} \triangleq \mathsf{b}, z\colon \mathsf{b} \triangleq \mathsf{c}\};\ \{\mathsf{c}\#y, \mathsf{d}\#y, \mathsf{a}\#z, \mathsf{d}\#z\};\ \{x \mapsto f(y,z)\}\\
\Longrightarrow_{\mathsf{Mer}}\quad & \varnothing;\ \{y\colon \mathsf{a} \triangleq \mathsf{b}\};\ \{\mathsf{c}\#y, \mathsf{d}\#y\};\ \{x \mapsto f(y,(\mathsf{a}\,\mathsf{b})(\mathsf{b}\,\mathsf{c})\cdot y)\}
\end{aligned}$$

Hence, $p = \langle\{\mathsf{c}\#y, \mathsf{d}\#y\}, f(y,(\mathsf{a}\,\mathsf{b})(\mathsf{b}\,\mathsf{c})\cdot y)\rangle$ *is the computed result. It generalizes the input pairs:* $p \preceq \langle\nabla, t\rangle$ *and* $p \preceq \langle\nabla, s\rangle$. *From the final store* $S = \{y\colon \mathsf{a} \triangleq \mathsf{b}\}$ *we get the substitutions* $\sigma_{L}(S) = \{y \mapsto \mathsf{a}\}$ *and* $\sigma_{R}(S) = \{y \mapsto \mathsf{b}\}$ *so that* $f(y,(\mathsf{a}\,\mathsf{b})(\mathsf{b}\,\mathsf{c})\cdot y)\sigma_{L}(S) =_{\alpha} t$ *and* $f(y,(\mathsf{a}\,\mathsf{b})(\mathsf{b}\,\mathsf{c})\cdot y)\sigma_{R}(S) =_{\alpha} s$. *Note that* $\langle\{\mathsf{c}\#y\}, f(y,(\mathsf{a}\,\mathsf{b})(\mathsf{b}\,\mathsf{c})\cdot y)\rangle$ *would also be an* $\mathcal{A}$*-based generalization of* $\langle\nabla, t\rangle$ *and* $\langle\nabla, s\rangle$, *but it is strictly more general than* p.

Example 3.10. *We give three more examples to illustrate how* $\mathfrak{G}_{\mathsf{N}}$ *computes* $\mathcal{A}$*-based lggs for two* $\mathcal{A}$*-based terms and an* $\mathcal{A}$*-based freshness context:*

- *Let* $t = f(\mathsf{b},\mathsf{a})$, $s = f(y,(\mathsf{a}\,\mathsf{b})\cdot y)$, $\nabla = \{\mathsf{b}\#y\}$, *and* $\mathcal{A} = \{\mathsf{a},\mathsf{b}\}$. *Then* $\mathfrak{G}_{\mathsf{N}}$ *computes the term-in-context* $\langle\varnothing, f(z,(\mathsf{a}\,\mathsf{b})\cdot z)\rangle$. *It generalizes the input pairs.*
- *Let* $t = f(g(x),x)$, $s = f(g(y),y)$, $\nabla = \varnothing$, *and* $\mathcal{A} = \varnothing$. *It is a first-order anti-unification problem.* $\mathfrak{G}_{\mathsf{N}}$ *computes* $\langle\varnothing, f(g(z),z)\rangle$. *It generalizes the input pairs.*
- *Let* $t = f(\mathsf{a}.\mathsf{b}, x)$, $s = f(\mathsf{b}.\mathsf{a}, y)$, $\nabla = \{\mathsf{c}\#x\}$, $\mathcal{A} = \{\mathsf{a},\mathsf{b},\mathsf{c},\mathsf{d}\}$. *Then* $\mathfrak{G}_{\mathsf{N}}$ *computes the term-in-context* $p = \langle\{\mathsf{c}\#z_1, \mathsf{d}\#z_1\}, f(\mathsf{c}.z_1, z_2)\rangle$. *It generalizes the input pairs:* $p \preceq \langle\nabla, t\rangle$ *and* $p \preceq \langle\nabla, s\rangle$. *From the store we get* $\sigma_{L}(S) = \{z_1 \mapsto \mathsf{b}, z_2 \mapsto x\}$ *and* $\sigma_{R}(S) = \{z_1 \mapsto \mathsf{a}, z_2 \mapsto y\}$ *so that* $p\sigma_{L}(S) = \langle\varnothing, f(\mathsf{c}.\mathsf{b}, x)\rangle$ *and* $p\sigma_{L}(S) = \langle\varnothing, f(\mathsf{c}.\mathsf{a}, y)\rangle$. *We have* $t =_{\alpha} f(\mathsf{c}.\mathsf{b}, x)$ *and* $s =_{\alpha} f(\mathsf{c}.\mathsf{a}, y)$.

3.1.5 Deciding Equivariance: The Algorithm $\mathcal{E}$

Computation of π in the condition of the rule Mer above requires an algorithm that solves the following problem:

Given: Nominal terms t, s and a freshness context ∇.

Find: An $\mathcal{A}(\{t,s\})$-based permutation π such that $\nabla \vdash \pi{\bullet}t =_{\alpha} s$.

This is the problem of deciding whether t and s are equivariant with respect to ∇. In this Section we describe a rule-based algorithm that solves this problem, called $\mathcal{E}$. Note that our problem differs from the problem of equivariant unification considered in [25]: We do not solve unification problems, since we do not allow variable substitution. We only look for permutations to *decide equivariance constructively* and provide a dedicated algorithm for that.

The algorithm $\mathcal{E}$ works on tuples of the form E; ∇; $\mathcal{A}$; π (also called states) where

- E is a set of equivariance equations of the form $t \sim s$ where t, s are nominal terms,

- ∇ is a freshness context,
- $\mathcal{A}$ is a finite set of available atoms,
- π is a permutation (computed so far).

The algorithm is split into two phases. The first one is a simplification phase where function applications, abstractions and suspensions are decomposed as long as possible. The second phase is the permutation computation, where given a set of equivariance equations between atoms of the form $a \sim b$ we compute the permutation which will be returned in case of success. The rules of the *first phase* are the following:

Dec-E: **Decomposition**

$\{f(t_1, \ldots, t_m) \sim f(s_1, \ldots, s_m)\} \uplus E;\ \nabla;\ \mathcal{A};\ Id \Longrightarrow \{t_1 \sim s_1, \ldots, t_m \sim s_m\} \cup E;\ \nabla;\ \mathcal{A};\ Id.$

Alp-E: **Alpha Equivalence**

$\{\mathsf{a}.t \sim \mathsf{b}.s\} \uplus E;\ \nabla;\ \mathcal{A};\ Id \Longrightarrow \{(\acute{\mathsf{c}}\ \mathsf{a}) \bullet t \sim (\acute{\mathsf{c}}\ \mathsf{b}) \bullet s\} \cup E;\ \nabla;\ \mathcal{A};\ Id,$

where $\acute{\mathsf{c}}$ is a fresh atom of the same sort as a and b.

Sus-E: **Suspension**

$\{\pi_1 \cdot x \sim \pi_2 \cdot x\} \uplus E;\ \nabla;\ \mathcal{A};\ Id \Longrightarrow \{\pi_1 \bullet a \sim \pi_2 \bullet \mathsf{a} \mid \mathsf{a} \in \mathcal{A} \wedge \mathsf{a}\#x \notin \nabla\} \cup E;\ \nabla;\ \mathcal{A};\ Id.$

The rules of the *second phase* are the following:

Rem-E: **Remove**

$\{\mathsf{a} \sim \mathsf{b}\} \uplus E;\ \nabla;\ \mathcal{A};\ \pi \Longrightarrow E;\ \nabla;\ \mathcal{A}\backslash\{\mathsf{b}\};\ \pi, \qquad \text{if } \pi \bullet \mathsf{a} = \mathsf{b}.$

Sol-E: **Solve**

$\{\mathsf{a} \sim \mathsf{b}\} \uplus E;\ \nabla;\ \mathcal{A};\ \pi \Longrightarrow E;\ \nabla;\ \mathcal{A}\backslash\{\mathsf{b}\};\ (\pi \bullet \mathsf{a}\ \ \mathsf{b})\pi, \qquad \text{if } \pi \bullet \mathsf{a}, \mathsf{b} \in \mathcal{A} \text{ and } \pi \bullet \mathsf{a} \neq \mathsf{b}.$

Note that in Alp-E, $\acute{\mathsf{c}}$ is fresh means that $\acute{\mathsf{c}} \notin \mathcal{A}$ and, therefore, $\acute{\mathsf{c}}$ will not appear in π. These atoms are an auxiliary means which play a role during the computation but do not appear in the final result.

Given nominal terms t, s, and a freshness context ∇. We construct the *initial state* $\{t \sim s\};\ \nabla;\ \mathcal{A}(\{t, s\});\ Id$ and apply the above rules successively. We will prove that when the rules transform a state into $\varnothing;\ \nabla;\ \mathcal{A};\ \pi$, then π is an $\mathcal{A}(\{t, s\})$-based permutation such that $\nabla \vdash \pi \bullet t =_\alpha s$. We call this state the *success state*. When no rule is applicable to a state $E';\ \nabla';\ \mathcal{A}';\ \pi'$, and the set of equations in E' is not empty, we will also prove that there is no solution. This state is called the *failure state*. We say that $\mathcal{E}$ *returns* (or computes) *the result* $\perp$ in the case of failure and π if we reach the success state.

3.1.6 Illustration of the Algorithm $\mathcal{E}$

Before discussing the properties of $\mathcal{E}$, we illustrate its usage on a couple of examples.

Example 3.11. *To decide the equivariance problem of* $E = \{\mathsf{a} \sim \mathsf{a},\ \mathsf{a}.(\mathsf{a}\ \mathsf{b})(\mathsf{c}\ \mathsf{d}) \cdot x \sim \mathsf{b}.x\}$ *with respect to the freshness context* $\nabla = \{\mathsf{a}\#x\}$ *by the algorithm* $\mathcal{E}$*, we create the initial*

state $\{\mathsf{a} \sim \mathsf{a}, \mathsf{a}.(\mathsf{a}\,\mathsf{b})(\mathsf{c}\,\mathsf{d})\cdot x \sim \mathsf{b}.x\}$; $\{\mathsf{a}\#x\}$; $\{\mathsf{a},\mathsf{b},\mathsf{c},\mathsf{d}\}$; *Id and apply the transformation rules exhaustively:*

$$
\begin{array}{ll}
 & \{\mathsf{a} \sim \mathsf{a}, \mathsf{a}.(\mathsf{a}\,\mathsf{b})(\mathsf{c}\,\mathsf{d})\cdot x \sim \mathsf{b}.x\};\ \{\mathsf{a}\#x\};\ \{\mathsf{a},\mathsf{b},\mathsf{c},\mathsf{d}\};\ \mathit{Id} \\
\Longrightarrow_{\mathit{Alp\text{-}E}} & \{\mathsf{a} \sim \mathsf{a}, (\acute{\mathsf{e}}\ \mathsf{a})(\mathsf{a}\,\mathsf{b})(\mathsf{c}\,\mathsf{d})\cdot x \sim (\acute{\mathsf{e}}\ \mathsf{b})\cdot x\};\ \{\mathsf{a}\#x\};\ \{\mathsf{a},\mathsf{b},\mathsf{c},\mathsf{d}\};\ \mathit{Id} \\
\Longrightarrow_{\mathit{Sus\text{-}E}} & \{\mathsf{a} \sim \mathsf{a}, \acute{\mathsf{e}} \sim \acute{\mathsf{e}}, \mathsf{c} \sim \mathsf{d}, \mathsf{d} \sim \mathsf{c}\};\ \{\mathsf{a}\#x\};\ \{\mathsf{a},\mathsf{b},\mathsf{c},\mathsf{d}\};\ \mathit{Id} \\
\Longrightarrow_{\mathit{Rem\text{-}E}} & \{\acute{\mathsf{e}} \sim \acute{\mathsf{e}}, \mathsf{c} \sim \mathsf{d}, \mathsf{d} \sim \mathsf{c}\};\ \{\mathsf{a}\#x\};\ \{\mathsf{b},\mathsf{c},\mathsf{d}\};\ \mathit{Id} \\
\Longrightarrow_{\mathit{Rem\text{-}E}} & \{\mathsf{c} \sim \mathsf{d}, \mathsf{d} \sim \mathsf{c}\};\ \{\mathsf{a}\#x\};\ \{\mathsf{b},\mathsf{c},\mathsf{d}\};\ \mathit{Id} \\
\Longrightarrow_{\mathit{Sol\text{-}E}} & \{\mathsf{d} \sim \mathsf{c}\};\ \{\mathsf{a}\#x\};\ \{\mathsf{b},\mathsf{c}\};\ (\mathsf{c}\,\mathsf{d}) \\
\Longrightarrow_{\mathit{Rem\text{-}E}} & \varnothing;\ \{\mathsf{a}\#x\};\ \{\mathsf{b}\};\ (\mathsf{c}\,\mathsf{d}).
\end{array}
$$

Hence, the permutation $(\mathsf{c}\,\mathsf{d})$ *is the result computed by* $\mathcal{E}$*.*

Example 3.12. *We show the results of applying* $\mathcal{E}$ *on some more examples:*

- *For* $E = \{\mathsf{a}.f(\mathsf{b}, x) \sim \mathsf{b}.f(\mathsf{a}, x)\}$ *and* $\nabla = \{\mathsf{a}\#x\}$*,* $\mathcal{E}$ *returns* $\bot$*.*
- *For* $E = \{a.f(\mathsf{b}, (\mathsf{a}\,\mathsf{b})\cdot x) \sim \mathsf{b}.f(\mathsf{a}, x)\}$ *and* $\nabla = \{\mathsf{a}\#x\}$*,* $\mathcal{E}$ *returns* $(\mathsf{b}\,\mathsf{a})$*.*
- *For* $E = \{\mathsf{a}.\mathsf{b}.(\mathsf{a}\,\mathsf{b})(\mathsf{a}\,\mathsf{c})\cdot x = \mathsf{b}.\mathsf{a}.(\mathsf{a}\,\mathsf{c})\cdot x\}$ *and* $\nabla = \varnothing$*,* $\mathcal{E}$ *returns Id.*
- *For* $E = \{\mathsf{a}.\mathsf{b}.(\mathsf{a}\,\mathsf{b})(\mathsf{a}\,\mathsf{c})\cdot x = \mathsf{a}.\mathsf{b}.(\mathsf{b}\,\mathsf{c})\cdot x\}$ *and* $\nabla = \varnothing$*,* $\mathcal{E}$ *returns* $\bot$*.*

3.1.7 Properties of the Algorithm $\mathcal{E}$

Theorem 3.5 (Termination)**.** *The procedure* $\mathcal{E}$ *terminates on any input.*

Proof. We define the complexity measure of a quadruple E; ∇; $\mathcal{A}$; π as a tuple of multisets $(M_1(E), M_2(E))$, where $M_1(E)$ is the number of function symbols, plus the number of abstractions, plus the number of suspensions, and $M_2(E)$ is the sum of the lengths of the terms in the problem. The measures are compared by the well-founded lexicographic ordering. Each rule strictly reduces the complexity. Notice that Alp-E removes two abstractions but may increasse the length of some (already existing) suspensions. □

The Soundness Theorem for $\mathcal{E}$ states that, indeed, the permutation the algorithm computes shows that the input terms are equivariant:

Theorem 3.6 (Soundness)**.** *Let* $\{t \sim s\}$; ∇; $\mathcal{A}$; *Id* $\Longrightarrow^*$ $\varnothing$; ∇; $\mathcal{A}'$; π *be a derivation in* $\mathcal{E}$*, then* π *is an* $\mathcal{A}$*-based permutation such that* $\nabla \vdash \pi\bullet t =_\alpha s$*.*

Proof. We assume the success state with π being the computed permutation. Since Sol-E is the only rule which adds a new swapping to π and the swapped atoms are required to be from $\mathcal{A}$, π is $\mathcal{A}$-based.

The proof is by induction on the length of the derivation, and then, by case analysis on the applied rule. Let Γ be the freshness environment containing all statements $\acute{\mathsf{c}}\#x$ form by a fresh atom $\acute{\mathsf{c}}$ introduced along all the derivation and a variable x of the initial equation.

For any transformation step E; ∇; $\mathcal{A}$; $\pi \Longrightarrow E'$; ∇; $\mathcal{A}'$; π' we will prove that if $\nabla \cup \Gamma \vdash \pi'\bullet t'_i =_\alpha s'_i$, for any $t'_i \sim s'_i \in E'$, then $\nabla \cup \Gamma \vdash \pi\bullet t_i =_\alpha s_i$ for any $t_i \sim s_i \in E$, for any possible applied rule. By induction, we will have $\nabla \cup \Gamma \vdash \pi\bullet t =_\alpha s$ for the

initial equivariance equation $t \sim s$. Since Γ is not relevant to prove $t =_\alpha s$, we have also $\nabla \vdash \pi{\bullet}t =_\alpha s$.

Soundness of Dec-E: From $\nabla \vdash \pi{\bullet}t_1 =_\alpha s_1, \ldots, \nabla \vdash \pi{\bullet}t_n =_\alpha s_n$, follows directly, by the theory of alpha-equivalence $\nabla \vdash f(\pi{\bullet}t_1, \cdots, \pi{\bullet}t_n) =_\alpha f(s_1, \cdots, s_n)$ and by the rule of swapping application $\pi{\bullet}f(t_1, \cdots, t_n) = f(\pi{\bullet}t_1, \cdots, \pi{\bullet}t_n)$, that $\nabla \vdash \pi{\bullet}f(t_1, \cdots, t_n) =_\alpha f(s_1, \cdots, s_n)$. In this case the permutation, the set of atoms and the freshness context are not transformed by the rule.

Soundness of Alp-E: Let ∇ be a freshness context containing Γ, in particular $\acute{\mathsf{c}}\#x$ for any variable $x \in \mathcal{V}(t, s)$. Assume $\nabla \vdash \pi(\mathsf{a}\,\acute{\mathsf{c}}){\bullet}t =_\alpha (\mathsf{b}\,\acute{\mathsf{c}}){\bullet}s$ by induction hypothesis. From this, using $=_\alpha$-abs-1 and the fact that π does not affect to $\acute{\mathsf{c}}$, we can deduce $\nabla \vdash \pi{\bullet}\mathsf{c}.(\mathsf{a}\,\acute{\mathsf{c}}){\bullet}t =_\alpha \mathsf{c}.(\mathsf{b}\,\acute{\mathsf{c}}){\bullet}s$. We can also construct a proof for $\nabla \vdash \acute{\mathsf{c}}\#t$ and $\nabla \vdash \acute{\mathsf{c}}\#s$. Therefore, using $=_\alpha$-abs-2, we can deduce $\nabla \vdash \acute{\mathsf{c}}.(\mathsf{a}\,\acute{\mathsf{c}}){\bullet}t =_\alpha \mathsf{a}.t$ and $\nabla \vdash \acute{\mathsf{c}}.(\mathsf{b}\,\acute{\mathsf{c}}){\bullet}s =_\alpha \mathsf{b}.s$. Now using the lemmas about the transitivity of $=_\alpha$ and additivity of permutation application:

$$\textit{If } \nabla \vdash t =_\alpha s \textit{ and } \nabla \vdash s =_\alpha u \textit{ then } \nabla \vdash t =_\alpha u$$
$$\textit{If } \nabla \vdash t =_\alpha s \textit{ then } \nabla \vdash \pi{\bullet}t =_\alpha \pi{\bullet}s$$

we can deduce $\nabla \vdash \pi{\bullet}(\mathsf{a}.t) =_\alpha \mathsf{b}.s$. This proof proves the soundness of Alp-E. Notice that π does not change in this rule.

Soundness of Sus-E: By induction hypothesis, assume $\nabla \vdash \pi\,\pi_1{\bullet}\mathsf{a} =_\alpha \pi_2{\bullet}\mathsf{a}$, for any atom a such that $\mathsf{a} \in \mathcal{A}$ and $\mathsf{a}\#x \notin \nabla$. Assume also $\Gamma \subset \nabla$, hence, for all fresh atoms, we have $\acute{\mathsf{c}}\#x \in \nabla$. The rest of atoms b are not fresh and satisfy $\mathsf{b} \notin \mathcal{A}$ and $\mathsf{b}\#x \notin \nabla$. Since π_1 and π_2 only affect to atoms from $\mathcal{A}$ or fresh[†], and π is $\mathcal{A}$-based, we have $\pi\,\pi_1{\bullet}\mathsf{b} = \pi_2{\bullet}\mathsf{b} = \mathsf{b}$. Therefore, $\nabla \vdash \pi\,\pi_1{\bullet}\mathsf{a} =_\alpha \pi_2{\bullet}\mathsf{a}$, for any atom $\mathsf{a}\#x \notin \nabla$, and by $=_\alpha$-susp we deduce $\nabla \vdash \pi\,\pi_1{\cdot}x =_\alpha \pi_2{\cdot}x$.

In the second phase we have to take into account that, in all derivations of the form $E;\ \nabla;\ \mathcal{A};\ \pi \Longrightarrow^* \varnothing;\ \nabla;\ \mathcal{A}';\ \pi'\pi$, permutation π' only affects to atoms from $\mathcal{A}$. This can be proved by inspection of the rules.

Soundness of Rem-E: Let be the complete derivation as follows

$$\{\mathsf{a} \sim \mathsf{b}\} \cup E;\ \nabla;\ \mathcal{A};\ \pi \Longrightarrow$$
$$E;\ \nabla;\ \mathcal{A}\backslash\{\mathsf{b}\};\ \pi \Longrightarrow^*$$
$$\varnothing;\ \nabla;\ \mathcal{A}';\ \pi'\pi$$

By induction hypothesis, $\pi'\pi$ solves E. Since the rule has been applied we also have $\pi{\bullet}\mathsf{a} = \mathsf{b}$. Now, the property above proves $\pi'{\bullet}\mathsf{b} = \mathsf{b}$, since $\mathsf{b} \notin \mathcal{A}\backslash\{\mathsf{b}\}$. Therefore $\pi'\pi{\bullet}\mathsf{a} = \mathsf{b}$.

Soundness of Sol-E: let the derivation be:

$$\{\mathsf{a} \sim \mathsf{b}\} \cup E;\ \nabla;\ \mathcal{A};\ \pi \Longrightarrow$$
$$E;\ \nabla;\ \mathcal{A}\backslash\{\mathsf{b}\};\ (\pi{\bullet}\mathsf{a}\ \mathsf{b})\pi \Longrightarrow^*$$
$$\varnothing;\ \nabla;\ \mathcal{A}';\ \pi'(\pi{\bullet}\mathsf{a}\ \mathsf{b})\pi$$

By induction hypothesis, $\pi'(\pi{\bullet}\mathsf{a}\ \mathsf{b})\pi$ solves E. Since $\mathsf{b} \notin \mathcal{A}\backslash\{\mathsf{b}\}$, we have $\pi'{\bullet}\mathsf{b} = \mathsf{b}$. Hence $\pi'(\pi{\bullet}\mathsf{a}\ \mathsf{b})\pi{\bullet}a = \pi'{\bullet}\mathsf{b} = \mathsf{b}$, and the computed permutation also solves the equivariance equation $\mathsf{a} \sim \mathsf{b}$. □

[†]Notice that π_1 and π_2 can only contain swappings of the original equation (i.e. $\mathcal{A}$-based) and swappings introduced by Alp-E.

We now prove an invariant lemma that is needed in order to prove Theorem 3.8.

Lemma 3.7 (Invariant Lemma). *Let $\mathcal{A}$ be a finite set of atoms, E_1 be a set of equivariance equations for terms based on $\mathcal{A}$, π_1 be an $\mathcal{A}$-based permutation and $\mathcal{A}_1 \subseteq \mathcal{A}$. Let $E_1; \nabla; \mathcal{A}_1; \pi_1 \Longrightarrow E_2; \nabla; \mathcal{A}_2; \pi_2$ be any step performed by a rule in $\mathcal{E}$. Let $\Gamma = \{\acute{c}\#x \mid x \in \mathcal{V}(E_1), \acute{c}$ is a fresh variable$\}$. Let μ be an $\mathcal{A}$-based permutation such that $\nabla \cup \Gamma \vdash \mu\bullet t =_\alpha s$, for all $t \sim s \in E_1$. Then*

1. *$\nabla \cup \Gamma \vdash \mu\bullet t' =_\alpha s'$, for all $t' \sim s' \in E_2$.*
2. *If $\mu^{-1}\bullet \mathsf{b} = \pi_1^{-1}\bullet \mathsf{b}$, for all $\mathsf{b} \in \mathcal{A}\backslash\mathcal{A}_1$, then $\mu^{-1}\bullet \mathsf{b} = \pi_2^{-1}\bullet \mathsf{b}$, for all $\mathsf{b} \in \mathcal{A}\backslash\mathcal{A}_2$.*

Proof. By case distinction on the applied rule.

Dec-E: The proposition is obvious.

Alp-E: In this case it follows from the definitions of $=_\alpha$ and permutation application.

Sus-E: In this case $t = \tau_1 \cdot x$, $s = \tau_2 \cdot x$, and by the assumption we have $\nabla \vdash \mu\tau_1 \cdot x =_\alpha \tau_2 \cdot x$. By the definition of $=_\alpha$, it means that we have $\mathsf{a}\#x \in \nabla$, for all atoms a such that $\mu\tau_1\bullet \mathsf{a} \neq \tau_2\bullet \mathsf{a}$. Hence, for all $\mathsf{a} \in \mathcal{A}$ with $\mathsf{a}\#x \notin \nabla$ we have $\nabla \vdash \mu\tau_1\bullet \mathsf{a} =_\alpha \tau_2\bullet \mathsf{a}$. This implies that μ also solves the equations in E_2, hence item 1 of the lemma.

Item 2 of the lemma is trivial for these three rules, since $\mathcal{A}_1 = \mathcal{A}_2$ and $\pi_1 = \pi_2 = Id$.

Rem-E: The item 1 is trivial. To prove the item 2, note that $t = \mathsf{a}$, $s = \mathsf{b}$, $\pi_1 = \pi_2$ and we only need to show $\mu^{-1}\bullet \mathsf{b} = \pi_2^{-1}\bullet \mathsf{b}$. By the assumption we have $\nabla \vdash \mu\bullet \mathsf{a} =_\alpha \mathsf{b}$. Since a and b are atoms, the latter simply means that $\mu\bullet \mathsf{a} = \mathsf{b}$. From the rule condition we also know that $\pi_1\bullet \mathsf{a} = \mathsf{b}$. From these two equalities we get $\mu^{-1}\bullet \mathsf{b} = \mathsf{a} = \pi_2^{-1}\mathsf{b}$.

Sol-E: The item 1 is trivial also in this case. To prove the item 2, note that $t = \mathsf{a}$, $s = \mathsf{b}$, $\pi_2 = (\pi_1\bullet \mathsf{a}\ \mathsf{b})\pi_1$ and we only need to show $\mu^{-1}\bullet \mathsf{b} = \pi_2^{-1}\bullet \mathsf{b}$. By the assumption we have $\nabla \vdash \mu\bullet \mathsf{a} =_\alpha \mathsf{b}$, which means that $\mu\bullet \mathsf{a} = \mathsf{b}$ and, hence, $\mathsf{a} = \mu^{-1}\bullet \mathsf{b}$. As for $\pi_2^{-1}\bullet \mathsf{b}$, we have $\pi_2^{-1}\bullet \mathsf{b} = \pi_1^{-1}(\pi_1\bullet \mathsf{a}\ \mathsf{b})\bullet \mathsf{b} = \pi_1^{-1}\bullet(\pi_1\bullet \mathsf{a}) = \mathsf{a}$. Hence, we get $\mu^{-1}\bullet \mathsf{b} = \mathsf{a} = \pi_2^{-1}\bullet \mathsf{b}$. □

Theorem 3.8 (Completeness). *Let $\mathcal{A}$ be a finite set of atoms, t, s be $\mathcal{A}$-based terms, and ∇ be a freshness context. If $\nabla \vdash \mu\bullet t =_\alpha s$ holds for some $\mathcal{A}$-based permutation μ, then there exists a derivation $\{t \sim s\}; \nabla; \mathcal{A}; Id \Longrightarrow^* \varnothing; \Gamma; \mathcal{A}'; \pi$, obtained by an execution of $\mathcal{E}$, such that $\pi\bullet \mathsf{a} = \mu\bullet \mathsf{a}$ for any atom $\mathsf{a} \in \mathrm{FA}(t)$.*

Proof. First show that under the conditions of the theorem, if $\{t \sim s\}; \nabla; \mathcal{A}; Id \Longrightarrow^* \varnothing; \Gamma; \mathcal{A}''; \pi$ is a derivation obtained by an execution of $\mathcal{M}$, then $\pi\bullet \mathsf{a} = \mu\bullet \mathsf{a}$, for any atom $\mathsf{a} \in \mathrm{FA}(t)$. Afterwards we prove that (under the conditions of the theorem) there is no failing derivation with the rules of $\mathcal{M}$ starting from $\{t \sim s\}; \nabla; \mathcal{A}; Id$. Since all derivations are finite, it will imply the existence of $\{t \sim s\}; \nabla; \mathcal{A}; Id \Longrightarrow^* \varnothing; \Gamma; \mathcal{A}''; \pi$.

Let $\{t \sim s\}; \nabla; \mathcal{A}; Id \Longrightarrow^* E'; \Gamma'; \mathcal{A}'; \pi' \Longrightarrow^* \varnothing; \Gamma; \mathcal{A}''; \pi$ be a derivation, where $E'; \Gamma'; \mathcal{A}'; \pi'$ is the first state in the second phase of the algorithm. It means that E' contains equations between atoms only, and the atoms of t (except, maybe, some bound ones which disappear after the application of the Alp-E rule) appear in the left hand sides of equations in E. By Lemma 3.7, $\Gamma' \vdash \mu\bullet \mathsf{a}_1 =_\alpha \mathsf{a}_2$, for all $\mathsf{a}_1 \sim \mathsf{a}_2 \in E'$. By Theorem 3.6 and Lemma 3.7 the same is true for π. Therefore, $\Gamma' \vdash \mu\bullet \mathsf{a}_1 =_\alpha \pi\bullet \mathsf{a}_1$, for all $\mathsf{a}_1 \sim \mathsf{a}_2 \in E'$. For atoms, $\nabla \vdash \mathsf{a} =_\alpha \mathsf{b}$ iff $\mathsf{a} = \mathsf{b}$. Hence, we get $\mu\bullet \mathsf{a}_1 = \pi\bullet \mathsf{a}_1$, for all $\mathsf{a}_1 \in S$, where $\mathrm{FA}(t) \subseteq S \subseteq \mathcal{A}(t)$. It proves $\pi\bullet \mathsf{a} = \mu\bullet \mathsf{a}$, for all $\mathsf{a} \in \mathrm{FA}(t)$, when the desired successful derivation exists.

Now we show that no derivation with the rules of $\mathcal{M}$ starting from $\{t \sim s\}$; ∇; $\mathcal{A}$; Id fails. Assume by contradiction that there exists such a failing derivation. Let E'; ∇'; $\mathcal{A}'$; π' be the final state in it, to which no rule applies. Analyzing the rules in $\mathcal{M}$, one can easily conclude that it can be caused by one of the following two cases:

1. E' contains an equivariance equation of the form $f(t_1, \ldots, t_n) \sim g(s_1, \ldots, s_m)$, where $f \neq g$.
2. E' contains an equivariance equation of the form $\mathsf{a} \sim \mathsf{b}$, where $\pi'\bullet\mathsf{a} \neq \mathsf{b}$, such that $\pi'\bullet\mathsf{a} \notin \mathcal{A}'$ or $\mathsf{b} \notin \mathcal{A}'$.

In the first case, by Lemma 3.7 $\nabla \vdash \mu\bullet f(t_1, \ldots, t_n) =_\alpha g(s_1, \ldots, s_m)$ should hold, but $f \neq g$ forbids it. Hence, this case is impossible.

Now we analyze the second case. Consider each condition.

Condition 1: $\pi'\bullet\mathsf{a} \notin \mathcal{A}'$. Then either $\pi'\bullet\mathsf{a}$ is a fresh atom, or $\pi'\bullet\mathsf{a} \in \mathcal{A}\backslash\mathcal{A}'$.

- $\pi'\bullet\mathsf{a}$ *is a fresh atom:* Since π' does not affect fresh atoms, we get $\mathsf{a} \neq \mathsf{b}$. On the other hand, we have $\Gamma' \vdash \mu\bullet\mathsf{a} =_\alpha \mathsf{b}$ and, hence, $\mu\bullet\mathsf{a} = \mathsf{b}$, because $\mu\bullet\mathsf{a}$ and b are atoms. Since μ is $\mathcal{A}$-based, $\mathsf{b} \notin \mathcal{A}$ implies $\mathsf{a} = \mathsf{b}$. A contradiction.
- $\pi'\bullet\mathsf{a} \in \mathcal{A}\backslash\mathcal{A}'$*:* By Lemma 3.7 we get $\mu^{-1}\pi'\bullet\mathsf{a} = \pi'^{-1}\pi'\bullet\mathsf{a} = \mathsf{a}$. Therefore, $\pi'\bullet\mathsf{a} = \mu\bullet\mathsf{a}$ and we get $\mu\bullet\mathsf{a} \neq \mathsf{b}$, which contradicts $\Gamma' \vdash \mu\bullet\mathsf{a} =_\alpha \mathsf{b}$, because $\mu\bullet\mathsf{a}$ and b are atoms.

Condition 2: $\mathsf{b} \notin \mathcal{A}'$. Then either b is a fresh atom, or $\mathsf{b} \in \mathcal{A}\backslash\mathcal{A}'$.

- b *is a fresh atom:* We obtain a contradiction by a reasoning similar to the case when $\pi'\bullet\mathsf{a}$ is a fresh atom.
- $\mathsf{b} \in \mathcal{A}\backslash\mathcal{A}'$*:* The atom b has been removed from the set of atoms in the derivation earlier either at Sus-E, Rem-E, or Sol-E step, which indicates that there is $\mathsf{c} \in \mathcal{A}\backslash\mathcal{A}'$ such that $\mathsf{c} = \pi'^{-1}\bullet\mathsf{b}$. Moreover, $\mathsf{c} \neq \mathsf{a}$. From Lemma 3.7 we get $\mathsf{c} = \mu^{-1}\bullet\mathsf{b}$ which, together with $\mathsf{c} \neq \mathsf{a}$, implies $\mu\bullet\mathsf{a} \neq \mathsf{b}$. But it contradicts $\Gamma' \vdash \mu\bullet\mathsf{a} =_\alpha \mathsf{b}$.

The obtained contradiction proves that no derivation with the rules of $\mathcal{M}$ starting from $\{t \sim s\}$; ∇; $\mathcal{A}$; Id fails. □

3.1.8 Properties of the Algorithm $\mathfrak{G}_\mathsf{N}$

Theorem 3.9 (Termination)**.** *The procedure* $\mathfrak{G}_\mathsf{N}$ *terminates on any input (provided that the computation of* π *in the Merge rule terminates).*

Proof. We associate to each state P; S; Γ; σ its measure, a triple $(n, M(P), M(S))$, where n is a number of abstractions in P, and $M(U)$ is a multiset defined for a set of AUEs U as follows:

$$M(U) := \{|s| + |t| \mid x : t \triangleq s \in U\}.$$

Measures are compared lexicographically. Obviously, each rule in $\mathfrak{G}_\mathsf{N}$ strictly reduces it. The ordering is well-founded. In the conditions of the rules, proving atomic freshness formulas from freshness contexts terminates. Computation of π in the Merge rule terminates by Theorem 3.5. Hence, $\mathfrak{G}_\mathsf{N}$ terminates. □

The Soundness Theorem states that the result computed by $\mathfrak{G}_{\mathsf{N}}$ is indeed an $\mathcal{A}$-based generalization of the input terms-in-context:

Theorem 3.10 (Soundness)**.** *Given terms t and s and a freshness context ∇, all based on a finite set of atoms $\mathcal{A}$, if $\{x\colon t \triangleq s\}$; $\varnothing$; $\varnothing$; $\varepsilon \Longrightarrow^{+} \varnothing$; S; Γ; σ is a derivation obtained by an execution of $\mathfrak{G}_{\mathsf{N}}$, then $\langle \Gamma, x\sigma\rangle$ is an $\mathcal{A}$-based generalization of $\langle \nabla, t\rangle$ and $\langle \nabla, s\rangle$.*

Proof. Since all atoms introduced by the rules of $\mathfrak{G}_{\mathsf{N}}$ in Γ and σ are from $\mathcal{A}$, $\langle \Gamma, x\sigma\rangle$ is $\mathcal{A}$-based. To prove that $\langle \Gamma, x\sigma\rangle$ generalizes both $\langle \nabla, t\rangle$ and $\langle \nabla, s\rangle$, we use well-founded induction on the length of derivations. In fact, we will prove a more general statement:

Assume P_0; S_0; Γ_0; $\vartheta_0 \Longrightarrow^{+} \varnothing$; S_n; Γ_n; $\vartheta_0\vartheta_1\cdots\vartheta_n$ is a derivation in $\mathfrak{G}_{\mathsf{N}}$ (with ∇ and $\mathcal{A}$) with the following property: If $z_0 : t_0 \triangleq s_0 \in S_0$, then $\mathsf{a}\#z_0 \in \Gamma_0$ for an $\mathsf{a} \in \mathcal{A}$ iff $\nabla \vdash \mathsf{a}\#t_0$ and $\nabla \vdash \mathsf{a}\#s_0$. Notice that requiring this property does not imply a lose of generality: our algorithm starts with no equation in the store, and each time an equation is moved to the store the Sol rule adds the required freshness constraints (by inspection of Sol). Moreover, freshness constraints are only removed from the freshness context when Mer removes the corresponding equation from the store (by inspection of Mer). Then for any $z_0 : t_0 \triangleq s_0 \in P_0 \cup S_0$ we have $\langle \Gamma_n\backslash\Gamma_0, z_0\vartheta_1\cdots\vartheta_n\rangle \preceq \langle \nabla, t_0\rangle$ and $\langle \Gamma_n\backslash\Gamma_0, z_0\vartheta_1\cdots\vartheta_n\rangle \preceq \langle \nabla, s_0\rangle$.

Assume the statement is true for any derivation of the length $l < n$ and prove it for a derivation P_0; S_0; Γ_0; $\vartheta_0 \Longrightarrow^{+} \varnothing$; S_n; Γ_n; $\vartheta_0\vartheta_1\cdots\vartheta_n$ of the length n. Below the composition $\vartheta_i\vartheta_{i+1}\cdots\vartheta_k$ is abbreviated as ϑ_i^k with $k \geqslant i$.

Let $z_0 : t_0 \triangleq s_0$ be an AUE selected for transformation from $P_0 \cup S_0$. We consider each rule:

Dec: $z_0 = x$, $t_0 = \hbar(t_1, \ldots, t_m)$, $s_0 = \hbar(s_1, \ldots, s_m)$, $\Gamma_1 = \Gamma_0$, and $\vartheta_1 = \{x \mapsto \hbar(y_1, \ldots, y_m)\}$. By the induction hypothesis (IH), $\langle \Gamma_n\backslash\Gamma_1, y_i\vartheta_2^n\rangle \preceq \langle \nabla, t_i\rangle$ and $\langle \Gamma_n\backslash\Gamma_1, y_i\vartheta_2^n\rangle \preceq \langle \nabla, s_i\rangle$ for all $1 \leqslant i \leqslant m$. Hence by definition of $\preceq$ for terms-in-context, there exist substitutions σ and φ such that:

- $\vartheta_2^n\sigma$ and $\vartheta_2^n\varphi$ respect $\Gamma_n\backslash\Gamma_1$,
- $(\Gamma_n\backslash\Gamma_1)\sigma \subseteq \nabla$ and $(\Gamma_n\backslash\Gamma_1)\varphi \subseteq \nabla$, and
- $\nabla \vdash y_i\vartheta_2^n\sigma =_\alpha t_i$ and $\nabla \vdash y_i\vartheta_2^n\varphi =_\alpha s_i$ for all $1 \leqslant i \leqslant m$.

Finally, since $\vartheta_1 = \{x \mapsto \hbar(y_1, \ldots, y_m)\}$ and $\Gamma_n\backslash\Gamma_0 = \Gamma_n\backslash\Gamma_1$, we obtain $\langle \Gamma_n\backslash\Gamma_0, z_0\vartheta_1^n\rangle \preceq \langle \nabla, t_0\rangle$ and $\langle \Gamma_n\backslash\Gamma_0, z_0\vartheta_1^n\rangle \preceq \langle \nabla, s_0\rangle$.

Abs: $z_0 = x$, $t_0 = \mathsf{a}.t$, $s_0 = \mathsf{b}.s$, $\Gamma_1 = \Gamma_0$, and $\vartheta_1 = \{x \mapsto \mathsf{c}.y\}$, where $\nabla \vdash \mathsf{c}\#\mathsf{a}.t$ and $\nabla \vdash \mathsf{c}\#\mathsf{b}.s$. P_1 contains the AUE $y\colon (\mathsf{c}\,\mathsf{a})\bullet t \triangleq (\mathsf{c}\,\mathsf{b})\bullet s$. By the IH, $\langle \Gamma_n\backslash\Gamma_1, y_i\vartheta_2^n\rangle \preceq \langle \nabla, (\mathsf{c}\,\mathsf{a})\bullet t\rangle$ and $\langle \Gamma_n\backslash\Gamma_1, y_i\vartheta_2^n\rangle \preceq \langle \nabla, (\mathsf{c}\,\mathsf{b})\bullet s\rangle$ hence $\nabla \vdash y\vartheta_2^n\sigma =_\alpha (\mathsf{c}\,\mathsf{a})\bullet t$ and $\nabla \vdash y\vartheta_2^n\varphi =_\alpha (\mathsf{c}\,\mathsf{b})\bullet s$ for some σ and φ that in addition satisfy the other properties (as above) for $\preceq$. Then, since we also have that $\nabla \vdash \mathsf{c}\#\mathsf{a}.t$ and $\nabla \vdash \mathsf{c}\#\mathsf{b}.s$ we can prove, with the $=_\alpha$-abs rules, that $\nabla \vdash \mathsf{c}.y\vartheta_2^n\sigma =_\alpha \mathsf{a}.t$ and $\nabla \vdash \mathsf{c}.y\vartheta_2^n\varphi =_\alpha \mathsf{b}.s$. Finally, since $\vartheta_1 = \{x \mapsto \mathsf{c}.y\}$ and $\Gamma_n\backslash\Gamma_0 = \Gamma_n\backslash\Gamma_1$, we obtain $\langle \Gamma_n\backslash\Gamma_0, z_0\vartheta_1^n\rangle \preceq \langle \nabla, t_0\rangle$ and $\langle \Gamma_n\backslash\Gamma_0, z_0\vartheta_1^n\rangle \preceq \langle \nabla, s_0\rangle$.

Sol: $z_0 = x$, $t_0 = t$, $s_0 = s$, $\Gamma_1\backslash\Gamma_0 = \{\mathsf{a}\#x \mid \mathsf{a} \in \mathcal{A}, \nabla \vdash \mathsf{a}\#t, \text{ and } \nabla \vdash \mathsf{a}\#s\}$ and $\vartheta_1 = \varepsilon$. By the IH we have that $(\Gamma_n\backslash\Gamma_1)\sigma \subseteq \nabla$, $(\Gamma_n\backslash\Gamma_1)\varphi \subseteq \nabla$, $\nabla \vdash x\vartheta_2^n\sigma =_\alpha t$, and $\nabla \vdash x\vartheta_2^n\varphi =_\alpha s$ for some σ and φ respecting $\Gamma_n\backslash\Gamma_1$.

Since $\vartheta_1 = \varepsilon$, from the IH we get $\nabla \vdash x\vartheta_1^n\sigma =_\alpha t$. To show that $(\Gamma_n\backslash\Gamma_0)\sigma \subseteq \nabla$ take $\mathsf{a}\#y \in \Gamma_n\backslash\Gamma_0$ for some a.

- If $\mathsf{a}\#y \in \Gamma_n\backslash\Gamma_1$, then $\{\mathsf{a}\#y\}\sigma \subseteq \nabla$ by the IH,
- otherwise, if $\mathsf{a}\#y \notin \Gamma_n\backslash\Gamma_1$, then $\mathsf{a}\#y \in \Gamma_1\backslash\Gamma_0$ with $x = y$ and $x\vartheta_2^n = x$. By the IH, $\nabla \vdash x\sigma =_\alpha t$, besides, we know $\nabla \vdash \mathsf{a}\#t$. Therefore, we know $\nabla \vdash \mathsf{a}\#x\sigma$, which by Theorem 3.1 implies $\{\mathsf{a}\#x\}\sigma = \{\mathsf{a}\#y\}\sigma \subseteq \nabla$. Thus, $(\Gamma_n\backslash\Gamma_0)\sigma \subseteq \nabla$.

Hence, we proved $\langle\Gamma_n\backslash\Gamma_0, x\vartheta_1^n\rangle \preceq \langle\nabla, t\rangle$, which is the same as $\langle\Gamma_n\backslash\Gamma_0, z_0\vartheta_1^n\rangle \preceq \langle\nabla, t_0\rangle$. $\langle\Gamma_n\backslash\Gamma_0, z_0\vartheta_1^n\rangle \preceq \langle\nabla, s_0\rangle$ can be proved analogously.

Mer: First, we show that the following holds: For all $k \geqslant 0$, If $z_k \colon t_k \triangleq s_k \in S_k$ and $\mathsf{c}\#z_k \in \Gamma_k$ for a $\mathsf{c} \in \mathcal{A}$, then $\nabla \vdash \mathsf{c}\#t_k$ and $\nabla \vdash \mathsf{c}\#s_k$.

Proceed by induction on k. If $k = 0$, then it follows from the assumption on P_0; S_0; Γ_0; ϑ_0. Assume it is true for k and show it for $k+1$. Take $z_{k+1} \colon t_{k+1} \triangleq s_{k+1} \in S_{k+1}$. We have two alternatives: Either $z_{k+1} \colon t_{k+1} \triangleq s_{k+1}$ has been a subject of the Mer rule at this step, or not. If not, then either it was already in S_k or was introduced at this step. In either case, by IH or because it has been introduced with Sol rule, if $\mathsf{c}\#z_k \in \Gamma_k$ for a $\mathsf{c} \in \mathcal{A}$, then $\nabla \vdash \mathsf{c}\#t_k$ and $\nabla \vdash \mathsf{c}\#s_k$. If $z_{k+1} \colon t_{k+1} \triangleq s_{k+1}$ was a subject of the Mer rule, then there exists some $u_k \colon r_k \triangleq q_k \in S_k$, such that $\nabla \vdash \pi_k{\bullet}t_{k+1} =_\alpha r_k$, $\nabla \vdash \pi_k{\bullet}s_{k+1} =_\alpha q_k$. Moreover, for all $\mathsf{d}\#u_k \in S_k$ we now have $\pi_k^{-1}{\bullet}\mathsf{d}\#z_{k+1} \in S_{k+1}$, and all $\mathsf{c}\#z_{k+1} \in S_k$ are retained in S_{k+1}. For these c's, since $z_{k+1} \colon t_{k+1} \triangleq s_{k+1} \in S_k$, by the induction hypothesis we have $\nabla \vdash \mathsf{c}\#t_{k+1}$ and $\nabla \vdash \mathsf{c}\#s_{k+1}$. As for $\pi_k^{-1}{\bullet}\mathsf{d}\#z_{k+1} \in S_{k+1}$, here we need to show $\nabla \vdash \pi_k^{-1}{\bullet}\mathsf{d}\#t_{k+1}$ and $\nabla \vdash \pi_k^{-1}{\bullet}\mathsf{d}\#s_{k+1}$. By the induction hypothesis we know $\nabla \vdash \mathsf{d}\#r_k$. Then $\nabla \vdash \pi_k^{-1}{\bullet}\mathsf{d}\#\pi_k^{-1}{\bullet}r_k$ and since $\nabla \vdash \pi_k{\bullet}t_{k+1} =_\alpha r_k$, we get $\nabla \vdash \pi_k^{-1}{\bullet}\mathsf{d}\#t_{k+1}$. $\nabla \vdash \pi_k^{-1}{\bullet}\mathsf{d}\#s_{k+1}$ can be shown similarly, using $\nabla \vdash \mathsf{d}\#q_k$.

Now we turn to proving the Mer case itself. In this case, there exist $x \colon t_1 \triangleq s_1 \in S_0$, $y \colon t_2 \triangleq s_2 \in S_0$, and π such that $\nabla \vdash \pi{\bullet}t_1 =_\alpha t_2$ and $\nabla \vdash \pi{\bullet}s_1 =_\alpha s_2$. Moreover, by the construction of the derivation, $x \colon t_1 \triangleq s_1$ is either retained in S_n, or is removed from there because there exist an AUE $z \colon t_n \triangleq s_n \in S_n$ and a permutation ρ such that $\nabla \vdash \rho{\bullet}t_n =_\alpha t_1$, $\nabla \vdash \rho{\bullet}t_n =_\alpha s_1$, and $x\vartheta_1^n = \rho{\cdot}z$. We can turn these two cases into one, permitting $z = x$, $t_n = t_1$, $s_n = s_1$, and $\rho = Id$ to cover also the first case.

Therefore, we can say that there exists a AUE $z \colon t_n \triangleq s_n \in S_n$ such that for some permutation ρ, $x\vartheta_1^n = x\vartheta_2^n = \rho{\cdot}z$, $y\vartheta_1^n = \pi{\bullet}x\vartheta_2^n = \pi\rho{\cdot}z$, $\Gamma_n\backslash\Gamma_0 = \{(\pi\rho)^{-1}\mathsf{a}\#z \mid \mathsf{a}\#y \in \Gamma_0\}$, $\nabla \vdash \pi\rho{\bullet}t_n =_\alpha t_2$, $\nabla \vdash \pi\rho{\bullet}s_n =_\alpha s_2$, $\nabla \vdash \rho{\bullet}t_n =_\alpha t_1$, and $\nabla \vdash \rho{\bullet}s_n =_\alpha s_1$.

We want to prove $\langle\Gamma_n\backslash\Gamma_0, z_0\vartheta_1^n\rangle \preceq \langle\nabla, t_0\rangle$. First, we take σ such that $z\sigma = t_n$ and show $(\Gamma_n\backslash\Gamma_0)\sigma \subseteq \nabla$. For this, we try to prove $\{\mathsf{b}\#u\}\sigma \subseteq \nabla$ for all $\mathsf{b}\#u \in \Gamma_n\backslash\Gamma_0$. By the IH, we have $\{\mathsf{b}\#u\}\sigma \subseteq \nabla$ for all $\mathsf{b}\#u \in \Gamma_n\backslash\Gamma_1$. Note that $z\sigma = t_n$ does not restrict generality, because if $u = z$, then by the proposition we proved at the beginning of the Mer case we have that $\mathsf{b}\#u \in \Gamma_n\backslash\Gamma_1$ implies $\nabla \vdash \mathsf{b}\#t_n$.

Therefore, $\{\mathsf{b}\#u\}\sigma = \mathcal{C}t\chi(\{\mathsf{b}\#z\sigma\}) = \mathcal{C}t\chi(\{\mathsf{b}\#t_n\})$ and by Theorem 3.1 we indeed have $\{\mathsf{b}\#u\}\sigma \subseteq \nabla$. Now assume $\mathsf{b}\#u \in (\Gamma_n\backslash\Gamma_0)\backslash(\Gamma_n\backslash\Gamma_1)$. Then $\mathsf{b}\#u \in \Gamma_n \cap (\Gamma_1\backslash\Gamma_0)$. That means, $\mathsf{b}\#u = \pi^{-1}{\bullet}\mathsf{a}\#x$, where $\mathsf{a}\#y \in \Gamma_0$. Moreover, the AUE $x \colon t_1 \triangleq s_1$ has

been retained in S_n. From the latter we have, in fact, $z = x$, $t_n = t_1$, and $s_n = s_1$. Then $\{\mathsf{b}\#u\}\sigma = \mathcal{C}t\mathcal{X}(\{\pi^{-1}\bullet\mathsf{a}\#x\sigma \mid \mathsf{a}\#y \in \Gamma_0\}) = \mathcal{C}t\mathcal{X}(\{\mathsf{a}\#\pi\bullet t_1 \mid \mathsf{a}\#y \in \Gamma_0\})$. On the other hand, from the assumption on P_0; S_0; Γ_0; ϑ_0 we know that for all $\mathsf{a}\#y \in \Gamma_0$ we have $\nabla \vdash \mathsf{a}\#t_2$, from which by $\nabla \vdash \pi\bullet t_1 =_\alpha t_2$ we get $\nabla \vdash \mathsf{a}\#\pi\bullet t_1$. Hence, we can apply Theorem 3.1 to $\mathcal{C}t\mathcal{X}(\{\mathsf{a}\#\pi\bullet t_1 \mid \mathsf{a}\#y \in \Gamma_0\})$, obtaining $\{\mathsf{b}\#u\}\sigma \subseteq \nabla$ also in this case. Hence, $(\Gamma_n \backslash \Gamma_0)\sigma \subseteq \nabla$.

It remains to prove $\nabla \vdash z_0\vartheta_1^n\sigma =_\alpha t_0$. First, assume $z_0 = x$, $t_0 = t_1$, $s_0 = s_1$. Then we have $\nabla \vdash z_0\vartheta_2^n\sigma =_\alpha t_0$, because $z_0\vartheta_2^n\sigma = \rho\bullet z\sigma = \rho\bullet t_n$ and we know that $\nabla \vdash \rho\bullet t_n =_\alpha t_1$. Since $z_0\vartheta_2^n = z_0\vartheta_1^n$, we get $\nabla \vdash z_0\vartheta_1^n\sigma =_\alpha t_0$. Hence, we proved $\langle \Gamma_n \backslash \Gamma_0, z_0\vartheta_1^n\rangle \preceq \langle \nabla, t_0\rangle$ for this case. $\langle \Gamma_n \backslash \Gamma_0, z_0\vartheta_1^n\rangle \preceq \langle \nabla, s_0\rangle$ can be proved similarly.

Now let $z_0 = y$, $t_0 = t_2$, $s_0 = s_2$ and prove again $\nabla \vdash z_0\vartheta_1^n\sigma =_\alpha t_0$. Then $z_0\vartheta_1^n\sigma = \pi\rho\bullet z\sigma = \pi\rho\bullet t_n$. But we have already seen that $\nabla \vdash \pi\rho\bullet t_n =_\alpha t_2$. Hence, $\nabla \vdash z_0\vartheta_1^n\sigma =_\alpha t_0$ is proved. It implies $\langle \Gamma_n \backslash \Gamma_0, z_0\vartheta_1^n\rangle \preceq \langle \nabla, t_0\rangle$ for this case.

$\langle \Gamma_n \backslash \Gamma_0, z_0\vartheta_1^n\rangle \preceq \langle \nabla, s_0\rangle$ can be proved similarly. □

The Completeness Theorem states that for any given $\mathcal{A}$-based generalization of two input terms-in-context, $\mathfrak{G}_\mathsf{N}$ can compute one which is at most as general than the given one.

Theorem 3.11 (Completeness)**.** *Given terms t and s and freshness contexts ∇ and Γ, all based on a finite set of atoms $\mathcal{A}$. If $\langle \Gamma, r\rangle$ is an $\mathcal{A}$-based generalization of $\langle \nabla, t\rangle$ and $\langle \nabla, s\rangle$, then there exists a derivation $\{x\colon t \triangleq s\}$; $\varnothing$; $\varnothing$; $\varepsilon \Longrightarrow^+ \varnothing$; S; Γ'; σ obtained by an execution of $\mathfrak{G}_\mathsf{N}$, such that $\langle \Gamma, r\rangle \preceq \langle \Gamma', x\sigma\rangle$.*

Proof. By structural induction on r. We can assume without loss of generality that $\langle \Gamma, r\rangle$ is an lgg of $\langle \nabla, t\rangle$ and $\langle \nabla, s\rangle$.

Let r be an atom a. Then $t = s = \mathsf{a}$. Therefore, the Dec rule gives $\langle \varnothing, \mathsf{a}\rangle$ as the computed answer. To show that $\langle \Gamma, \mathsf{a}\rangle \preceq \langle \varnothing, \mathsf{a}\rangle$, it is enough to take a substitution σ such that $x\sigma \neq \mathsf{b}$ for each $\mathsf{b}\#x \in \Gamma$. Note that it is not necessary $\mathsf{b} \in \mathcal{A}$.

Let r be an abstraction $\mathsf{c}.r'$. Then $t = \mathsf{a}.t'$, $s = \mathsf{b}.s'$, $\mathsf{c} \in \mathcal{A}$, $\nabla \vdash \mathsf{c}\#t$, $\nabla \vdash \mathsf{c}\#s$, and $\langle \Gamma, r'\rangle$ is an $\mathcal{A}$-based generalization of $\langle \nabla, t'\rangle$ and $\langle \nabla, s'\rangle$. In this case, the Abs rule can be applied, which gives $\{y\colon (\mathsf{c}\,\mathsf{a})\bullet t' \triangleq (\mathsf{c}\,\mathsf{b})\bullet s'\}$; $\varnothing$; $\varnothing$; σ_1, where $\sigma_1 = \{x \mapsto \mathsf{c}.y\}$. By the induction hypothesis, we can compute Γ' and σ_2 such that $\langle \Gamma, r'\rangle \preceq \langle \Gamma', y\sigma_2\rangle$. Let $\sigma = \sigma_1\sigma_2$. We get $\langle \Gamma, r\rangle = \langle \Gamma, \mathsf{c}.r'\rangle \preceq \langle \Gamma', \mathsf{c}.y\sigma_2\rangle = \langle \Gamma', x\sigma\rangle$.

Let r be a suspension $\pi\cdot z$. Since $\langle \Gamma, r\rangle$ is an lgg of $\langle \nabla, t\rangle$ and $\langle \nabla, s\rangle$, the context Γ contains all constraints $\pi^{-1}\bullet\mathsf{a}\#z$ such that $\nabla \vdash \mathsf{a}\#t$ and $\nabla \vdash \mathsf{a}\#s$, and the following alternatives are possible:

(a) t and s have distinct heads: $\mathrm{Top}(t) \neq \mathrm{Top}(s)$, or

(b) t and s are both suspensions: $t = \pi_1\cdot y_1$ and $s = \pi_2\cdot y_2$, where π_1, π_2 and y_1, y_2 are not necessarily distinct, or

(c) t and s are abstractions, but $\mathcal{A}$ does not contain an appropriate fresh atom to uniformly rename the bound atoms in t and s.

These alternatives give exactly the conditions of the Sol rule. Hence, we can apply it, getting $\varnothing$; $\{x\colon t \triangleq s\}$; Γ'; σ, where $\Gamma' = \{\mathsf{a}\#x \mid \mathsf{a} \in \mathcal{A} \wedge \nabla \vdash \mathsf{a}\#t \wedge \nabla \vdash \mathsf{a}\#s\}$ and $\sigma = \varepsilon$. Then $\langle \Gamma, r\rangle \preceq \langle \Gamma', x\sigma\rangle$, which can be confirmed by the substitution $\{z \mapsto \pi^{-1}\cdot x\}$.

Let r be a term $f(r_1, \ldots, r_n)$. Then $t = f(t_1, \ldots, t_n)$, $s = f(s_1, \ldots, s_n)$, and $\langle \Gamma, r_i \rangle$ is a generalization of $\langle \nabla, t_i \rangle$ and $\langle \nabla, s_i \rangle$. We proceed by the Dec rule, obtaining $\{y_i \colon t_i \triangleq s_i \mid 1 \leqslant i \leqslant n\}$; $\varnothing$; $\varnothing$; $\{x \mapsto f(y_1, \ldots, y_n)\}$. By the induction hypothesis, we can construct derivations $D_1, \ldots, D_n$ computing the substitutions $\sigma_1, \ldots, \sigma_n$, respectively, such that $\langle \Gamma, r_i \rangle \preceq \langle \Gamma'_i, y_i\sigma_i \rangle$ for $1 \leqslant i \leqslant n$. We combine these derivations, together with the initial Dec step, into one derivation of the form $\mathsf{D} = \{x \colon t \triangleq s\}$; S_0; Γ'_0; $\sigma_0 \Longrightarrow \{y_i \colon t_i \triangleq s_i \mid 1 \leqslant i \leqslant n\}$; S_1; Γ'_1; $\sigma_0\sigma_1 \Longrightarrow^* \varnothing$; S_n; Γ'_n; $\sigma_0\sigma_1 \cdots \sigma_n$, where $\Gamma'_0 = \Gamma'_1 = \varnothing$, $\sigma_0 = \varepsilon$, and $\sigma_1 = \{x \mapsto f(y_1, \ldots, y_n)\}$. If r does not contain the same variable more than once, $\langle \Gamma, r_i \rangle \preceq \langle \Gamma'_i, y_i\sigma_i \rangle$ for all $1 \leqslant i \leqslant n$ imply $\langle \Gamma, r \rangle = \langle \Gamma, f(r_1, \ldots, r_n) \rangle \preceq \langle \Gamma', f(y_1, \ldots, y_n) \rangle = \langle \Gamma', x\sigma \rangle$. If r contains the same variable at positions I_1 and I_2 (in subterms of the form $\pi_1 \cdot z$ and $\pi_2 \cdot z$), it indicates that

(a) the path to I_1 is the same (modulo bound atom renaming) in t and s. It equals (modulo bound atom renaming) the path to I_1 in r, and

(b) the path to I_2 is the same (modulo bound atom renaming) in t and s. It equals (modulo bound atom renaming) the path to I_2 in r.

(c) there exists a substitution ϑ_1, which respects Γ, such that $\Gamma \vdash \pi_1 \cdot z\vartheta_1 =_\alpha \tau_1 \bullet t|_{I_1}$ and $\Gamma \vdash \pi_2 \bullet z\vartheta_1 =_\alpha \tau_2 \bullet t|_{I_2}$, where τ_1 and τ_2 are permutations which rename atoms bound in t by fresh ones,

(d) there exists a substitution ϑ_2, which respects Γ, such that $\Gamma \vdash \pi_1 \bullet z\vartheta_2 =_\alpha \rho_1 \bullet s|_{I_1}$ and $\Gamma \vdash \pi_2 \bullet z\vartheta_2 =_\alpha \rho_2 \bullet s|_{I_2}$, where ρ_1 and ρ_2 are permutations which rename atoms bound in s by fresh ones,

Then, because of (a) and (b), we should have two AUEs in S_n: One, between (renamed variants of) $t|_{I_1}$ and $s|_{I_1}$, and the other one between (renamed variants of) $t|_{I_2}$ and $s|_{I_2}$. The possible renaming of bound atoms is caused by the fact that Abs might have been applied to obtain the AUEs. From (c) and (d) we know that $\tau_1, \tau_2, \rho_1, \rho_2$ are the names of those renaming permutations. Let those AUEs be $z_1 \colon \tau_1 \bullet t|_{I_1} \triangleq \rho_1 \bullet s|_{I_1}$ and $z_2 \colon \tau_2 \bullet t|_{I_2} \triangleq \rho_2 \bullet s|_{I_2}$.

From (c) we get $\Gamma \vdash z\vartheta_1 =_\alpha \pi_1^{-1}\tau_1 \bullet t|_{I_1}$ and $\Gamma \vdash z\vartheta_1 =_\alpha \pi_2^{-1}\tau_2 \bullet t|_{I_2}$, which imply $\Gamma \vdash \pi_1^{-1}\tau_1 \bullet t|_{I_1} =_\alpha \pi_2^{-1}\tau_2 \bullet t|_{I_2}$ and, finally, $\Gamma \vdash \pi_2\pi_1^{-1} \bullet \tau_1 \bullet t|_{I_1} =_\alpha \tau_2 \bullet t|_{I_2}$. Similarly, from (d) we get $\Gamma \vdash \pi_2\pi_1^{-1}\rho_1 \bullet s|_{I_1} =_\alpha \rho_2 \bullet s|_{I_2}$.

That means, we can make the step with the Mer rule for $z_1 \colon \tau_1 \bullet t|_{I_1} \triangleq \rho_1 \bullet s|_{I_1}$ and $z_2 \colon \tau_2 \bullet t|_{I_2} \triangleq \rho_2 \bullet s|_{I_2}$ with the substitution $\sigma'_1 = \{z_2 \mapsto \pi_2\pi_1^{-1} \cdot z_1\}$. We can repeat this process for all duplicated variables in r, extending D to the derivation $\{x \colon t \triangleq s\}$; S_0; Γ'_0; $\sigma_0 \Longrightarrow \{y_i \colon t_i \triangleq s_i \mid 1 \leqslant i \leqslant n\}$; S_1; Γ'_1; $\sigma_0 \Longrightarrow^* \varnothing$; S_n; Γ'_n; $\sigma_0\sigma_1 \cdots \sigma_n \Longrightarrow^+ \varnothing$; S_{n+m}; Γ'_{n+m}; $\sigma_0\sigma_1 \cdots \sigma_n\sigma'_1 \cdots \sigma'_m$, where $\sigma'_1, \ldots, \sigma'_m$ are substitutions introduced by the applications of the Mer rule. Let $\sigma = \sigma_0\sigma_1 \cdots \sigma_n\sigma'_1 \cdots \sigma'_m$ and $\Gamma' = \Gamma'_{n+m}$. By this construction, we have $\langle \Gamma, r \rangle \preceq \langle \Gamma', x\sigma \rangle$, which finishes the proof. □

Depending on the selection of AUEs to perform a step, there can be different derivations in $\mathfrak{G}_\mathsf{N}$ starting from the same AUE, leading to different generalizations. The next theorem states that all those generalizations are the same modulo variable renaming and α-equivalence.

Theorem 3.12 (Uniqueness Modulo $\simeq$)**.** *Let t and s be terms and ∇ be a freshness context that are based on the same finite set of atoms. Let $\{x \colon t \triangleq s\}$; $\varnothing$; $\varnothing$; $\varepsilon \Longrightarrow^+ \varnothing$;*

S_1; Γ_1; σ_1 and $\{x\colon t \triangleq s\}$; $\varnothing$; $\varnothing$; $\varepsilon \Longrightarrow^{+} \varnothing$; S_2; Γ_2; σ_2 be two maximal derivations in $\mathfrak{G}_{\mathsf{N}}$. Then $\langle \Gamma_1, x\sigma_1\rangle \simeq \langle \Gamma_2, x\sigma_2\rangle$.

Proof. It is not hard to notice that if it is possible to change the order of applications of rules (but sticking to the same selected AUEs for each rule) then the result remains the same (modulo fresh variable and atom names): Let D_1 and D_2 be two two-step derivations $\mathsf{D}_1 = P_1; S_1; \Gamma_1; \sigma_1 \Longrightarrow_{\mathsf{R}_1} P_2; S_2; \Gamma_2; \sigma_1\vartheta_1 \Longrightarrow_{\mathsf{R}_2} P_3; S_3; \Gamma_3; \sigma_1\vartheta_1\vartheta_2$ and $\mathsf{D}_2 = P_1; S_1; \Gamma_1; \sigma_1 \Longrightarrow_{\mathsf{R}_2} P_2'; S_2'; \Gamma_2'; \sigma_1\vartheta_2 \Longrightarrow_{\mathsf{R}_1} P_3'; S_3'; \Gamma_3'; \sigma_1\vartheta_2\vartheta_1$, where R_1 and R_2 are (not necessarily different) rules and each of them transforms *exactly the same AUE(s)* in both D_1 and D_2. Then these AUE(s) are already present in $P_1 \cup S_1$: They are introduced neither by R_1 nor by R_2. Therefore, $\mathrm{Dom}(\vartheta_2) \cap \mathrm{Ran}(\vartheta_1) = \mathrm{Dom}(\vartheta_1) \cap \mathrm{Ran}(\vartheta_2) = \varnothing$. Moreover, if we assume that the fresh variables and atoms introduced by the rules are the same in both derivations, then $P_3 = P_3'$, $S_3 = S_3'$, $\Gamma_3 = \Gamma_3'$, and $\sigma_1\vartheta_1\vartheta_2 = \sigma_1\vartheta_2\vartheta_1$.

Decomposition, Abstraction, and Solving rules transform the selected AUE in a unique way. We show that it is irrelevant in which order we decide equivariance in the Merging rule.

Let $P; \{z\colon t_1 \triangleq s_1, y\colon t_2 \triangleq s_2\} \cup S; \Gamma; \sigma \Longrightarrow P; \{z\colon t_1 \triangleq s_1\} \cup S; \Gamma\{y \mapsto \boldsymbol{\pi}\cdot z\}; \sigma\{y \mapsto \boldsymbol{\pi}\cdot z\}$ be the merging step with $\nabla \vdash \boldsymbol{\pi}\bullet t_1 =_\alpha t_2$ and $\nabla \vdash \boldsymbol{\pi}\bullet s_1 =_\alpha s_2$. If we do it in the other way around, we would get the step $P; \{z\colon t_1 \triangleq s_1, y\colon t_2 \triangleq s_2\} \cup S; \Gamma; \sigma \Longrightarrow P; \{y\colon t_2 \triangleq s_2\} \cup S; \Gamma\{z \mapsto \boldsymbol{\pi}^{-1}\cdot y\}; \sigma\{z \mapsto \boldsymbol{\pi}^{-1}\cdot y\}$.

Let $\vartheta_1 = \sigma\varphi_1$ with $\varphi_1 = \{y \mapsto \boldsymbol{\pi}\cdot z\}$ and $\vartheta_2 = \sigma\varphi_2$ with $\varphi_2 = \{z \mapsto \boldsymbol{\pi}^{-1}\cdot y\}$. Our goal is to prove that $\langle \Gamma\varphi_1, x\vartheta_1\rangle \simeq \langle \Gamma\varphi_2, x\vartheta_2\rangle$. For this, we need to prove both $\langle \Gamma\varphi_1, x\vartheta_1\rangle \preceq \langle \Gamma\varphi_2, x\vartheta_2\rangle$ and $\langle \Gamma\varphi_2, x\vartheta_2\rangle \preceq \langle \Gamma\varphi_1, x\vartheta_1\rangle$.

First, prove $\langle \Gamma\varphi_1, x\vartheta_1\rangle \preceq \langle \Gamma\varphi_2, x\vartheta_2\rangle$. We should find such a φ that $\Gamma\varphi_1\varphi \subseteq \Gamma\varphi_2$ and $\Gamma\varphi_2 \vdash x\vartheta_1\varphi =_\alpha x\vartheta_2$.

Take $\varphi = \varphi_2$. Note that for any term t, we have $\Gamma\varphi_2 \vdash t\varphi_1\varphi_2 =_\alpha t\varphi_2$, because $\varphi_1\varphi_2 = \{z \mapsto \boldsymbol{\pi}\boldsymbol{\pi}^{-1}\cdot y\}$ and we have $\Gamma\varphi_2 \vdash \boldsymbol{\pi}\boldsymbol{\pi}^{-1}\cdot y =_\alpha y$. Therefore, $\Gamma\varphi_2 \vdash x\vartheta_1\varphi =_\alpha x\sigma\varphi_1\varphi_2 =_\alpha x\sigma\varphi_2 =_\alpha x\vartheta_2$ holds.

As for $\Gamma\varphi_1\varphi \subseteq \Gamma\varphi_2$, note that φ_2 respects $\Gamma\varphi_1$, because it replaces a variable with a suspension and the $\mathcal{C}t\mathcal{X}$ algorithm will have to apply only $\mathsf{Sus\text{-}E}$ rule. We introduce notations Γ_u and $\overline{\Gamma}_u$ for any freshness context Γ and a variable u, denoting $\Gamma_u := \{\mathsf{a}\#u \mid \mathsf{a}\#u \in \Gamma\}$ and $\overline{\Gamma}_u := \Gamma\backslash\Gamma_u$. Then $\Gamma\varphi_1 = \overline{\Gamma}_y \cup \Gamma_y\varphi_1$ and $\Gamma\varphi_2 = \overline{\Gamma}_z \cup \Gamma_z\varphi_2$.

Under this notation, $\Gamma\varphi_1\varphi_2 = \overline{\Gamma}_y\varphi_2 \cup \Gamma_y\varphi_1\varphi_2$. Take $\overline{\Gamma}_y\varphi_2$. We have $\overline{\Gamma}_y\varphi_2 = (\overline{\Gamma}_y\backslash(\overline{\Gamma}_y)_z) \cup ((\overline{\Gamma}_y)_z)\varphi_2 = (\overline{\Gamma}_y\backslash\Gamma_z) \cup \Gamma_z\varphi_2$. Since $\Gamma_z \cap \Gamma_z\varphi_2 = \varnothing$, the we obtain $(\overline{\Gamma}_y\backslash\Gamma_z) \cup \Gamma_z\varphi_2 = (\overline{\Gamma}_y \cup \Gamma_z\varphi_2)\backslash\Gamma_z$. As for $\Gamma_y\varphi_1\varphi_2$, it is easy to see that $\Gamma_y\varphi_1\varphi_2 = \Gamma_y$.

Hence, we get $\Gamma\varphi_1\varphi_2 = ((\overline{\Gamma}_y \cup \Gamma_z\varphi_2)\backslash\Gamma_z) \cup \Gamma_y$. Since $\Gamma_z \cap \Gamma_y = \varnothing$, we get $((\overline{\Gamma}_y \cup \Gamma_z\varphi_2)\backslash\Gamma_z) \cup \Gamma_y = ((\overline{\Gamma}_y \cup \Gamma_z\varphi_2) \cup \Gamma_y)\backslash\Gamma_z = (\Gamma \cup \Gamma_z\varphi_2)\backslash\Gamma_z$. Since $\Gamma_z\varphi_2 \cap \Gamma_z = \varnothing$, we get $(\Gamma \cup \Gamma_z\varphi_2)\backslash\Gamma_z = (\Gamma\backslash\Gamma_z) \cup \Gamma_z\varphi_2$. Hence, $\Gamma\varphi_1\varphi_2 = \overline{\Gamma}_z \cup \Gamma_z\varphi_2 = \Gamma\varphi_2$.

We proved $\langle \Gamma\varphi_1, x\vartheta_1\rangle \preceq \langle \Gamma\varphi_2, x\vartheta_2\rangle$.

With a similar reasoning we can show $\langle \Gamma\varphi_2, x\vartheta_2\rangle \preceq \langle \Gamma\varphi_1, x\vartheta_1\rangle$. □

Theorems 3.10, 3.11, and 3.12 imply that nominal anti-unification is unitary: For any $\mathcal{A}$-based ∇, t, and s, there exists an $\mathcal{A}$-based lgg of $\langle \nabla, t\rangle$ and $\langle \nabla, s\rangle$, which is unique modulo $\simeq$ and can be computed by the algorithm $\mathfrak{G}_{\mathsf{N}}$.

We already discussed the relation between the set of atoms $\mathcal{A}$ we consider as the base of an lgg of two $\mathcal{A}$-based terms-in-context. Now we turn to proving Conjecture 3.1.

Lemma 3.13. *Let $\mathcal{A}_1$ and $\mathcal{A}_2$ be two finite sets of atoms with $\mathcal{A}_1 \subseteq \mathcal{A}_2$ such that the $\mathcal{A}_1$-based terms-in-context $\langle \nabla, t\rangle$ and $\langle \nabla, s\rangle$ have an $\mathcal{A}_1$-based lgg $\langle \Gamma_1, r_1\rangle$ and an $\mathcal{A}_2$-based lgg $\langle \Gamma_2, r_2\rangle$. Then $\Gamma_2 \vdash r_1 \preceq r_2$.*

Proof. $\langle \Gamma_1, r_1\rangle$ and $\langle \Gamma_2, r_2\rangle$ are unique modulo $\simeq$. Let D_i be the derivation in $\mathfrak{G}_\mathsf{N}$ that computes $\langle \Gamma_i, r_i\rangle$, $i = 1, 2$. The number of atoms in $\mathcal{A}_1$ and $\mathcal{A}_2$ makes a difference in the rule Abs: If there are not enough atoms in $\mathcal{A}_1$, an Abs step in D_2 is replaced by a Sol step in D_1. It means that for all positions I of r_1, $r_2|_I$ is also defined. Moreover, there might exist a subterm $r_1|_I$, which has a form of suspension, while $r_2|_I$ is an abstraction. For such positions, $r_1|_I \preceq r_2|_I$. For the other positions J of r_1, $r_1|_J$ and $r_2|_J$ may differ only by names of generalization variables or by names of bound atoms.

Another difference might be in the application of Sol in both derivations: It can happen that this rule produces a larger Γ' in D_2 than in D_1, when transforming the same AUE.

Hence, if there are positions $I_1, \ldots, I_n$ in r_1 such that $r_1|_{I_k} = \pi_k \cdot x$, then there exists a substitution σ_x such that $\Gamma_2 \vdash \pi_i \cdot x\sigma_x =_\alpha r_2|_{I_k}$, $1 \leqslant k \leqslant n$. Taking the union of all σ_x's where $x \in \mathcal{V}(r_1)$, we get σ with the property $\Gamma_2 \vdash r_1\sigma =_\alpha r_2$. □

Recall Example 3.8 which illustrates that we can not replace $\Gamma_2 \vdash r_1 \preceq r_2$ with $\Gamma_2 \vdash r_1 \simeq r_2$ in Lemma 3.13. The following theorem corresponds to Conjecture 3.1:

Theorem 3.14. *Under the conditions of Lemma 3.13, if $\mathcal{A}_1$ is saturated for t, s, ∇, then $\Gamma_2 \vdash r_1 \simeq r_2$.*

Proof. Let D_i be the derivation in $\mathfrak{G}_\mathsf{N}$ that computes $\langle \Gamma_i, r_i\rangle$, $i = 1, 2$. Note that in each of these derivations, the number of Abs steps does not exceed $\min\{\|t\|_{\mathsf{Abs}}, \|s\|_{\mathsf{Abs}}\}$. Since $\mathcal{A}_1$ is saturated for t, s, ∇ and $\mathcal{A}_1 \subseteq \mathcal{A}_2$, $\mathcal{A}_2$ is also saturated for t, s, ∇. Hence, whenever an AUE between two abstractions is encountered in the derivation D_i, there is always $c \in \mathcal{A}_1$ available which satisfies the condition of the Abs rule. Therefore, such AU-E's are never transformed by Sol. We can assume without loss of generality that the sequence of steps in D_1 and D_2 are the same. we may also assume that we take the same fresh variables, and the same atoms from $\mathrm{Fresh}(\mathcal{A}_1, t, s, \nabla)$ in the corresponding steps in D_1 and D_2. Then the only difference between these derivations is in the Γ's, caused by the Sol rule which might eventually make Γ_2 larger than Γ_1. The σ's computed by the derivations are the same and, therefore, r_1 and r_2 are the same (modulo the assumptions on the variable and fresh atom names). Hence, $\Gamma_2 \vdash r_1 \simeq r_2$. □

3.1.9 Complexity Analysis of $\mathcal{E}$ and $\mathfrak{G}_\mathsf{N}$

We represent a permutations π as two hash tables. One for the permutation itself, we call it T_π, and one for the inverse of the permutation, called $T_{\pi^{-1}}$. The key of a hash tables is an atom and we associate another atom, the mapping, with it. For instance the permutation $\pi = (\mathsf{a}\,\mathsf{b})(\mathsf{a}\,\mathsf{c})$ is represented as $T_\pi = \{\mathsf{a} \mapsto \mathsf{c}, \mathsf{b} \mapsto \mathsf{a}, \mathsf{c} \mapsto \mathsf{b}\}$ and $T_{\pi^{-1}} = \{\mathsf{a} \mapsto \mathsf{b}, \mathsf{b} \mapsto \mathsf{c}, \mathsf{c} \mapsto \mathsf{a}\}$. We write $T_\pi(\mathsf{a})$ to obtain from the hash table T_π the atom which is associated with the key a. If no atom is associated with the key a then $T_\pi(\mathsf{a})$ returns a. We write $T_\pi(\mathsf{a} \mapsto \mathsf{b})$, to set the mapping such that $T_\pi(\mathsf{a}) = \mathsf{b}$. As the set of atoms is small, we can assume a perfect hash function. It follows, that both defined operations are done in constant time, leading to constant time application of

a permutation. Swapping application to a permutation $(\mathsf{a}\,\mathsf{b})\pi$ is also done in constant time in the following way: Obtain $c = T_{\pi^{-1}}(\mathsf{a})$ and $\mathsf{d} = T_{\pi^{-1}}(\mathsf{b})$ and perform the following updates:

(a) $T_\pi(\mathsf{c} \mapsto \mathsf{b})$ and $T_\pi(\mathsf{d} \mapsto \mathsf{a})$,

(b) $T_{\pi^{-1}}(\mathsf{b} \mapsto \mathsf{c})$ and $T_{\pi^{-1}}(\mathsf{a} \mapsto \mathsf{d})$.

We also represent set membership of atoms to a set of atoms $\mathcal{A}$ with a hash table $\in_{\mathcal{A}}$ from atoms to Booleans such that $\in_{\mathcal{A}}(\mathsf{a}) = \mathit{true}$ iff $\mathsf{a} \in \mathcal{A}$. We also have a list $L_{\mathcal{A}}$ of the atoms representing the entries of the table such that $\in_{\mathcal{A}}(\mathsf{a}) = \mathit{true}$ to easily know all atoms in $\mathcal{A}$.

Finally we also represent set membership of freshness constraints to a freshness environment ∇ with a hash table $\in_\nabla$.

Theorem 3.15. *Given a set of equivariance equations E, and a freshness context ∇. Let m be the size of ∇, and let n be the size of E. The algorithm $\mathcal{E}$ has $O(n^2 + m)$ time complexity.*

Proof. Collecting the atoms from E in a separate set $\mathcal{A}$ does not affect the space complexity and can be done in time $O(n)$. The freshness environment ∇ will not be modified by rule applications and membership test in the rule Sus-E can be done in constant time. We only have to construct the corresponding hash tables in time $O(m)$. We analyze complexity of both phases.

For the first phase, notice that all rules can be applied only $O(n)$ many times, since Dec-E removes two function symbols and Alp-E two abstraction, and Sus-E two suspensions. The resulting equations after this phase only contain atoms. However, notice that the size of these equations is not necessarily linear. Every time we apply Alp-E a new swapping is applied to both subterms. This swappings may increase the size of suspensions occurring bellow the abstraction. Since there are $O(n)$ many suspensions and $O(n)$ many abstractions, the final size of suspensions is $O(n^2)$. This is the size of the atom equations at the beginning of the second phase. We can see that the application of Dec-E rule has $O(1)$ time complexity (with the appropriate representation of equations).

The application of Alp-E rule requires to find a fresh atom not in $\mathcal{A}$, this can be done in constant time. Later, a swapping has to be applied twice. Swapping application requires traversing the term hence has $O(n)$ time complexity. The application of Sus-E requires to traverse $L_{\mathcal{A}}$ ($O(n)$) and check for freshness membership in $\in_\nabla$ ($O(1)$). Finally it has to add equations like $(\pi_1 \bullet a =_\alpha \pi_2)$, this requires to build T_{π_1} and T_{π_2} that can be done in $O(n)$ time complexity and allow us to build each equation in $O(1)$ time. Summing up, this phase has $O(n^2)$ time complexity.

For the second phase, notice that both rules Rem-E and Sol-E remove an equation and do not introduce any other one. Hence, potentially having $O(n^2)$ many equations in this phase, these equations can be applied $O(n^2)$ may times. We construct a hash table T_π for π that will be maintained and used by both rules. Each application has time complexity $O(1)$. Rem-E uses T_π to check for applicability and if it is applied, it only removes b from $\mathcal{A}$, hence updating $\in_{\mathcal{A}}$ (notice that we do not care about $L_{\mathcal{A}}$ in this second phase of the algorithm). Sol-E uses $\in_{\mathcal{A}}$ and T_π to check for applicability

and if it is applied, it only removes b from $\mathcal{A}$ (hence updating $\in_{\mathcal{A}}$), and updates T_{π}. Summing up, this phase maintains the overall $O(n^2)$ time complexity. □

Theorem 3.16. *The nominal anti-unification algorithm $\mathfrak{G}_{\mathsf{N}}$ has $O(n^5)$ time complexity and $O(n^4)$ space complexity, where n is the input size.*

Proof. By design of the rules and theorem 3.12 we can arrange a maximal derivation like $\{x_0 \colon t_0 \triangleq s_0\}; \varnothing; \varnothing; Id \Longrightarrow^*_{\mathsf{Dec,Abs,Sol}} \varnothing; S_l; \Gamma_l; \sigma_l \Longrightarrow^*_{\mathsf{Mer}} \varnothing; S_m; \Gamma_m; \sigma_m$, postponing the application of Mer until the end. Rules Dec, Abs and Sol can be applied $O(n)$ many times. However, notice that every application of Abs may increase the size of every suspension below. Hence, the size of the store S_l is $O(n^2)$, although it only contain $O(n)$ equations, after an exhaustive derivation $\{x_0 \colon t_0 \triangleq s_0\}; \varnothing; \varnothing; Id \Longrightarrow^*_{\mathsf{Dec,Abs,Sol}} \varnothing; S_l; \Gamma_l; \sigma_l$.

Now we turn to analyzing the transformation phase $\varnothing; S_l; \Gamma_l; \sigma_l \Longrightarrow^*_{\mathsf{Mer}} \varnothing; S_m; \Gamma_m; \sigma_m$. Let $S_l = \{x_1 \colon t_1 \triangleq s_1, \ldots, x_k \colon t_k \triangleq s_k\}$ and n_i be the size of $X_i \colon t_i \triangleq s_i$, $1 \leqslant i \leqslant k$, then $\sum_{i=1}^{k} n_i = O(n^2)$ and $k = O(n)$. From theorem 3.15 we know that solving the equivariance problem for two AUEs $x_i \colon t_i \triangleq s_i$ and $x_j \colon t_j \triangleq s_j$ and an arbitrary freshness context ∇ requires $O((n_i + n_j)^2 + m)$ time and space, where m is the size of ∇ with $m = O(n)$.

Merging requires to solve this problem for each pair of AUEs. This leads to the time complexity $\sum_{i=1}^{k} \sum_{j=i+1}^{k} O((n_i+n_j)^2+m) \leqslant O(\sum_{i=1}^{k} \sum_{j=1}^{k} (n_i+n_j)^2)+O(\sum_{i=1}^{k} \sum_{j=1}^{k} m)$. The second sum is $\sum_{i=1}^{k} \sum_{j=1}^{k} m = k^2 m = O(n^3)$. Now we estimate an upper bound for the sum $\sum_{i=1}^{k} \sum_{j=1}^{k} (n_i + n_j)^2 = \sum_{i=1}^{k} \sum_{j=1}^{k} n_i^2 + \sum_{i=1}^{k} \sum_{j=1}^{k} 2n_i n_j + \sum_{i=1}^{k} \sum_{j=1}^{k} n_j^2 \leqslant \sum_{i=1}^{k} k n_i^2 + 2 \left(\sum_{i=1}^{k} n_i\right) \left(\sum_{j=1}^{k} n_j\right) + \sum_{i=1}^{k} (\sum_{j=1}^{k} n_j)^2 \leqslant k(\sum_{i=1}^{k} n_i)^2 + 2O(n^2)O(n^2) + \sum_{i=1}^{k} O(n^2) = kO(n^2)^2 + 2O(n^2)^2 + kO(n^2)^2 = O(n^5)$, resulting into the stated bounds.

The space is bounded by the space required by a single call to the equivariance algorithm with an imput of size $O(n^2)$, hence $O(n^4)$. □

The problem of anti-unification for nominal terms-in-context is sensitive to the set of atoms permitted in generalizations: If this set is infinite, there is no least general generalization. Otherwise there exists a unique lgg. If this set is finite and satisfies the notion of being saturated, defined in the paper, then the lgg retains the common structure of the input nominal terms maximally.

3.2 Anti-Unification for Lambda Terms

For higher-order terms, in general, there is no unique higher-order lgg. Therefore, special classes have been considered for which the uniqueness is guaranteed (see related work in the introduction). Here, we give an algorithm that computes a unique (modulo $\simeq_{=_\alpha}$) higher-order pattern generalization for two arbitrary lambda terms in the simply-typed calculus. Higher-order pattern are λ-terms where the arguments of free variables are distinct bound variables. They have been introduced by Miller [59] and gained popularity because of their attractive combination of expressive power and good computational behavior. Our anti-unification algorithm is formulated in a rule-based manner. It runs in $O(n^2)$ time and requires $O(n)$ space, where n is the size of the

two input terms. The algorithm can be a useful ingredient for refactoring and clone detection tools for languages that are based on simply-typed lambda calculus. We will illustrate this usage scenario for the programming language λProlog (see, e.g., Nadathur and Miller [63]) in subsection 3.2.3. Example 3.13 shows two λ-terms and their least general higher-order pattern generalization.

Example 3.13. *The term* $t = \lambda z_1.\lambda z_2.g(X(z_1, z_2), X(z_2, z_1))$ *is a higher-order pattern generalization of the two terms* $t_1 = \lambda x_1.\lambda x_2.g(f(x_1, x_2), f(x_2, x_1))$ *and* $t_2 = \lambda y_1.\lambda y_2.g(h(y_2, \lambda y_3.f(y_3, y_1)), h(y_1, \lambda y_4.f(y_4, y_2)))$*. The first term can be obtained from the generalization by replacing the variable* X *with the lambda term* $\lambda x.\lambda y.f(x, y)$ *and the second term by replacing* X *with* $\lambda x.\lambda y.h(y, \lambda z.f(z, x))$*. Figure 3.4 shows the tree representations of the two terms* t_1, t_2*, and their higher-order pattern generalization* t*.*

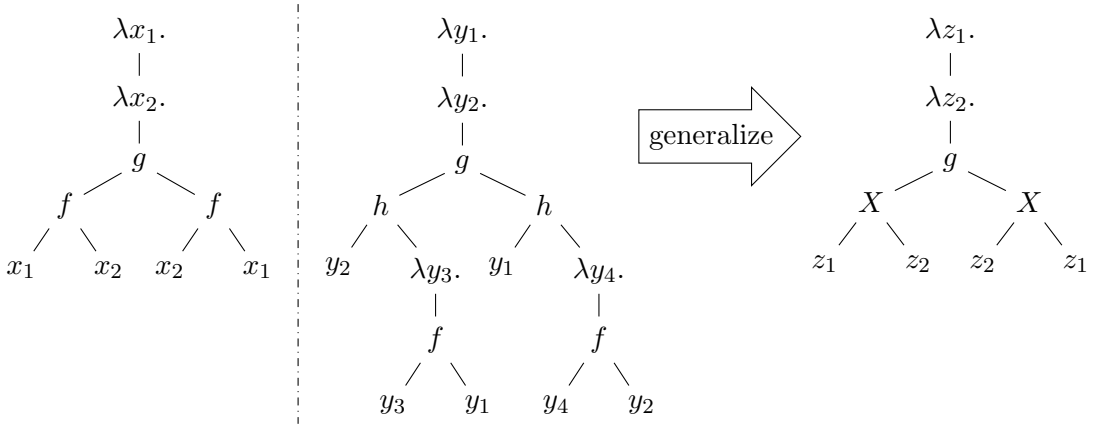

Figure 3.4: Two λ-terms and their higher-order pattern lgg.

3.2.1 Simply-Typed Lambda Terms (Preliminaries)

Characters that denote function symbols, terms, substitutions, etc. are the same as in the previous chapters. Previously used notations, like $\mathcal{V}(t)$, etc., that trivially generalize to the language of simply-typed lambda terms are used without defining them again. Standard notions of the simply-typed λ-calculus, like α-conversion, β-reduction, η-long β-normal form, etc. are defined as usual (see, e.g., Dowek [31]).

Definition 3.21 (Higher-order signature)**.** *In* higher-order signatures *we have a set of* basic types *and a set of* function symbols*. We denote basic types by* δ*. Function symbols have a type given by the grammar:*

$$\tau ::= \delta \mid \tau \to \tau$$

where $\to$ *is associative to the right.*

Definition 3.22 (λ-terms)**.** *Given a countably infinite set of* term variables $\mathcal{V}^{\dagger}$ *and a signature* Σ*.* λ-terms *are built using the grammar*

$$t ::= x \mid c \mid \lambda x.t \mid t_1\ t_2$$

where x *is a variable and* c *is a constant.*

A term t is said to have type τ if either:

- t is a constant of type τ,
- t is a variable of type τ,
- $t = (t_1\ t_2)$ and t_1 has type $\tau' \to \tau$ and t_2 has type τ' for some type τ',
- $t = \lambda x.t_1$, the variable x has type τ', the term t_1 has type τ'' and $\tau = \tau' \to \tau''$.

A term t is said to be *well-typed* if there exists a type τ such that t has type τ. In this case τ is unique and is called the type of t. The set of well-typed terms constructed over Σ and $\mathcal{V}$ is denoted by $\mathcal{T}(\Sigma, \mathcal{V})$, or simply by $\mathcal{T}$ if the concrete instances of Σ and $\mathcal{V}$ are unimportant.

Definition 3.23 (Depth of a term)**.** *The* depth *of a term* t*, denoted* $\mathrm{Depth}(t)$ *is defined recursively as follows:*

$$\begin{aligned}&\mathrm{Depth}(x) = \mathrm{Depth}(c) = 1,\\ &\mathrm{Depth}(\lambda x.t) = 1 + \mathrm{Depth}(t),\\ &\mathrm{Depth}(t_1\ t_2) = \max(\mathrm{Depth}(t_1), \mathrm{Depth}(t_2)).\end{aligned}$$

Definition 3.24 (Free variables)**.** *We denote by* $\mathrm{FV}(t)$ *the set of all variables which occur freely in* t*:*

$$\begin{aligned}&\mathrm{FV}(x) = \{x\}, \qquad \mathrm{FV}(c) = \varnothing,\\ &\mathrm{FV}(\lambda x.t) = \mathrm{FV}(t) \backslash \{x\},\\ &\mathrm{FV}(t_1\ t_2) = \mathrm{FV}(t_1) \cup \mathrm{FV}(t_2).\end{aligned}$$

A variable x in t is said to be *bound* if t contains a subterm $\lambda x.t'$. To increase readability, we use uppercase letters X, Y, Z to denote free variables and lowercase letters x, y, z for bound variables. Notice that a variable can be bound and free in a term, for this case either style is fine. Terms of the form $(\ldots(\hbar\ t_1)\ldots t_m)$, where $\hbar$ is a constant or a variable, will be written as $\hbar(t_1, \ldots, t_m)$, and terms of the form $\lambda x_1.\cdots.\lambda x_n.t$ as $\lambda x_1, \ldots, x_n.t$. We use $\vec{x}$ as a short-hand for $x_1, \ldots, x_n$, e.g., $\lambda\vec{x}.t$. Terms are assumed to be well-typed and in η-long β-normal form. Therefore, all terms have the form $\lambda x_1, \ldots, x_n.\hbar(t_1, \ldots, t_m)$, where $n, m \geqslant 0$, $t_1, \ldots, t_m$ have also this form, and the term $\hbar(t_1, \ldots, t_m)$ has a basic type.

In contrast to the definition of a top symbol (or head of a term) from section 3.1, we do not take abstraction into account for the top symbol of a lambda term.

Definition 3.25 (Top symbol)**.** *For a term* $t = \lambda x_1, \ldots, x_n.\hbar(t_1, \ldots, t_m)$ *with* $n, m \geqslant 0$*, its* head *(or top symbol) is defined as* $\mathrm{Top}(t) = \hbar$*.*

[†]We assume that $\mathcal{V}$ contains countably infinite term variables of each type.

Positions in λ-terms are defined with respect to their tree representation in the usual way, as string of integers. For instance, in the term $f(\lambda x.\lambda y.g(\lambda z.h(z,y,x)),\lambda u.g(u))$, the symbol f stands in the position ϵ, the occurrence of $\lambda x.$ stands in the position 1, the bound occurrence of y in $1{\cdot}1{\cdot}1{\cdot}1{\cdot}1{\cdot}2$, the bound occurrence of u in $2{\cdot}1{\cdot}1$, etc. We denote the subterm at position I of a term t by $t|_I$, e.g., $f(\lambda x.g(x),\lambda u.g(u))|_{2\cdot 1} = g(u)$.

The *path to a position* in a λ-term is defined as the sequence of symbols from the root to the node at that position (not including) in the tree representation of the term. For instance, the path to the position $1{\cdot}1{\cdot}1{\cdot}1{\cdot}1$ in $f(\lambda x.\lambda y.g(\lambda z.h(z,y,x)),\lambda u.g(u))$ is $f,\lambda x,\lambda y,g,\lambda z$.

Definition 3.26 (Higher-order pattern)**.** *A* higher-order pattern *is a λ-term where, when written in η-long β-normal form, all free variable occurrences are applied to lists of pairwise distinct (η-long forms of) bound variables.*

Example 3.14. *For instance, $\lambda x.f(X(x),Y)$, $f(c,\lambda x.x)$ and $\lambda x.\lambda y.X(\lambda z.x(z),y)$ are patterns, while $\lambda x.f(X(X(x)),Y)$, $f(X(c),c)$ and $\lambda x.\lambda y.X(x,x)$ are not.*

Definition 3.27 (Substitution)**.** *A* substitution *is a mapping $\sigma : \mathcal{V} \to \mathcal{T}(\Sigma,\mathcal{V})$ from variables to terms of the same type, which is identity almost everywhere. Capturing of free variables is avoided by renaming of bound variables.*

Any substitution σ can be extended to a mapping $\hat{\sigma} : \mathcal{T} \to \mathcal{T}$ that can be applied to lambda terms so that capturing of free variables is avoided. By alpha equivalence, we can enforce that a bound variable is distinct from a given finite set of variables. The notion of substitution *domain* and *range* are defined similarly to subsection 2.1.1. Substitution application is defined by induction on the structure of terms:

$$t\hat{\sigma} ::= \begin{cases} \sigma(X) & \text{if } t = X, \\ c & \text{if } t = c, \\ \lambda x.(t'\hat{\sigma'}) & \text{if } t =_\alpha \lambda x.t' \text{ so that } x \notin \mathcal{V}(\mathrm{Ran}(\sigma')) \\ & \text{where } \sigma' \text{ is defined by } \{Y \mapsto \sigma(Y) \mid Y \in \mathrm{Dom}(\sigma)\backslash\{x\}\} \\ t_1\hat{\sigma}\ t_2\hat{\sigma} & \text{if } t = (t_1\ t_2). \end{cases}$$

To simplify the notation, we do not distinguish between a substitution σ and its extension $\hat{\sigma}$. We write $\vec{x}\sigma$ for $x_1\sigma,\ldots,x_n\sigma$, if $\vec{x} = x_1,\ldots,x_n$. Similarly, for a set of terms S, we define $S\sigma = \{t\sigma \mid t \in S\}$.

We denote mappings from variables to variables of the same type by π,ρ,μ and use the notation $\pi :: [X \mapsto Y]$ to recursively define a mapping: Let $\pi' = \pi :: [X \mapsto Y]$, then

$$\pi' ::= \begin{cases} \pi'(Z) = Y & \text{if } Z = X, \\ \pi'(Z) = \pi(Z) & \text{if } Z \neq X. \end{cases}$$

The identity mapping is denoted by Id. We use $[X_1 \mapsto Y_1,\ldots,X_n \mapsto Y_n]$ as a shortcut for $Id :: [X_1 \mapsto Y_1] :: \ldots :: [X_n \mapsto Y_n]$, where all the X_i's are pairwise disjoint. Since variables are terms, we can use such mappings as a definition of a variable renaming substitution. To simplify the notation, we implicitly convert them into substitutions whenever we use the postfix notation. For instance, $t[X_1 \mapsto Y_1,\ldots,X_n \mapsto Y_n]$ implicitly denotes the application of the substitution $\{X_1 \mapsto Y_1,\ldots,X_n \mapsto Y_n\}$ to the term t.

A term t is *more general* that a term s, written $s \preceq_{=_\alpha} t$, if there exists a substitution σ such that $s\sigma =_\alpha t$. The strict part of this relation is denoted by $\prec_{=_\alpha}$. The relation $\preceq_{=_\alpha}$ is a partial order and generates the equivalence relation which we denote by $\simeq_{=_\alpha}$. To simplify the notation, we drop the subscript for the equivalence relation $=_\alpha$ and write, e.g., $\preceq$ instead of $\preceq_{=_\alpha}$ in the current section.

Definition 3.28 (Higher-order pattern generalization)**.** *A term t is called a* generalization *of two terms t_1 and t_2 if $t \preceq t_1$ and $t \preceq t_2$. It is a* higher-order pattern generalization *(pattern generalization in short) if additionally t is a higher-order pattern. It is a* least general *higher-order pattern generalization, (pattern lgg in short), of t_1 and t_2, if there is no pattern generalization s of t_1 and t_2 which satisfies $t \prec s$.*

3.2.2 Anti-Unification Algorithm for Lambda Terms $\mathfrak{G}_\mathsf{P}$

Here we consider a variant of higher-order anti-unification problem, where the input terms are arbitrary lambda terms in η-long β-normal form and the output is a higher-order pattern. We present a rule-based anti-unification algorithm that solves the following problem:

Given: Higher-order terms t and s of the same type in η-long β-normal form.

Find: A least general higher-order *pattern generalization* of t and s.

We are looking for a higher-order pattern r which is least general among all higher-order patterns which generalize t and s. There can still be a generalization of t and s which is strictly less general than r, but is not a higher-order pattern. For instance, if $t = \lambda x, y.f(h(x, x, y), h(x, y, y))$ and $s = \lambda x, y.f(g(x, x, y),\ g(x, y, y))$, then $r = \lambda x, y.f(Y_1(x, y), Y_2(x, y))$ is a higher-order pattern lgg of t and s. However, the term $\lambda x, y.f(Z(x, x, y), Z(x, y, y))$, which is not a higher-order pattern, is less general than r and generalizes t and s.

Definition 3.29 (Anti-unification equation)**.** *An* anti-unification equation *(AUE) is a triple $X(\vec{x})\colon t \triangleq s$ where*

- *$\lambda\vec{x}.X(\vec{x})$, $\lambda\vec{x}.t$, and $\lambda\vec{x}.s$ are terms of the same type,*
- *$\lambda\vec{x}.X(\vec{x})$ is a higher-order pattern,*
- *t and s are in η-long β-normal form,*
- *X is a variable that does neither occur in t nor in s.*

The variable X is called a generalization variable. *The term $X(\vec{x})$ is called the* generalization term.

Definition 3.30 (Higher-order pattern anti-unification algorithm)**.** *The higher-order anti-unification algorithm that computes pattern generalizations is formulated in a rule-based manner. It works on triples of the form P; S; σ, where*

- *P is the set of AUEs to be solved (the problem set);*
- *S is a set of already solved AUEs (the store);*

- σ *is a substitution (computed so far) mapping variables to patterns.*
- *for all* $X(\vec{x})\colon t \triangleq s \in P \cup S$ *holds that* X *occurs in* $P \cup S$ *only once.*

We call such a tuple a state *and the algorithm is called* $\mathfrak{G}_{\mathsf{P}}$*, where* P *stands for the fact that we compute pattern generalizations. The four transformation rules of the algorithm, which are defined below, operate on states.*

Remark 3.1. *One assumption we make on the set* $P \cup S$ *is that each occurrence of* λ *binds a distinct name variable (in other words, all names of bound variables are distinct). By alpha equivalence, this assumption does not impose a loss of generality.*

In the transformation rules, we use the symbol Y for a fresh variable of the corresponding type and the symbol $\cup$ stands for disjoint union.

Dec: **Decomposition**

$\{X(\vec{x})\colon \hbar(t_1,\ldots,t_m) \triangleq \hbar(s_1,\ldots,s_m)\} \cup P;\ S;\ \sigma \Longrightarrow$
$\{Y_1(\vec{x})\colon t_1 \triangleq s_1,\ldots,Y_m(\vec{x}) : t_m \triangleq s_m\} \cup P;\ S;\ \sigma\{X \mapsto \lambda\vec{x}.\hbar(Y_1(\vec{x}),\ldots,Y_m(\vec{x}))\}$,
where $\hbar$ is a constant or $\hbar \in \vec{x}$.

Abs: **Abstraction**

$\{X(\vec{x})\colon \lambda y.t \triangleq \lambda z.s\} \cup P;\ S;\ \sigma \Longrightarrow$
$\{Y(\vec{x},y)\colon t \triangleq s\{z \mapsto y\}\} \cup P;\ S;\ \sigma\{X \mapsto \lambda\vec{x},y.Y(\vec{x},y)\}$.

Sol: **Solve**

$\{X(\vec{x})\colon t \triangleq s\} \cup P;\ S;\ \sigma \Longrightarrow P;\ \{Y(\vec{y})\colon t \triangleq s\} \cup S;\ \sigma\{X \mapsto \lambda\vec{x}.Y(\vec{y})\}$,
where t and s are of a basic type, $\mathrm{Top}(t) \neq \mathrm{Top}(s)$ or $\mathrm{Top}(t) = \mathrm{Top}(s) = Z \notin \vec{x}$, and $\vec{y}$ is a subsequence of $\vec{x}$ consisting of the variables that appear freely in t or in s.

Mer: **Merge**

$P;\ \{X(\vec{x})\colon t_1 \triangleq t_2, Z(\vec{z})\colon s_1 \triangleq s_2\} \cup S;\ \sigma \Longrightarrow$
$P;\ \{X(\vec{x})\colon t_1 \triangleq t_2\} \cup S;\ \sigma\{Z \mapsto \lambda\vec{z}.X(\vec{x}\pi)\}$,
where $\pi : \{\vec{x}\} \to \{\vec{z}\}$ is a bijection, extended as a substitution, with $t_1\pi =_\alpha s_1$ and $t_2\pi =_\alpha s_2$.

To compute generalizations for terms t and s, we start with the *initial state* $\{X\colon t \triangleq s\};\ \varnothing;\ Id$, where X is a fresh variable, and apply the rules as long as possible. The state to which no rule applies has the form $\varnothing;\ S;\ \sigma$, where Mer does not apply to S. It is called the *final state*. We will prove termination, soundness, and completeness of $\mathfrak{G}_{\mathsf{P}}$, as well as uniqueness of the final state (modulo $\simeq_{=_\alpha}$) in subsection 3.2.7. The unique final state for two terms t and s to be generalized is denoted by $\mathfrak{G}_{\mathsf{P}}(X\colon t \triangleq s)$, where X is the fresh generalization variables. The pattern generalization that corresponds to a final state $\varnothing;\ S;\ \sigma = \mathfrak{G}_{\mathsf{P}}(X\colon t \triangleq s)$ is $X\sigma = r$ and the store S contains all the differences of the input terms so that $r\sigma_{\mathrm{L}}(S) =_\alpha t$ and $r\sigma_{\mathrm{R}}(S) =_\alpha s$, where the substitutions $\sigma_{\mathrm{L}}(S)$ and $\sigma_{\mathrm{R}}(S)$ are defined by:

Definition 3.31. *We define two substitutions obtained from a set* S *of AUEs:*

$$\sigma_{L}(S) ::= \{Y \mapsto \lambda\vec{y}.t_1 \mid Y(\vec{y})\colon t_1 \triangleq t_2 \in S\}$$
$$\sigma_{R}(S) ::= \{Y \mapsto \lambda\vec{y}.t_2 \mid Y(\vec{y})\colon t_1 \triangleq t_2 \in S\}$$

Discussion of the Algorithm. One can easily verify that a triple obtained from P; S; σ by applying any of the transformation rules of $\mathfrak{G}_{\mathsf{P}}$ to a state is indeed a state: For each expression $X(\vec{x})\colon t \triangleq s \in P \cup S$, the terms $X(\vec{x})$, t and s have the same type, $\lambda\vec{x}.X(\vec{x})$ is a higher-order pattern, s and t are in η-long β-normal form, and X does not occur in t and s. Moreover, all generalization variables are distinct and substitutions map variables to patterns. The property that each occurrence of λ in $P \cup S$ binds a unique variable is also maintained. It guarantees that in the Abs rule, the variable y is fresh for s. After the application of the rule, y will appear nowhere else in $P \cup S$ except $Y(\vec{x}, y)$ and, maybe, t and s.

The idea of the store S is to keep track of already solved AUEs in order to reuse in generalizations an existing variable. The Mer rule merges variables that can be reused. This is important, since we aim at computing lggs. Merging requires to solve a matching problem $\{t_1 \sim s_1, t_2 \sim s_2\}$ with the mapping π which bijectively maps the variables from $\vec{x}$ to the variables from $\vec{y}$. In general, when we want to find a solution of a matching problem M, which bijectively maps variables from a finite set D to a finite set R, we say that we are looking for a *permuting matcher* of M from D to R. The sets D and R are supposed to have the same cardinality.

Note that a permuting matcher, if it exists, is unique. It follows from the fact that there can be only one capture-avoiding renaming of free variables which matches a higher-order term to another. Since M is a matching problem for higher-order terms with free variables from D and their potential values from R, it can have at most one such matcher. An algorithm that computes it is given in subsection 3.2.4 below.

3.2.3 Illustration of the Algorithm $\mathfrak{G}_{\mathsf{P}}$

Before introducing the algorithm that constructively solves the decision problem of the existence of a permuting matcher, we illustrate the algorithm $\mathfrak{G}_{\mathsf{P}}$ on a couple of examples.

Example 3.15. *Let* $t = \lambda x, y.f(U(g(x), y), U(g(y), x))$ *and* $s = \lambda x', y'.f(h(y', g(x')), h(x', g(y')))$. *Then* $\mathfrak{G}_{\mathsf{P}}$ *performs the following transformations:*

$$
\begin{aligned}
& \{X\colon \lambda x, y.f(U(g(x), y), U(g(y), x)) \triangleq \lambda x', y'.f(h(y', g(x')), h(x', g(y')))\}; \varnothing; Id \\
\Longrightarrow^{2}_{\mathsf{Abs}}\ & \{X'(x, y)\colon f(U(g(x), y), U(g(y), x)) \triangleq f(h(y, g(x)), h(x, g(y)))\}; \varnothing; \\
& \{X \mapsto \lambda x, y.X'(x, y)\} \\
\Longrightarrow_{\mathsf{Dec}}\ & \{Y_1(x, y)\colon U(g(x), y) \triangleq h(y, g(x)), Y_2(x, y)\colon U(g(y), x) \triangleq h(x, g(y))\}; \varnothing; \\
& \{X \mapsto \lambda x, y.f(Y_1(x, y), Y_2(x, y)), \dots\} \\
\Longrightarrow_{\mathsf{Sol}}\ & \{Y_2(x, y)\colon U(g(y), x) \triangleq h(x, g(y))\}; \{Y_1(x, y)\colon U(g(x), y) \triangleq h(y, g(x))\}; \\
& \{X \mapsto \lambda x, y.f(Y_1(x, y), Y_2(x, y)), \dots\} \\
\Longrightarrow_{\mathsf{Sol}}\ & \varnothing; \{Y_1(x, y)\colon U(g(x), y) \triangleq h(y, g(x)), Y_2(x, y)\colon U(g(y), x) \triangleq h(x, g(y))\}; \\
& \{X \mapsto \lambda x, y.f(Y_1(x, y), Y_2(x, y)), \dots\} \\
\Longrightarrow_{\mathsf{Mer}}\ & \varnothing; \{Y_1(x, y)\colon U(g(x), y) \triangleq h(y, g(x))\} \\
& \{X \mapsto \lambda x, y.f(Y_1(x, y), Y_1(y, x)), \dots, Y_2 \mapsto \lambda x, y.Y_1(y, x)\}
\end{aligned}
$$

The computed higher-order pattern is $r = \lambda x, y.f(Y_1(x, y), Y_1(y, x))$. *It generalizes the input terms* t *and* s *and the store* $S = \{Y_1(x, y)\colon U(g(x), y) \triangleq h(y, g(x))\}$ *con-*

tains all the differences so that $r\sigma_{_L}(S) = r\{Y_1 \mapsto \lambda x, y.U(g(x), y)\} = t$ *and* $r\sigma_{_R}(S) = r\{Y_1 \mapsto \lambda x, y.h(y, g(x))\} = s$.

Example 3.16. *Two more examples illustrating the generalizations computed by* $\mathfrak{G}_\mathsf{P}$*:*

- *For* $\lambda x, y, z.g(f(x, z), f(y, z), f(y, x))$ *and* $\lambda x', y', z'.g(h(y', x'), h(x', y'), h(z', y'))$, $\mathfrak{G}_\mathsf{P}$ *computes their generalization* $\lambda x, y, z.g(Y_1(x, y, z), Y_1(y, x, z), Y_1(y, z, x))$
- *For* $\lambda x, y.f(\lambda z.U(z, y, x), U(x, y, x))$ *and* $\lambda x', y'.f(\lambda z'.h(y', z', x'), h(y', x', x'))$, $\mathfrak{G}_\mathsf{P}$ *computes their generalization* $\lambda x, y.f(\lambda z.Y_1(x, y, z), Y_2(x, y))$.

Example 3.17. *In order to illustrate how* $\mathfrak{G}_\mathsf{P}$ *can be used to detect clones of code pieces that are based on* λ*-terms, we borrow an example from Miller and Nadathur [60]. There, they give some tail recursive programs with the indention to use* λ*Prolog to transform them into iterative programs. We use those examples of tail recursive programs for another purpose: We aim at finding the similarities and, based on the generalization result, we suggest to refactor the two programs into one generalized version that can be easily reused. Figure 3.5 shows the two programs.*

In order to represent simple recursive schemes, the constants fixpt and cond are introduced in [60] in the following way:

$$\forall x\ ((\mathsf{fixpt}\ x) = (x\ (\mathsf{fixpt}\ x)))$$
$$\forall x \forall y\ ((\mathsf{cond}\ \mathsf{truth}\ x\ y) = x)$$
$$\forall x \forall y\ ((\mathsf{cond}\ \mathsf{false}\ x\ y) = y)$$

We consider the following two pieces of programs to illustrate the use case of embedding $\mathfrak{G}_\mathsf{P}$ in a refactoring tool for languages like λProlog:

$$\mathsf{fixpt}(\lambda fact.\lambda n.\lambda m.\mathsf{cond}(eq(0, n), m, fact(sub(n, 1), mult(n, m))))$$
$$\mathsf{fixpt}(\lambda sum.\lambda n.\lambda m.\mathsf{cond}(eq(0, n), m, sum(sub(n, 1), add(m, 1))))$$

Figure 3.5: Two programs represented as simply-typed λ-terms.

Figure 3.6 shows the generalization (modulo bound variable renaming) computed by $\mathfrak{G}_\mathsf{P}$ for the two input terms from Figure 3.5. From the store S of the corresponding final state we get the substitutions $\sigma_{\mathrm{L}}(S) = \{X \mapsto \lambda x.\lambda y.mult(x, y)\}$ and $\sigma_{\mathrm{R}}(S) = \{X \mapsto \lambda x.\lambda y.add(y, 1)\}$.

$$\mathsf{fixpt}(\lambda fun.\lambda n.\lambda m.\mathsf{cond}(eq(0, n), m, fun(sub(n, 1), \mathbf{X}(n, m))))$$

Figure 3.6: Generalization of the programs from Figure 3.5.

The generalization λ-term can easily be used in a λProlog predicate that instantiates the variable X (in addition to other variables that get instantiated by the λ-term itself and its arguments) by a given function, e.g., $\lambda x.\lambda y.add(y, 1)$. Therefore, we suggest to use the generalized version for the sake of reusability in languages like λProlog.

The results computed by $\mathfrak{G}_\mathsf{P}$ are higher-order pattern generalizations of the input terms. We will prove it formally, when we establish soundness of $\mathfrak{G}_\mathsf{P}$. The computed results are, in fact, pattern lggs. We will prove this fact in Theorem 3.25. The other properties of $\mathfrak{G}_\mathsf{P}$ are also discussed later in subsection 3.2.7.

3.2.4 Computation of Permuting Matchers: The Algorithm $\mathcal{M}$

In this section we describe the algorithm $\mathcal{M}$ to compute permuting matchers. It is a rule-based algorithm working on quintuples of the form D; R; M; ρ; π (called state) where

- D is a set of domain variables,
- R is a set of range variables,
- D and R have the same cardinality,
- M is a set of matching problems of the form $\{s_1 \sim t_1, \ldots, s_m \sim t_m\}$,
- ρ and π are mappings from variables to variables (computed so far).

The mapping ρ is supposed to keep bound variable renamings to deal with abstractions, while in π we compute the permuting matcher to be returned in case of success. Like in the rules for $\mathfrak{G}_{\mathsf{P}}$, also here each occurrence of λ in the problem set M binds a unique variable. Furthermore, all the bound variables are distinct from the variables in $D \cup R$. The three transformation rules of the algorithm are the following:

Dec-M: **Decomposition**

D; R; $\{h(t_1, \ldots, t_m) \sim g(s_1, \ldots, s_m)\} \uplus M$; ρ; $\pi \Longrightarrow$
D; R; $\{t_1 \sim s_1, \ldots, t_m \sim s_m\} \cup M$; ρ; π,

where each of h and g is a constant or a variable, $h \notin D$, $g \notin R$, and $h\pi = g\rho$.

Abs-M: **Abstraction**

D; R; $\{\lambda x.t \sim \lambda y.s\} \uplus M$; ρ; $\pi \Longrightarrow$ D; R; $\{t \sim s\} \cup M$; $\rho :: [x \mapsto y]$; π.

Per-M: **Permuting**

$\{x\} \uplus D$; $\{y\} \uplus R$; $\{x(t_1, \ldots, t_m) \sim y(s_1, \ldots, s_m)\} \uplus M$; ρ; $\pi \Longrightarrow$
D; R; $\{t_1 \sim s_1, \ldots, t_m \sim s_m\} \cup M$; ρ; $\pi :: [x \mapsto y]$,

where x and y have the same type.

The input for $\mathcal{M}$ is initialized in the Mer rule, which needs to compute a bijection $\pi : \{\vec{x}\} \to \{\vec{z}\}$ such that $t_1\pi =_\alpha s_1$ and $t_2\pi =_\alpha s_2$. Notice that, in the sequence $\vec{x}$ all the occurrences of variable names are pairwise distinct, since bound variables are assumed to be unique. The same holds for $\vec{z}$. We proceed in the following way:

We create the set of domain variables D from $\vec{x}$ and the set of range variables R from $\vec{z}$. Then we create the *initial state* D; R; $\{t_1 \sim s_1, t_2 \sim s_2\}$; Id; Id and apply the rules Dec-M, Abs-M and Per-M exhaustively. If no rule applies to a state D; R; M; ρ; π with $M \neq \varnothing$, then it is transformed into $\bot$, called the *failure state*. The quintuple D; R; $\varnothing$; ρ; π is called the *success state*. No rule applies to it either. When $\mathcal{M}$ reaches the success state, we say that $\mathcal{M}$ computes the permuting matcher π. When $\mathcal{M}$ reaches the failure state, we say that it fails. The *result computed by* $\mathcal{M}$ is $\bot$ in the case of failure and $\boldsymbol{\pi}$ if we reach the success state. We denoted this result by $\mathcal{M}(D, R, M)$.

3.2.5 Illustration of the Algorithm $\mathcal{M}$

Before we turn to discussing the properties of $\mathcal{M}$, we illustrate its usage on some examples.

Example 3.18. *To compute the permuting matcher of* $\{x(y,z) \sim x(z,y), X(y,\lambda u.u) \sim X(z,\lambda v.v)\}$ *from* $\{x,y,z\}$ *to* $\{x,y,z\}$ *by* $\mathcal{M}$*, we create the initial state* $\{x,y,z\}; \{x,y,z\}; \{x(y,z) \sim x(z,y), X(y,\lambda u.u) \sim X(z,\lambda v.v)\}; Id; Id$ *and apply the transformation rules exhaustively:*

$$\begin{aligned}
&\{x,y,z\}; \{x,y,z\}; \{x(y,z) \sim x(z,y), X(y,\lambda u.u) \sim X(z,\lambda v.v)\}; Id; Id\\
\Longrightarrow_{\mathsf{Per\text{-}M}}\ &\{y,z\}; \{y,z\}; \{y \sim z, z \sim y, X(y,\lambda u.u) \sim X(z,\lambda v.v)\}; Id; [x \mapsto x]\\
\Longrightarrow_{\mathsf{Per\text{-}M}}\ &\{z\}; \{y\}; \{z \sim y, X(y,\lambda u.u) \sim X(z,\lambda v.v)\}; Id; [x \mapsto x, y \mapsto z]\\
\Longrightarrow_{\mathsf{Per\text{-}M}}\ &\varnothing; \varnothing; \{X(y,\lambda u.u) \sim X(z,\lambda v.v)\}; Id; [x \mapsto x, y \mapsto z, z \mapsto y]\\
\Longrightarrow_{\mathsf{Dec\text{-}M}}\ &\varnothing; \varnothing; \{y \sim z, \lambda u.u \sim \lambda v.v\}; Id; [x \mapsto x, y \mapsto z, z \mapsto y]\\
\Longrightarrow_{\mathsf{Dec\text{-}M}}\ &\varnothing; \varnothing; \{\lambda u.u \sim \lambda v.v\}; Id; [x \mapsto x, y \mapsto z, z \mapsto y]\\
\Longrightarrow_{\mathsf{Abs\text{-}M}}\ &\varnothing; \varnothing; \{v \sim u\}; [u \mapsto v]; [x \mapsto x, y \mapsto z, z \mapsto y]\\
\Longrightarrow_{\mathsf{Dec\text{-}M}}\ &\varnothing; \varnothing; \varnothing; [v \mapsto u]; [x \mapsto x, y \mapsto z, z \mapsto y]
\end{aligned}$$

From the computed mapping we obtain $[x \mapsto x, y \mapsto z, z \mapsto y]$*. One can easily see that it is a permuting matcher of* $\{x(y,z) \sim x(z,y), X(y,\lambda u.u) \sim X(z,\lambda v.v)\}$ *from* $\{x,y,z\}$ *to* $\{x,y,z\}$*.*

Example 3.19. *To compute the permuting matcher of* $\{x(z,z) \sim x(z,y), f(y) \sim f(z)\}$ *from* $\{x,y,z\}$ *to* $\{x,y,z\}$ *by* $\mathcal{M}$*, we create the initial state and apply the transformation rules exhaustively:*

$$\begin{aligned}
&\{x,y,z\}; \{x,y,z\}; \{x(z,z) \sim x(z,y), f(y) \sim f(z)\}; Id; Id\\
\Longrightarrow_{\mathsf{Dec\text{-}M}}\ &\{x,y,z\}; \{x,y,z\}; \{x(z,z) \sim x(z,y), y \sim z\}; Id; Id\\
\Longrightarrow_{\mathsf{Per\text{-}M}}\ &\{y,z\}; \{y,z\}; \{z \sim z, z \sim y, y \sim z\}; Id; [x \mapsto x]\\
\Longrightarrow_{\mathsf{Per\text{-}M}}\ &\{y\}; \{y\}; \{z \sim y, y \sim z\}; Id; [x \mapsto x, z \mapsto z].
\end{aligned}$$

The derivation by $\mathcal{M}$ *fails because no more rule is applicable to the nonempty set of matching problems.*

3.2.6 Properties of the Algorithm $\mathcal{M}$

The algorithm $\mathcal{M}$ maintains the following invariants:

Lemma 3.17 (Invariant 1)**.** *For each tuple* $D; R; M; \rho; \pi$ *in a derivation performed by* $\mathcal{M}$*, the sets* D *and* R *have the same number of elements.*

Proof. In the initial state, D and R have the same number of elements because this is implied by the condition in the rule Mer which causes computation of permuting matcher. The only rule that changes D and R is Per-M and it removes one element of both sets. □

Lemma 3.18 (Invariant 2)**.** *For each tuple D; R; $\{t_1 \sim s_1, \ldots, t_m \sim s_m\}$; ρ; π in a derivation performed by $\mathcal{M}$, $D \subseteq \bigcup_{i=1}^{m} \mathrm{FV}(t_i)$ and $R \subseteq \bigcup_{i=1}^{m} \mathrm{FV}(s_i)$.*

Proof. In the initial tuple this property holds, because the tuple originates from the AUEs in the store of the anti-unification algorithm. AUEs enter the store only with the Solve rule, which makes sure that the list of variables given as arguments to the generalization variable appear in the terms to be generalized. Furthermore, during the derivation in $\mathcal{M}$, the rules do not violate the invariant property:

- In the case of Dec-M, the condition $\mathcal{h} \notin D$ and $\mathcal{g} \notin R$ keeps this property intact.
- For Abs-M it is true because of the assumption that all bound variables are distinct from the variables in $D \cup R$. This assumption is fulfilled by the initial assumption of the anti-unification algorithm that all the bound variables are unique.
- For Per-M it is obvious. □

Lemma 3.19 (Invariant 3)**.** *For each tuple D_i; R_i; M_i; ρ_i; π_i in a derivation performed by $\mathcal{M}$ starting from D; R; M; ρ; π, the following equalities hold: $D_i \cup Dom(\pi_i) = D$ and $R_i \cup Ran(\pi_i) = R$.*

Proof. The initial tuple D_0; R_0; M_0; ρ_0; π_0 has this property, because $D_0 = D$, $R_0 = R$, and $\pi_0 = Id$. During the derivation, the rules either keep D_i and R_i unchanged, or remove one element from each, obtaining in this way D_{i+1} and R_{i+1}, and adding to π_{i+1} a pair consisting of those elements. Hence, the property is maintained during the derivation. □

Theorem 3.20 (Termination)**.** *The algorithm $\mathcal{M}$ terminates on any input.*

Proof. Each rule strictly reduces the multiset of sizes of matching problems M in the tuples it operates on. Since each tuple D; R; M; ρ; π with $M \neq \varnothing$ can be transformed by one of the rules or leads to failure, the final state in the derivation is either the success or the failure state. □

Theorem 3.21 (Soundness)**.** *If $\mathcal{M}$ computes a mapping π for a given tuple D; R; M; Id; Id, then π is a permuting matcher of M from D to R.*

Proof. Obviously, π maps variables from D to R. It follows from the way how the Per-M rule constructs π. The differences between t and s for $t \sim s \in M$ are either repaired by the bindings from π constructed by Per-M, or the differences are α-equivalences repaired by the bindings from ρ constructed by Abs-M, or the failure occurs since no rule can be applied. The Dec-M rule only applies if the heads are the same symbols (under ρ and π) and those symbols cannot be permuted, since $\mathcal{h} \notin D$ and $\mathcal{g} \notin R$.

The bijection property is more involved: The Per-M rule (namely, the fact that it removes x and y from D and R) guarantees that there is an injective mapping from a subset of D onto a subset of R. Since all variables of D (resp. R) appear freely in the left (resp. right) hand sides of equations in M (the second invariant), each derivation either stops with failure, or eventually reduces D and R to $\varnothing$ by applications of Per-M (see the first invariant, the same number of elements in D and R). The latter, by the third invariant, means that there is an injective mapping from D onto R, expressed by π. Hence, π is a bijection from D to R and $\mathcal{M}$ is sound. □

Theorem 3.22 (Completeness)**.** *If there exists a permuting matcher of M from D to R, then $\mathcal{M}$ computes it and it is unique.*

Proof. Uniqueness follows from the fact that there can be only one capture-avoiding renaming of free variables which matches a higher-order term to another. Since we have already proved soundness of $\mathcal{M}$, we have only to show that if there exists a permuting matcher of M from D to R, then $\mathcal{M}$ does not fail for D; R; M; Id; Id. Let μ be such a matcher. Then $t\mu = s$ for all $t \sim s \in M$. This means that, if t has a form $\hbar(t_1, \ldots, t_n)$, then s should be $g(s_1, \ldots, s_n)$ and $\hbar\mu = g$, $t_i\mu = s_i$ for all $1 \leqslant i \leqslant n$. If t has a form $\lambda x.t'$, then s should be of the form $\lambda y.s'$ and $t'\mu = s'\{y \mapsto x\}$.

Assume by contradiction that $\mathcal{M}$ fails. That means that there exists the state D_k; R_k; $\{t \sim s\} \cup M_k$; ρ_k; π_k to which no rule applies. Since the steps performed by $\mathcal{M}$ before it either decompose the terms argumentwise (Dec-M and Per-M), or remove abstraction (Abs-M), by the definitions of matcher and substitution application we should have $t\mu = s\rho_k$. This equation means that t and s have the same types. Hence, the only case why no rule in $\mathcal{M}$ applies to the state is that t and s should be, respectively, of the form $\hbar(t_1, \ldots, t_n)$ and $g(s_1, \ldots, s_m)$ with $\hbar\pi_k \neq g\rho_k$, where $\hbar \notin D_k$ or $g \notin R_k$. Because of the uniqueness of the matcher, $x\pi_k = x\mu$, for all $x \in D \backslash D_k$. On the other hand, $\hbar\mu = g\rho_k$, because μ matches t to $s\rho_k$.

Hence, we have $\hbar \in D_k$. This also implies that $g \in R_k$ because μ is a bijection from D to R and $\hbar\mu = g\rho_k = g$. (Notice that R is disjoint from all bound variables.) Now Per-M is applicable and we have a contradiction to the assumption that no rule applies to D_k; R_k; $\{t \sim s\} \cup M_k$; ρ_k; π_k. The obtained contradiction shows that if there exists a permuting matcher of M from D to R, then $\mathcal{M}$ does not fail for D; R; M; Id; Id, which implies completeness of $\mathcal{M}$. □

Remark 3.2. *The algorithm $\mathcal{M}$ is an extended alpha-equivalence test. It searches for a renaming of variables from the domain D into the range R while deciding alpha-equivalence.*

3.2.7 Properties of the Algorithm $\mathfrak{G}_\mathsf{P}$

Theorem 3.23 (Termination)**.** *The procedure $\mathfrak{G}_\mathsf{P}$, which uses $\mathcal{M}$ to compute permuting matchers, terminates for all input terms t and s.*

Proof. We define the complexity measure of the triple P; S; σ as a pair of multisets $(M(P), M(S))$, where the multiset $M(L)$ is defined as

$$M(L) = \{\min(\mathrm{Depth}(t), \mathrm{Depth}(s)) \mid X(\vec{x})\colon t \triangleq s \in L\}$$

for any L. Measures are compared lexicographically. Obviously, each rule in $\mathfrak{G}_\mathsf{P}$ strictly reduces it. The ordering is well-founded. The procedure $\mathcal{M}$ in the rule Mer is terminating. Hence, $\mathfrak{G}_\mathsf{P}$ terminates. □

Theorem 3.24 (Soundness)**.** *If $\{X\colon t \triangleq s\}; \varnothing; Id \Longrightarrow^* \varnothing; S; \sigma$ is a derivation in $\mathfrak{G}_\mathsf{P}$, then*

(a) $X\sigma$ is a higher-order pattern in η-long β-normal form,

(b) $X\sigma \preceq t$ *and* $X\sigma \preceq s$.

Proof. To prove that $X\sigma$ is a higher-order pattern, we use the facts that first, X is a higher order pattern and, second, at each step $P_1;\ S_1;\ \varphi \Longrightarrow P_2;\ S_2;\ \varphi\vartheta$ if $X\varphi$ is a higher-order pattern, then $X\varphi\vartheta$ is also a higher-order pattern. The latter property follows from stability of patterns under substitution application and from the fact that substitutions in the rules map variables to higher-order patterns. As for $X\sigma$ being in η-long β-normal form, this is guaranteed by the series of applications of the Abs rule, even if Dec introduces an AUE whose generalization term is not in this form. It finishes the (sketch of the) proof of (a).

Proving (b) is more involved. First, we prove that if $P_1;\ S_1;\ \varphi \Longrightarrow P_2;\ S_2;\ \varphi\vartheta$ is one step, then for any $X(\vec{x})\colon t \triangleq s \in P_1 \cup S_1$, we have $X(\vec{x})\vartheta \preceq t$ and $X(\vec{x})\vartheta \preceq s$. Note that if $X(\vec{x})\colon t \triangleq s$ was not transformed at this step, then this property trivially holds for it. Therefore, we assume that $X(\vec{x})\colon t \triangleq s$ is selected and prove the property for each rule:

Dec: Here $t = h(t_1, \ldots, t_m)$, $s = h(s_1, \ldots, s_m)$, and $\vartheta = \{X \mapsto \lambda\vec{x}.h(Y_1(\vec{x}), \ldots, Y_m(\vec{x}))\}$. Then $X(\vec{x})\vartheta = h(Y_1(\vec{x}), \ldots, Y_m(\vec{x}))$. Let σ_1 and σ_2 be substitutions defined, respectively, by $Y_i\sigma_1 = \lambda\vec{x}.t_i$ and $Y_i\sigma_2 = \lambda\vec{x}.s_i$ for all $1 \leqslant i \leqslant m$. Such substitutions obviously exist since the Y's introduced by the Dec rule are fresh. Then $X(\vec{x})\vartheta\sigma_1 = h(t_1, \ldots, t_m)$, $X(\vec{x})\vartheta\sigma_2 = h(s_1, \ldots, s_m)$ and, hence, $X(\vec{x})\vartheta \preceq t$ and $X(\vec{x})\vartheta \preceq s$.

Abs: Here $t = \lambda y_1.t'$, $s = \lambda y_2.s'$, and $\vartheta = \{X \mapsto \lambda\vec{x}, y.X'(\vec{x}, y)\}$. Then $X(\vec{x})\vartheta = \lambda y.X'(\vec{x}, y)$. Let $\sigma_1 = \{X' \mapsto \lambda\vec{x}, y.t'\}$ and $\sigma_2 = \{X' \mapsto \lambda\vec{x}, y.s'\}$. Then $X(\vec{x})\vartheta\sigma_1 = \lambda y.t' = t$, $X(\vec{x})\vartheta\sigma_2 = \lambda y.s' = s$, and, hence, $X(\vec{x})\vartheta \preceq t$ and $X(\vec{x})\vartheta \preceq s$.

Sol: We have $\vartheta = \{X \mapsto \lambda\vec{x}.Y(\vec{y})\}$, where $\vec{y}$ is the subsequence of $\vec{x}$ consisting of the variables that appear freely in t or s. Let $\sigma_1 = \{Y \mapsto \lambda\vec{y}.t\}$ and $\sigma_2 = \{Y \mapsto \lambda\vec{y}.s\}$. Then $X(\vec{x})\vartheta\sigma_1 = t$, $X(\vec{x})\vartheta\sigma_2 = s$, and, hence, $X(\vec{x})\vartheta \preceq t$ and $X(\vec{x})\vartheta \preceq s$.

If Mer applies, then there exists $Y(\vec{y})\colon t' \triangleq s' \in S_1$ such that $\mathcal{M}(\{\vec{x}\}, \{\vec{y}\}, t \sim t', s \sim s')$ is a permuting matcher π, and $\vartheta = \{Y \mapsto \lambda\vec{y}.X(\vec{x}\pi)\}$. Then $X(\vec{x})\vartheta \preceq t$ and $X(\vec{x})\vartheta \preceq s$ obviously hold. As for the $Y(\vec{y})\colon t' \triangleq s'$, let $\sigma_1 = \{X \mapsto \lambda\vec{x}.t\}$ and $\sigma_2 = \{X \mapsto \lambda\vec{x}.s\}$. Then $Y(\vec{y})\vartheta\sigma_1 = (\lambda\vec{x}.t)(\vec{x}\pi) = t\pi = t'$, $Y(\vec{y})\vartheta\sigma_2 = (\lambda\vec{x}.s)(\vec{x}\pi) = s\pi = s'$, and, hence, $Y(\vec{y})\vartheta \preceq t'$ and $Y(\vec{y})\vartheta \preceq s'$.

Now, we proceed by induction on the length of derivation l. In fact, we will prove a more general statement: If $P_0;\ S_0;\ \vartheta_0 \Longrightarrow^* \varnothing;\ S_n;\ \vartheta_0\vartheta_1 \cdots \vartheta_n$ is a derivation in $\mathfrak{G}_{\mathsf{P}}$, then for any $X(\vec{x})\colon t \triangleq s \in P_0 \cup S_0$ we have $X(\vec{x})\vartheta_1 \cdots \vartheta_n \preceq t$ and $X(\vec{x})\vartheta_1 \cdots \vartheta_n \preceq s$.

When $l = 1$, it is exactly the one-step case we just proved. Assume that the statement is true for any derivation of the length n and prove it for a derivation $P_0;\ S_0;\ \vartheta_0 \Longrightarrow P_1;\ S_1;\ \vartheta_0\vartheta_1 \Longrightarrow^* \varnothing;\ S_n;\ \vartheta_0\vartheta_1 \cdots \vartheta_n$ of the length $n + 1$.

Below the composition $\vartheta_i\vartheta_{i+1} \cdots \vartheta_k$ is abbreviated as ϑ_i^k with $k \geqslant i$. Let $X(\vec{x})\colon t \triangleq s$ be an AUE selected for transformation at the current step. (Again, the property trivially holds for the AUEs which are not selected.) We consider each rule:

Dec: $t = h(t_1, \ldots, t_m)$, $s = h(s_1, \ldots, s_m)$ and $X(\vec{x})\vartheta_1^1 = h(Y_1(\vec{x}), \ldots, Y_m(\vec{x}))$. By the induction hypothesis, $Y_i(\vec{x})\vartheta_2^n \preceq t_i$ and $Y_i(\vec{x})\vartheta_2^n \preceq s_i$ for all $1 \leqslant i \leqslant m$. By construction of ϑ_2^n, if there is $U \in \mathrm{FV}(\mathrm{Ran}(\vartheta_2^n))$, then there is an AUE of the form $U(\vec{u})\colon t' \triangleq s' \in S_n$. Let σ (resp. φ) be a substitution which maps each such U to the corresponding t' (resp. s'). Then $Y_i(\vec{x})\vartheta_2^n\sigma = t_i$ and $Y_i(\vec{x})\vartheta_2^n\varphi = s_i$. Since $X(\vec{x})\vartheta_1^n = h(Y_1(\vec{x}), \ldots, Y_m(\vec{x}))\vartheta_2^n$, we get that $X(\vec{x})\vartheta_1^n\sigma = t$, $X(\vec{x})\vartheta_1^n\varphi = s$, and, hence, $X(\vec{x})\vartheta_1^n \preceq t$ and $X(\vec{x})\vartheta_1^n \preceq s$.

Abs: Here $t = \lambda y_1.t'$, $s = \lambda y_2.s'$, $X(\vec{x})\vartheta_1^1 = \lambda y.X'(\vec{x}, y)$, and P_1 contains the AUE $X'(\vec{x}, y)\colon t'\{y_1 \mapsto y\} \triangleq s'\{y_2 \mapsto y\}$. By the induction hypothesis, $X'(\vec{x}, y)\vartheta_2^n \preceq t'\{y_1 \mapsto y\}$ and $X'(\vec{x}, y)\vartheta_2^n \preceq s'\{y_1 \mapsto y\}$. Since $X(\vec{x})\vartheta_1^n = \lambda y.X'(\vec{x}, y)\vartheta_2^n$ and due to the way how y was chosen, we finally get $X(\vec{x})\vartheta_1^n \preceq \lambda y.t'\{y_1 \mapsto y\} = t$ and $X(\vec{x})\vartheta_1^n \preceq \lambda y.s'\{y_2 \mapsto y\} = s$.

Sol: We have $X(\vec{x})\vartheta_1^1 = Y(\vec{y})$ where Y is in the store. By the induction hypothesis, $Y(\vec{y})\vartheta_2^n \preceq t$ and $Y(\vec{y})\vartheta_2^n \preceq s$. Therefore, $X(\vec{x})\vartheta_1^n \preceq t$ and $X(\vec{x})\vartheta_1^n \preceq s$.

For Mer, there exists $Y(\vec{y})\colon t' \triangleq s' \in S_0$ such that $\mathcal{M}(\{\vec{x}\}, \{\vec{y}\}, t \sim t', s \sim s')$ is a permuting matcher π, and $\vartheta_1^1 = \{Y \mapsto \lambda\vec{y}.X(\vec{x}\pi)\}$. By the induction hypothesis, $X(\vec{x})\vartheta_1^n = X(\vec{x})\vartheta_2^n \preceq t$ and $X(\vec{x})\vartheta_1^n = X(\vec{x})\vartheta_2^n \preceq s$. These imply that $X(\vec{x}\pi)\vartheta_1^n \preceq t'$ and $X(\vec{x}\pi)\vartheta_1^n \preceq s'$, which, together $Y\vartheta_1^n = X(\vec{x}\pi)$, yields $Y(\vec{y})\vartheta_1^n \preceq t'$and $Y(\vec{y})\vartheta_1^n \preceq s'$. □

Hence, the result computed by $\mathfrak{G}_\mathsf{P}$ for $X\colon t \triangleq s$ generalizes both t and s. We call $X\sigma$, a *generalization of t and s computed by $\mathfrak{G}_\mathsf{P}$*.

Theorem 3.25 (Completeness)**.** *Let $\lambda\vec{x}.t_1$ and $\lambda\vec{x}.t_2$ be higher-order terms and $\lambda\vec{x}.s$ be a higher-order pattern such that $\lambda\vec{x}.s$ is a generalization of both $\lambda\vec{x}.t_1$ and $\lambda\vec{x}.t_2$. Then $\lambda\vec{x}.s \preceq X\sigma$, where $X\sigma$ is a generalization computed by $\mathfrak{G}_\mathsf{P}$ for $X\colon \lambda\vec{x}.t_1 \triangleq \lambda\vec{x}.t_2$.*

Proof. By structural induction on s. We can assume without loss of generality that $\lambda\vec{x}.s$ is an lgg of $\lambda\vec{x}.t_1$ and $\lambda\vec{x}.t_2$. We also assume that it is in the η-long β-normal form.

If s is a variable, then there are two cases: Either $s \in \vec{x}$, or $s \notin \vec{x}$. In the first case, we have $s = t_1 = t_2$. The Dec rule gives $\sigma = \{X \mapsto \lambda\vec{x}.s\}$ and, hence, $\lambda\vec{x}.s \preceq \lambda\vec{x}.X(\vec{x})\sigma = s$. In the second case, either $\mathrm{Top}(t_1) \neq \mathrm{Top}(t_2)$, or $\mathrm{Top}(t_1) = \mathrm{Top}(t_2) \notin \vec{x}$. Sol is supposed to give us $\sigma = \{X \mapsto \lambda\vec{x}.X'(\vec{x'})\}$, where $\vec{x'}$ is a subsequence of $\vec{x}$ consisting of variables occurring freely in t_1 or in t_2. But $\vec{x'}$ should be empty, because otherwise s would not be just a variable (remember that $\lambda\vec{x}.s$ is an lgg of $\lambda\vec{x}.t_1$ and $\lambda\vec{x}.t_2$ in the η-long β-normal form). Hence, we have $\sigma = \{X \mapsto \lambda\vec{x}.X'\}$ and $\lambda\vec{x}.s \preceq \lambda\vec{x}.X(\vec{x})\sigma$, because $s\{s \mapsto X'\} = X(\vec{x})\sigma$.

If s is a constant c, then $t_1 = t_2 = c$. We can apply the Dec rule, obtaining $\sigma = \{X \mapsto \lambda\vec{x}.c\}$ and, hence, $s = c \preceq X(\vec{x})\sigma = c$. Therefore, $\lambda\vec{x}.s \preceq \lambda\vec{x}.X(\vec{x})\sigma$.

If $s = \lambda x.s'$, then t_1 and t_2 must have the forms $t_1 = \lambda x.t_1'$ and $t_2 = \lambda y.t_2'$, and s' must be an lgg of t_1' and t_2'. Abs gives a new state $\{X'(\vec{x}, x)\colon t_1' \triangleq t_2'\{x \mapsto y\}\}; \varnothing; \sigma_1$, where $\sigma_1 = \{X \mapsto \lambda\vec{x}, x.X'(\vec{x}, x)\}$. By the induction hypothesis, we can compute a substitution σ_2 such that $\lambda\vec{x}, x.s' \preceq \lambda\vec{x}, x.X'(\vec{x}, x)\sigma_2$. Composing σ_1 and σ_2 into σ, we have $X(\vec{x})\sigma = \lambda x.X'(\vec{x}, x)\sigma_2$. Hence, we get $\lambda\vec{x}.s = \lambda\vec{x}.\lambda x.s' \preceq \lambda\vec{x}.\lambda x.X'(\vec{x}, x)\sigma_2 = \lambda\vec{x}.X(\vec{x})\sigma$.

Finally, assume that s is a compound term $h(s_1, \ldots, s_n)$. If $h \notin \vec{x}$ is a variable, then $s_1, \ldots, s_n$ are distinct variables from $\vec{x}$ (because $\lambda\vec{x}.s$ is a higher-order pattern). That means that $s_1, \ldots, s_n$ appear freely in t_1 or t_2. Moreover, either $\mathrm{Top}(t_1) \neq \mathrm{Top}(t_2)$, or $\mathrm{Top}(t_1) = \mathrm{Top}(t_2) = h$. In both cases, we can apply the Sol rule to obtain $\sigma = \{X \mapsto \lambda\vec{x}.Y(s_1, \ldots, s_n)\}$. Obviously, $\lambda\vec{x}.s \preceq \lambda\vec{x}.X(\vec{x})\sigma = \lambda\vec{x}.Y(s_1, \ldots, s_n)$.

If $h \in \vec{x}$ or if it is a constant, then we should have $\mathrm{Top}(t_1) = \mathrm{Top}(t_2)$. Assume they have the forms $t_1 = h(t_1^1, \ldots, t_n^1)$ and $t_2 = h(t_1^2, \ldots, t_n^2)$. We proceed by the Dec rule, obtaining $\{Y_i(\vec{x})\colon t_i^1 \triangleq t_i^2 \mid 1 \leqslant i \leqslant n\}; \varnothing; \sigma_0$, where $\sigma_0 = \{X \mapsto \lambda\vec{x}.h(Y_1(\vec{x}), \ldots, Y_n(\vec{x}))\}$. By the induction hypothesis, we can construct derivations $\Delta_1, \ldots, \Delta_n$ computing the substitutions $\sigma_1, \ldots, \sigma_n$, respectively, such that $\lambda\vec{x}.s_i \preceq \lambda\vec{x}.Y_i(\vec{x})\sigma_i$ for $1 \leqslant i \leqslant n$. These derivations, together with the initial Dec step, can be combined into one derivation, of the form $\Delta = \{X(\vec{x})\colon t_1 \triangleq t_2\}; \varnothing; \sigma_0 \Longrightarrow \{Y_i(\vec{x})\colon t_i^1 \triangleq t_i^2 \mid 1 \leqslant i \leqslant n\}; \varnothing; \sigma_0 \Longrightarrow^* \varnothing; S_n; \sigma_0\sigma_1 \cdots \sigma_n$.

If s does not contain duplicate variables free in $\lambda\vec{x}.s$, then the construction of Δ and the fact that $\lambda\vec{x}.s_i \preceq \lambda\vec{x}.Y_i(\vec{x})\sigma_i$ for $1 \leqslant i \leqslant n$ guarantee $\lambda\vec{x}.s \preceq \lambda\vec{x}.X(\vec{x})\sigma_0\sigma_1 \cdots \sigma_n$. If s contains duplicate variables free in $\lambda\vec{x}.s$ (e.g., of the form $\lambda\vec{u_1}.Z(\vec{z_1})$ and $\lambda\vec{u_2}.Z(\vec{z_2})$, where $\vec{z_1}$ and $\vec{z_2}$ have the same length) at positions I and J, it indicates that

(a) $t_1|_I$ and $t_1|_J$ differ from each other by a permutation of variables bound in t_1,

(b) $t_2|_I$ and $t_2|_J$ differ from each other by the same (modulo variable renaming) permutation of variables bound in t_2,

(c) the path to position I is the same (modulo bound variable renaming) in t_1 and t_2. It equals (modulo bound variable renaming) the path to position I in s, and

(d) the path to position J is the same (modulo bound variable renaming) in t_1 and t_2. It equals (modulo bound variable renaming) the path to position J in s.

Then, because of (c) and (d), we should have two AUEs in S_n: One, between (renamed variants of) $t_1|_I$ and $t_2|_I$, and the other one between (renamed variants of) $t_1|_J$ and $t_2|_J$. The possible renaming of variables is caused by the fact that Abs might have been applied to obtain the AUEs. Let those AUEs be $Z(\vec{z_1})\colon r_1^1 \triangleq r_1^2$ and $Z'(\vec{z_2})\colon r_2^1 \triangleq r_2^2$. The conditions (a) and (b) make sure that $\mathcal{M}(\{\vec{z_1}\}, \{\vec{z_2}\}, \{r_1^1 \sim r_1^1, r_1^2 \sim r_2^2\})$ is a permuting matcher π, which means that we can apply the rule Mer with the substitution $\sigma_1' = \{Z' \mapsto \lambda\vec{z_2}.Z(\vec{z_1}\pi)\}$. We can repeat this process for all duplicated variables in s, extending Δ to the derivation $\Delta' = \{X(\vec{x})\colon t_1 \triangleq t_2\}; \varnothing; \sigma_0 \Longrightarrow \{Y_i(\vec{x})\colon t_i^1 \triangleq t_i^2 \mid 1 \leqslant i \leqslant n\}; \varnothing; \sigma_0 \Longrightarrow^* \varnothing; S_n; \sigma_0\sigma_1 \cdots \sigma_n \Longrightarrow^* \varnothing; S_{n+m}; \sigma_0\sigma_1 \cdots \sigma_n\sigma_1' \cdots \sigma_m'$, where $\sigma_1', \ldots, \sigma_m'$ are substitutions introduced by the applications of the Mer rule. Let $\sigma = \sigma_0\sigma_1 \cdots \sigma_n\sigma_1' \cdots \sigma_m'$. By this construction, we have $\lambda\vec{x}.s \preceq \lambda\vec{x}.X(\vec{x})\sigma$. It finishes the proof, since $\vec{x}.X(\vec{x})\sigma = \vec{x}.(X\sigma)(\vec{x}) = X\sigma$. □

Depending which AUE is selected to perform a step, there can be different derivations in $\mathfrak{G}_\mathsf{P}$ starting from the same AUE, leading to different generalizations. The next theorem states that all those generalizations are equivalent.

Theorem 3.26 (Uniqueness Modulo $\simeq_{=_\alpha}$)**.** *Let* $\{X\colon t \triangleq s\}; \varnothing; Id \Longrightarrow^* \varnothing; S_1; \sigma_1$ *and* $\{X\colon t \triangleq s\}; \varnothing; Id \Longrightarrow^* \varnothing; S_2; \sigma_2$ *be two maximal derivations in* $\mathfrak{G}_\mathsf{P}$ *from* $X\colon t \triangleq s$*. Then* $X\sigma_1 \simeq X\sigma_2$*.*

Proof. It is not hard to notice that if it is possible to change the order of applications of rules (but sticking to the same selected AUEs for each rule) then the result remains the same: If $\Delta_1 = P_1; S_1; \sigma_1 \Longrightarrow_{\mathsf{R1}} P_2; S_2; \sigma_1\vartheta_1 \Longrightarrow_{\mathsf{R2}} P_3; S_3; \sigma_1\vartheta_1\vartheta_2$ and $\Delta_2 = P_1; S_1; \sigma_1 \Longrightarrow_{\mathsf{R2}} P_2'; S_2'; \sigma_1\vartheta_2 \Longrightarrow_{\mathsf{R1}} P_3'; S_3'; \sigma_1\vartheta_2\vartheta_1$ are two two-step derivations, where $\mathsf{R1}$ and $\mathsf{R2}$ are (not necessarily different) rules and each of them transforms the same AUE(s) in both Δ_1 and Δ_2, then $P_3 = P_3'$, $S_3 = S_3'$, and $\sigma_1\vartheta_1\vartheta_2 = \sigma_1\vartheta_2\vartheta_1$ (modulo the names of fresh variables).

Decomposition, Abstraction, and Solve rules transform the selected AUE in a unique way. We show that it is irrelevant in which order we perform matching in the Merge rule.

Let $P; \{Z(\vec{z})\colon t_1 \triangleq s_1, Y(\vec{y})\colon t_2 \triangleq s_2\} \cup S; \sigma \Longrightarrow P; \{Z(\vec{z})\colon t_1 \triangleq s_1\} \cup S; \sigma\{Y \mapsto \lambda\vec{y}.Z(\vec{z}\pi)\}$ be the merging step with $\pi = \mathcal{M}(\{\vec{z}\}, \{\vec{y}\}, \{t_1 \sim t_2, s_1 \sim s_2\})$. If we do it in the other way around, we would get the step $P; \{Z(\vec{z})\colon t_1 \triangleq s_1, Y(\vec{y})\colon t_2 \triangleq s_2\} \cup S; \sigma \Longrightarrow P; \{Y(\vec{y})\colon t_2 \triangleq s_2\} \cup S; \sigma\{Z \mapsto \lambda\vec{z}.Y(\vec{y}\mu)\}$, where $\mu = \mathcal{M}(\{\vec{y}\}, \{\vec{z}\}, \{t_2 \sim t_1, s_2 \sim s_1\})$. But $\mu = \pi^{-1}$, because of bijection.

Let $\vartheta_1 = \sigma\rho_1$ with $\rho_1 = \{Y \mapsto \lambda\vec{y}.Z(\vec{z}\pi)\}$ and $\vartheta_2 = \sigma\rho_2$ with $\rho_2 = \{Z \mapsto \lambda\vec{z}.Y(\vec{y}\pi^{-1})\}$. Our goal is to prove that $X\vartheta_1 \simeq X\vartheta_2$. For this, we have to prove two inequalities: $X\vartheta_1 \preceq X\vartheta_2$ and $X\vartheta_2 \preceq X\vartheta_1$. To show $X\vartheta_1 \preceq X\vartheta_2$, we first need to prove the equality:

$$\lambda\vec{y}.Z(\vec{z}\pi)\rho_2 = \lambda\vec{y}.Y(\vec{y}). \tag{3.1}$$

Its left hand side is transformed as $\lambda\vec{y}.Z(\vec{z}\pi)\rho_2 = \lambda\vec{y}.Z(\vec{z}\pi)\{Z \mapsto \lambda\vec{z}.Y(\vec{y}\pi^{-1})\} = \lambda\vec{y}.(\lambda\vec{z}.Y(\vec{y}\pi^{-1})(\vec{z}\pi))$. β-reduction of $\lambda\vec{z}.Y(\vec{y}\pi^{-1})(\vec{z}\pi)$ replaces each occurrence of $z_i \in \vec{z}$ in $Y(\vec{y}\pi^{-1})$ with $z_i\pi$, which is the same as applying π to $Y(\vec{y}\pi^{-1})$. Since $\vec{y}\pi^{-1}\pi = \vec{y}$, we get $\lambda\vec{y}.(\lambda\vec{z}.Y(\vec{y}\pi^{-1})(\vec{z}\pi)) = \lambda\vec{y}.Y(\vec{y}\pi^{-1}\pi) = \lambda\vec{y}.Y(\vec{y})$ and (3.1) is proved.

Next, starting from $X\vartheta_1\rho_2$, we can transform it as $X\vartheta_1\rho_2 = X\sigma\rho_1\rho_2 = X\sigma\{Y \mapsto \lambda\vec{y}.Z(\vec{z}\pi)\rho_2, Z \mapsto \lambda\vec{z}.Y(\vec{y}\pi^{-1})\} =_{\text{by (3.1)}} X\sigma\{Y \mapsto \lambda\vec{y}.Z(\vec{z}\pi)\rho_2, Z \mapsto \lambda\vec{z}.Y(\vec{y}\pi^{-1})\} = X\sigma\{Y \mapsto \lambda\vec{y}.Y(\vec{y}), Z \mapsto \lambda\vec{z}.Y(\vec{y}\pi^{-1})\} = X\sigma\{Y \mapsto \lambda\vec{y}.Y(\vec{y})\}\{Z \mapsto \lambda\vec{z}.Y(\vec{y}\pi^{-1})\}$. At this step, since the equality $=$ is $\alpha\beta\eta$-equivalence, we can omit the application of the substitution $\{Y \mapsto \lambda\vec{y}.Y(\vec{y})\}$ and proceed: $X\sigma\{Y \mapsto \lambda\vec{y}.Y(\vec{y})\}\{Z \mapsto \lambda\vec{z}.Y(\vec{y}\pi^{-1})\} = X\sigma\{Z \mapsto \lambda\vec{z}.Y(\vec{y}\pi^{-1})\} = X\sigma\rho_2X\vartheta_2$. Hence, we got $X\vartheta_1\rho_2 = X\vartheta_2$, which implies $X\vartheta_1 \preceq X\vartheta_2$.

$X\vartheta_2 \preceq X\vartheta_1$ can be proved analogously. Hence, $X\vartheta_1 \simeq X\vartheta_2$, which means that it is irrelevant in which order we perform matching in the Merge rule. Therefore, no matter how different derivations are constructed, the computed generalizations are equivalent. □

Hence, for given terms t and s, the anti-unification algorithm $\mathfrak{G}_\mathsf{P}$ computes their generalization, a higher-order pattern, which is less general than any other higher-order pattern which generalizes t and s.

3.2.8 Complexity Analysis of $\mathcal{M}$ and $\mathfrak{G}_\mathsf{P}$

Now we turn to discussing the computational behavior of the algorithms introduced in this section. In both algorithms, we assume that each bound variable is unique,

i.e., it does neither occur freely in a term nor exists another binding for the same variable name. This assumption does not impose a loss of generality neither does it effect the computational behavior of the algorithms: If the input does not satisfy the condition each bound variable to be unique, we can rename the variables before calling $\mathfrak{G}_{\mathsf{P}}$ (resp. $\mathcal{M}$). It can be done in linear time, using a "chained-like" hash table whose buckets are stacks (instead of linked lists of chained hash tables) for variable renaming, and traversing the terms in preorder.

Theorem 3.27. *The algorithm $\mathcal{M}$ has linear space and time complexity in the size of the input.*

Proof. For the input consisting of the sets of domain variables D, range variables R, and matching equations M, the size is the cardinality of D plus the cardinality of R plus the number of symbols in M.

The terms to be matched can be represented as trees in the standard way. The sets D and R can be encoded as hash tables. These representations occupy space linear to the size of the input. The space can grow at most twice by representing renaming and permuting substitutions as hash tables. Hence, the space complexity is linear.

As for the time complexity, we can see that the algorithm visits each node of the trees to be matched at most once. We perform the following linear time steps: Collecting the set of bound variables V_r appearing in the right sides of matching equations in M, constructing the initial hash tables T_D and T_R for D and R (we can assume that the hash functions are perfect), and constructing two hash tables for the mappings. The one for computing the permuting matcher is denoted by T_π. Its set of keys is D. We can reuse the same hash function as for T_D. Each address in T_π is initialized with null. Another table, T_ρ, is designed for renaming of bound variables. Its set of keys is V_r. We assume a perfect hash function also here.

The operations performed at each node are the following ones:

By Dec-M: Look up the values for $\hbar$ in T_D and for $\jmath$ in T_R, to make sure that $\hbar \notin D$ and $\jmath \notin R$. If the test fails, the rule is not applicable.

Next, if $\hbar \notin D$ and $\jmath \notin R$, then look up the value for $\hbar$ in T_π, look up the value for $\jmath$ in T_ρ, and compare them with each other. If the values of $\hbar$ or $\jmath$ are not found in the tables, then just use the corresponding value (i.e., $\hbar$ or $\jmath$) in the comparison.

By Abs-M: Modifying an entry in T_ρ: For a mapping $[y \mapsto x]$, we put x in the table at the address corresponding to the hash index of y: $T_\rho[hash(y)] = x$. Since all bound variables are distinct, we will not have to modify the same entry in T_ρ again.

By Per-M: Modifying an entry for x in T_π: For a mapping $[x \mapsto y]$, we put y in the address corresponding to the hash index of x: $T_\pi[hash(x)] = y$. As we destroy the entries for x in T_D and for y in T_R, we will not modify the same entry again.

All our hash functions are perfect. Searching, insertion and deletion in hash tables with perfect hash functions are done in constant time. We assume that two alphabet symbols can be compared in constant time. Hence, all the operations performed by $\mathcal{M}$

at each node of the input trees are done in constant time. It implies that $\mathcal{M}$ has linear time complexity. □

Theorem 3.28 (Complexity of $\mathfrak{G}_{\mathsf{P}}$)**.** *The algorithm $\mathfrak{G}_{\mathsf{P}}$, when using $\mathcal{M}$ to compute permuting matchers, has space complexity $O(n)$ and time complexity $O(n^2)$, where n is the size (the number of symbols) of input.*

Proof. By design of the rules and Theorem 3.26 we can first apply the rules Dec and Abs exhaustively. Afterwards we apply Sol as long as possible, such that the applications of the Mer rule are postponed till the end. Doing so, we get a maximal derivation like $P_0; \varnothing; Id \Longrightarrow^*_{\mathsf{Dec},\mathsf{Abs}} P_k; \varnothing; \sigma_k \Longrightarrow^*_{\mathsf{Sol}} \varnothing; S_l; \sigma_l \Longrightarrow^*_{\mathsf{Mer}} \varnothing; S_m; \sigma_m$ where $P_0 = \{X_0 \colon t_0 \triangleq s_0\}$.

As $X_0\sigma_m$ is the final result, we keep only the mapping $X_0 \mapsto r_i$ in σ_i and omit all the others for all $0 \leqslant i \leqslant m$. The rules Dec, Abs, and Sol introduce unique occurrences of fresh variables at every application, therefore every free variable appears only once in r_i, for $i \leqslant l$.

Phase $\Longrightarrow^*_{\mathsf{Dec},\mathsf{Abs}}$: In the first phase we share the representation of arguments (the abstracted bound variables) which are applied to one of the introduced fresh variables. For instance, the two applications $Y_1(\vec{x})$ and $Y_2(\vec{x})$ point to the same argument vector $\vec{x}$. Furthermore we use linked lists to avoid copying the variable vectors when applying the Abs rule (e.g. using $y :: \vec{x}$, were arguments are stored in reverse order). This sharing is used for the problem set as well as for the mapping in our substitution. The substitution composition $\sigma_i\{X \mapsto \lambda\vec{x}.h(Y_1(\vec{x}), \ldots, Y_m(\vec{x}))\}$, which is needed in the Dec rule, is equivalent to replacing the subterm $X(\vec{x})$ in r_i with $h(Y_1(\vec{x}), \ldots, Y_m(\vec{x}))$. Similarly, the substitution composition $\sigma\{X \mapsto \lambda\vec{x}, y.X'(\vec{x}, y)\}$ used by Abs is equivalent to replacing $X(\vec{x})$ by $\lambda y.X'(\vec{x}, y)$. As X only appears once and we share representation of arguments, these compositions of subtitutions can be done in linear time on the size of σ. The renaming $s\{z \mapsto y\}$ in the Abs rule can also be done in linear time on the size of s. Along this phase, the size of σ, and the size of any AUE is bounded by n. Moreover, the sum of the sizes of terms t and s of all the AUEs $X(\vec{x}) \colon t \triangleq s$ is also bounded by n. Therefore, these rule applications can be done in linear time on the size of the original problem.

Phase $\Longrightarrow^*_{\mathsf{Sol}}$: In the second phase, the arguments of generalization variables are narrowed. As only those argument variables which appear in one of the terms of an AUE are kept, the reduced argument vector $\vec{y}$ is bound by the size of the respective terms t and s. There is no need to share the representation of those narrowed argument vectors anymore. The representation of $\vec{y}$ can be constructed without reusing the representation of $\vec{x}$ in linear time on the size of the AUE $X(\vec{x}) \colon t \triangleq s$, hence on n. The substitution composition $\sigma\{X \mapsto \lambda\vec{x}.Y(\vec{y})\}$ used by Sol is equivalent to replacing the subterm $X(\vec{x})$ by $Y(\vec{y})$. Again, like in the previous phase, this composition can be done in linear time.

Since $\vec{y}$ is bound by the size of the respective terms t and s, after applying Sol exhaustively, the entire state, namely the store, is of size $O(n)$.

The number of applications of the rules Dec, Abs, and Sol is bounded by the size of the input. Since each of the rule applications require time $O(n)$, all this together leads to $O(n^2)$ time complexity and $O(n)$ space complexity for the first two phases.

Phase $\Longrightarrow^*_{Mer}$: The space complexity of the store $\{X_1(\overrightarrow{x_1})\colon s_1 \triangleq t_1, \ldots, X_k(\overrightarrow{x_k})\colon s_k \triangleq t_k\}$ is $O(n)$. Let n_i be the size of $X_i(\overrightarrow{x_i})\colon s_i \triangleq t_i$, for all $1 \leqslant i \leqslant k$. From Theorem 3.27 we know that solving the matching problem $\mathcal{M}(D,\, R,\, \{s_i \sim s_j, t_i \sim t_j\})$ with $D = \{\overrightarrow{x_i}\}$ and $R = \{\overrightarrow{x_j}\}$ has space and time complexity $O(n_i+n_j)$, for any $1 \leqslant i < j \leqslant k$. The third phase requires to solve this problem for each pair of AUEs. This leads to the time complexity $\sum_{i=1}^{k}\sum_{j=i+1}^{k} O(n_i+n_j) \leqslant O(\sum_{i=1}^{k}\sum_{j=1}^{k} n_i+n_j) = O(\sum_{i=1}^{k} kn_i+n) = O(kn+kn) \leqslant O(n^2)$ for all the applications of $\mathcal{M}(D,\, R,\, \{s_i \sim s_j, t_i \sim t_j\})$ together.

As there are only $O(n)$ many AUEs in the store and one AUE is removed for each success case of $\mathcal{M}(D,\, R,\, \{s_i \sim s_j, t_i \sim t_j\})$, there are at most $O(n)$ substitution compositions at the third phase. We choose the following strategy for the merging phase: Fix one AUE $\{X_i(\overrightarrow{x_i})\colon s_i \triangleq t_i\}$ and merge it with all the other AUEs, such that the generalization variable X_i gets duplicated in the substitution for each success case. With this strategy, it is guaranteed, that for each success case only one generalization variable will be replaced by the substitution composition. Note, that the fixed AUE will never be removed, but for each success case, the other AUE which is merged with the fixed one will be removed. We can choose this strategy because matchability of AUEs forms an equivalence relation. E.g., for the store $\{X_1(\overrightarrow{x_1})\colon s_1 \triangleq t_1, \ldots, X_k(\overrightarrow{x_k})\colon s_k \triangleq t_k\}$, we first fix $X_1(\overrightarrow{x_1})\colon s_1 \triangleq t_1$ and try to match it with $\{X_j(\overrightarrow{x_j})\colon s_j \triangleq t_j\}$, for all $2 \leqslant j \leqslant k$. If a certain AUE, say $\{X_i(\overrightarrow{x_i})\colon s_i \triangleq t_i\}$, matches, then we remove it from the store and compose $\sigma\{X_i \mapsto \lambda\overrightarrow{x_i}.X_1(\overrightarrow{x_1}\pi)\}$. With this strategy X_i is unique in σ. Afterwards $X_1(\overrightarrow{x_1})\colon s_1 \triangleq t_1$ is not touched anymore. Now we fix $X_2(\overrightarrow{x_2})\colon s_2 \triangleq t_2$ from the remaining store and try to match it with $\{X_j(\overrightarrow{x_j})\colon s_j \triangleq t_j\}$, for all $3 \leqslant j \leqslant k$. Note that all those AUEs have not been merged yet. And so on. The uniqueness of the generalization variable X_j again leads to linear time for one substitution composition. This concludes the proof of $O(n^2)$ time complexity for the merging phase. It is easy to see, that the space is reduced by merging two AUEs. Putting everything together, gives the claimed complexity result. □

3.2.9 A Remark on Untyped Lambda Terms

One can observe that $\mathfrak{G}_\mathsf{P}$ can be adapted with a relatively little effort to work on untyped terms that are in β-normal form (cf. the formulation of the unification algorithm both for untyped and simply-typed patterns in [65]). One thing to be added is lazy η-expansion: The AUE of the form $X(\overrightarrow{x})\colon \lambda y.t \triangleq h(s_1, \ldots, s_m)$ should be transformed into $X(\overrightarrow{x})\colon \lambda y.t \triangleq \lambda z.h(s_1, \ldots, s_m, z)$ for a fresh z. (Dually for abstractions in the right hand side.) The expansion should be performed both in $\mathfrak{G}_\mathsf{P}$ and $\mathcal{M}$. In addition, Sol needs an extra condition for the case when $\mathrm{Top}(t) = \mathrm{Top}(s)$ but the terms have different number of arguments such as, e.g., in $f(a, x)$ and $f(b, x, y)$.

Chapter 4

A Library of Anti-Unification Algorithms

The open-source library described in this chapter offers anti-unification algorithms for unranked terms, nominal terms, and simply-typed lambda terms. Generalization problems in these theories arise in many areas of the computer sciences and computational mathematics, for instance, in proof generalization or analogical reasoning in higher-order or nominal logic, in learning or refactoring λ-Prolog and α-Prolog programs, in detection of similarities in XML documents or in pieces of software code, etc. (see section 1.1). Therefore, the algorithms provided by the library can be a valuable ingredient for tools that need to solve such generalization problems. We provide implementations of the following algorithms:

- first-order unranked anti-unification $\mathfrak{G}_{\mathcal{R}}$ from section 2.1 with term variables [54],
- higher-order unranked anti-unification $\mathfrak{G}_{\mathfrak{a}}^{2\mathcal{V}}$ from section 2.2,
  - its extension to $\mathfrak{G}_{\mathfrak{a}}^{4\mathcal{V}}$ from section 2.3 is straightforward and will be demonstrated in section 4.4,
- anti-unification for nominal terms $\mathfrak{G}_{\mathsf{N}}$ from section 3.1,
  - its subalgorithm for deciding equivariance $\mathcal{E}$ from subsection 3.1.5,
- anti-unification for simply-typed lambda terms $\mathfrak{G}_{\mathsf{P}}$ from section 3.2,
  - its subalgorithm for searching a permuting matcher $\mathcal{M}$ from subsection 3.2.4.

The library is written in Java and freely available under the conditions of the GNU Lesser General Public License (LGPL). All these algorithms can be accessed from the Web page of the SToUT project: `http://www.risc.jku.at/projects/stout/`. Each of them has a separate Web page with a convenient Web interface to try the algorithm online. There is also a brief explanation of the syntax and some examples. Besides using the Web interface, the user may try also a shell version of each algorithm, or download the sources, or embed the algorithm in her/his own project.

The algorithms $\mathfrak{G}_{\mathsf{N}}$ and $\mathfrak{G}_{\mathsf{P}}$ compute a unique result, as well as their subalgorithms which are needed to compute least general generalizations. In contrast to that, the algorithms $\mathfrak{G}_{\mathcal{R}}$, $\mathfrak{G}_{\mathfrak{a}}^{2\mathcal{V}}$, and $\mathfrak{G}_{\mathfrak{a}}^{4\mathcal{V}}$ compute complete sets of generalizations. In the current implementation, the minimization step is not performed automatically. Nevertheless, the user can easily perform this step by a recursive call of the respective anti-unificaiton algorithm itself.

In this chapter, for each algorithm mentioned above we explain the Web interface, discuss the implemented strategy for rule applications, and illustrate how the algorithm can be embedded in users projects.

4.1 Structure of the Library

We describe the structure of the library in a bit more detail. It consists of four Java libraries for four anti-unification algorithms (urau.jar, urauc.jar, nau.jar, and hoau.jar), which have the same structure. There is one main package which starts with the name `at.jku.risc.stout`, followed by a short abbreviation for the implemented algorithm: `urau`, `urauc`, `nau`, or `hoau`. The abbreviation stands for, respectively, unranked anti-unification ($\mathfrak{G}_{\mathcal{R}}$), unranked anti-unification with context variables ($\mathfrak{G}_{\mathfrak{a}}^{2\mathcal{V}}$), nominal anti-unification ($\mathfrak{G}_{\mathsf{N}}$), and higher-order anti-unification ($\mathfrak{G}_{\mathsf{P}}$). Under the main package there are three subpackages, namely `algo`, `data` and `util`. The `data` package has one subpackage of its own, which is called `data.atom`. Figure 4.1 illustrates the package structure. Notice that it does not contain all the Java classes but a selection of those we consider important for using the library. The main package is irrelevant for using the library, as it only contains some test cases and the user interfaces. For instance, the GWT[†] entry points which are used in the Web frontend. Nevertheless, the source code might be interesting as those Java classes serve as reference implementations of the library.

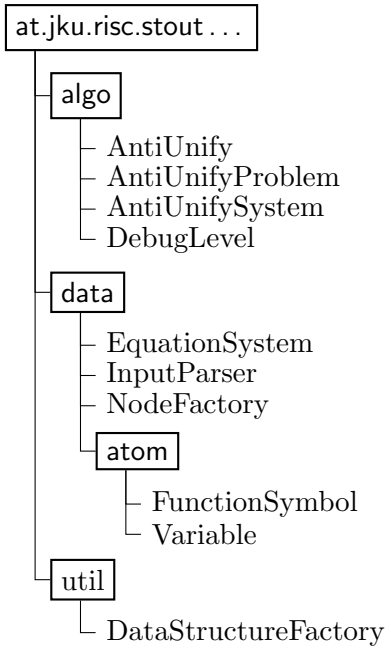

Figure 4.1: Package structure of the Java library

As the name suggests, the package `algo` contains the algorithmic part of the library. There is a Java class named `AntiUnify` which serves as entry point of the respective anti-unification algorithm. The `data` package contains some Java classes which are needed to build the term structure. Furthermore, it includes the equation system which consists of some term pairs, and it offers a default implementation of an input parser, named `InputParser`. The Java class `EquationSystem` is implemented in a generic way, so that it can be used for different types of equation systems. In the `util` package there are some utility classes like `DataStructureFactory` which is used by the library to instantiate structures (e.g., lists, queues, maps, sets). The user of the library is free to choose an arbitrary implementation for all of those data structures, which might have some advantages on the performance of the provided algorithms. The package `data.atom` contains the atomic building blocks for constructing the terms.

4.2 First-Order Unranked Anti-Unification

The unranked first-order anti-unification algorithm with hedge variables and term variables has been introduced in [54], section 5. It is an extended version of the algorithm $\mathfrak{G}_{\mathcal{R}}$ which we discussed in section 2.1. We call it $\mathfrak{G}_{\mathcal{R}}^{2\mathcal{V}}$, following the standard nota-

[†]GWT stands for Google Web Toolkit. It is used to translate the Java source code into JavaScript in order to present the algorithms on the Web.

tion used throughout this work. The $2\mathcal{V}$ stands for the two different kinds of variables and $\mathcal{R}$ stands for the rigidity function. Before discussing the implementation, we give the necessary definitions. Definition 4.1 is an extension of the terms and hedges from section 2.1. It introduces term variables in addition to the hedge variables which have been considered in Definition 2.1.

Definition 4.1 (Terms and hedges)**.** *Given pairwise disjoint countable sets of unranked function symbols $\mathcal{F}$ (symbols without fixed arity), hedge variables $\mathcal{V}_{\mathsf{H}}$, and term variables $\mathcal{V}_{\mathsf{T}}$, we define* terms *and* hedges *by the following grammar:*

$$\begin{array}{lcll} t & ::= & x \mid f(\tilde{s}) & \textit{(term)} \\ s & ::= & t \mid \tilde{x} & \textit{(hedge element)} \\ \tilde{s} & ::= & s_1, \ldots, s_n & \textit{(hedge)} \end{array}$$

where $x \in \mathcal{V}_{\mathsf{T}}$, $f \in \mathcal{F}$, $\tilde{x} \in \mathcal{V}_{\mathsf{H}}$, and $n \geqslant 0$.

A Substitution is defined by a mapping from term variables to terms and from hedge variables to hedges which is identity almost everywhere (see subsection 2.3.1). For instance, $\{x \mapsto f(a), \tilde{x} \mapsto (g(y,b),c), \tilde{y} \mapsto \epsilon\}$ defines such a substitution. Applying it to $f(x,\tilde{x},\tilde{y})$ gives $f(f(a),g(y,b),c)$. Positions, alignments, rigidity functions, etc. are defined as in subsection 2.1.2.

$\mathcal{R}_{\mathsf{T}}$-generalizations are defined similarly to $\mathcal{R}$-generalizations with the exception that consecutive term variables are allowed. In section 2.1 we did not consider term variables. Hence, Definition 4.2 relaxes the first item of Definition 2.12.

Definition 4.2 ($\mathcal{R}_{\mathsf{T}}$-generalization)**.** *Given two variable-disjoint hedges $\tilde{s}$ and $\tilde{q}$ and the rigidity function $\mathcal{R}$, we say that a hedge $\tilde{g}$ that generalizes both $\tilde{s}$ and $\tilde{q}$ is their* $\mathcal{R}_{\mathsf{T}}$-generalization*, if either $\mathcal{R}(\mathrm{Top}(\tilde{s}),\mathrm{Top}(\tilde{q})) = \varnothing$ and $\tilde{g}$ is a hedge variable, or there exists an alignment $a_1\langle i_1,j_1\rangle \cdots a_n\langle i_n,j_n\rangle \in \mathcal{R}(\mathrm{Top}(\tilde{s}),\mathrm{Top}(\tilde{q}))$ such that the following conditions are fulfilled:*

1. *If the sequence $\tilde{g}$ contains a pair of horizontal consecutive variables, then both of them are term variables.*
2. *If we remove all hedge variables that occur as elements of $\tilde{g}$, we get a sequence of the form $a_1(\tilde{g}_1),\ldots,a_n(\tilde{g}_n)$.*
3. *For every $1 \leqslant k \leqslant n$, there exists a pair of sequences $\tilde{s}_k$ and $\tilde{q}_k$ such that $\tilde{s}|_{i_k} = a_k(\tilde{s}_k)$, $\tilde{q}|_{j_k} = a_k(\tilde{q}_k)$ and $\tilde{g}_k$ is an $\mathcal{R}_{\mathsf{T}}$-generalization of $\tilde{s}_k$ and $\tilde{q}_k$.*

The implemented anti-unification algorithm solves the following problem:

Given: Two variable-disjoint hedges $\tilde{s}$ and $\tilde{q}$ and the rigidity function $\mathcal{R}$.

Find: A complete set of $\mathcal{R}_{\mathsf{T}}$-generalizations for $\tilde{s},\tilde{q}$ and $\mathcal{R}$.

For instance, $\{(g(a,a),\tilde{x},f(g(a),g(\tilde{y}))),\ (\tilde{x},g(x,x),f(g(a),g(\tilde{z})))\}$ is the minimal complete set of $\mathcal{R}_{\mathsf{T}}$-generalization of the hedges $(g(a,a),g(b,b),f(g(a),g(a)))$ and $(g(a,a),f(g(a),g))$, where $\mathcal{R}$ computes longest common subsequences.

Web page. The implementation of unranked rigid anti-unification is available from `http://www.risc.jku.at/projects/stout/software/urau.php`.

4.2.1 Explanation of the Web Interface for the Algorithm $\mathfrak{G}^{2\mathcal{V}}_{\mathcal{R}}$

The input form of the Web page of the first-order rigid unranked anti-unification algorithm consists of five rows:

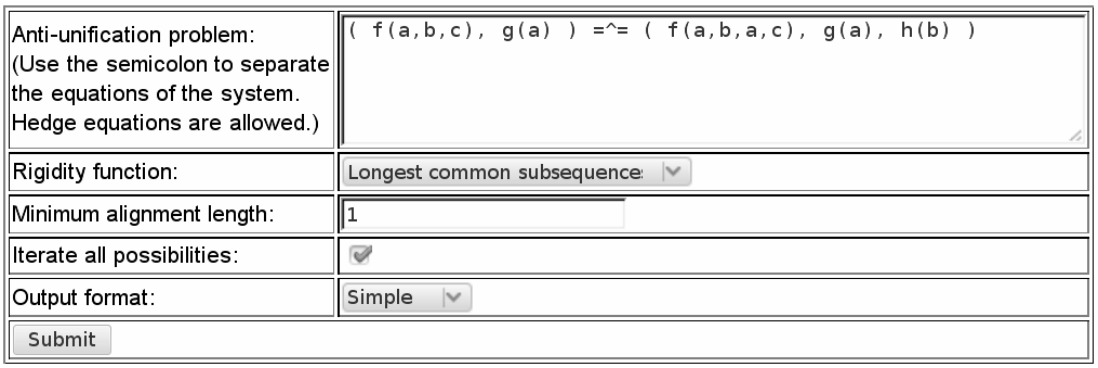

Figure 4.2: The input form of the Web presentation of the algorithm $\mathfrak{G}^{2\mathcal{V}}_{\mathcal{R}}$.

In the first row, the anti-unification problem should be given. It consists of some anti-unification equations, separated by semicolons. Each anti-unification equation consists of two hedges, with `=^=` in between. The second row contains a drop-down menu to chose a rigidity function. Currently, the only two possibilities are longest common subsequence and longest common substring.

Furthermore, in the third row, one can specify the minimal alignment length l. We define $\mathcal{R}_l(\mathrm{Top}(\tilde{s}), \mathrm{Top}(\tilde{q})) := \{\mathfrak{a} : |\mathfrak{a}| \geqslant l, \mathfrak{a} \in \mathcal{R}(\mathrm{Top}(\tilde{s}), \mathrm{Top}(\tilde{q}))\}$ as the rigidity function which corresponds to a given rigidity function $\mathcal{R}$ satisfying the length restriction. The implementation uses $\mathcal{R}_l$ and for any $\mathcal{R}$ holds $\mathcal{R}_0 = \mathcal{R}$. By unchecking the check-box from the fourth row, the user can specify to only compute the $\mathcal{R}_\mathsf{T}$-generalization for the first alignment which is returned by the rigidity function $\mathcal{R}_l$ (nondeterministically).

In the last row, the output format can be specified. One can choose form a drop-down box between simple, verbose and progress. The first choice only shows some basic facts and the computed $\mathcal{R}_\mathsf{T}$-generalizations. The verbose output format shows some additional information, like the differences at the input hedges. By choosing the progress output format, all the debug information will be shown to the user.

4.2.2 Implemented Transformation Strategy for $\mathfrak{G}^{2\mathcal{V}}_{\mathcal{R}}$

Here, we show the strategy that is used by the implementation in order to exhaustively transform a given set of AUEs P with respect to a given rigidity function $\mathcal{R}$ into a set of final states. The implementation invokes a given callback function $\mathcal{H}$ as soon as a final state is reached. We use $\uplus$ for disjoint union.

```
1 // INPUT:
2   P ← Given set of AUEs
3   R ← Given rigidity function
4   H ← Given callback function
5 // COMPUTE:
```

```
 6    V_P ← Set of generalization variables from P
 7    Q ← Set of states initialized by {(P, ∅, Id)}
 8    While Q ≠ ∅
 9      State ← (P, S, σ) ∪ Q
10      While State.P ≠ ∅
11        Fix AUE p ← x̃: s̃ ≜ q̃ from State.P
12        A ← R(Top(s̃), Top(q̃))
13        If A = ∅
14          Transform State ⟹^p_{R-Sol-H} State
15        Else
16          Transform State ⟹^{p, a ∪ A}_{R-Dec-H} State
17          While A ≠ ∅
18            Transform State ⟹^{p, a' ∪ A}_{R-Dec-H} State' ∪ Q
19      For each x̃: s̃ ≜ q̃ in State.S
20        If s̃ = q̃ = ϵ
21          Transform State ⟹^{x̃: s̃≜q̃}_{R-Clr-S} State
22      For each x̃: s̃ ≜ q̃ in State.S
23        If |s̃| = |q̃| and all elements of s̃ and q̃ are terms
24          Transform State ⟹^{x̃: s̃≜q̃}_{Nar-FO} State
25      For each p ← ξ: s̃ ≜ q̃ in State.S
26        For each p' ← ξ': s̃' ≜ q̃' in State.S
27          If p ≠ p' & s̃ = s̃' & q̃ = q̃'
28            Transform State ⟹^{p, p'}_{R-Mer-S} State
29  // CALLBACK:
30      For each x̃ in V_P
31        H(State, x̃)
```

Listing 4.1: The implemented strategy for $\mathfrak{G}^{2\mathcal{V}}_{\mathcal{R}}$.

The lines 1–4 in Listing 4.1 define the input assumptions. In line 6, generalization variables are collected from the set of input AUEs in order to apply the callback function (line 30 and 31) to the final `State` and each of that variables. A generalization that corresponds to an input AUE can be obtained by applying `State.`σ to the generalization variable.

Since the rigidity function computes a set of alignments (see line 12), the algorithm is nondeterministic. The set Q which is declared in line 7 is used to collect the states of the branching algorithm. The statement at line 9 selects and removes a state from Q. Then this state is transformed exhaustively. First the rules $\mathcal{R}$-Sol-H and $\mathcal{R}$-Dec-H are applied as long as possible. The lines 17 and 18 add new branches if $\mathcal{R}$ computes more than one alignment. When the problem set `State.`P is empty, then $\mathcal{R}$-Sol-H and $\mathcal{R}$-Dec-H are not applicable anymore. Therefore, we clear the empty AUEs from the store (line 19–21). Afterwards, hedge variables in the generalization which stand for sequences of terms of the same length in the input hedges can be transformed into a sequence of term variables. The transformation rule Nar-FO is the same as for the algorithm $\mathfrak{G}^{4\mathcal{V}}_{\mathfrak{a}}$. The lines 22–24 show its exhaustive application. Afterwards, a generalized version of the merge rule, similarly to the one of $\mathfrak{G}^{4\mathcal{V}}_{\mathfrak{a}}$, is applied exhaustively.

For each final state and each generalization variable, the callback function $\mathcal{H}$, which is provided by the user, is invoked. The user is free to process this result further as he/she needs. For instance, the minimization step can be performed by invoking $\mathfrak{G}^{2\mathcal{V}}_{\mathcal{R}}$ again within that callback function.

4.2.3 Using the Algorithm $\mathfrak{G}^{2\mathcal{V}}_{\mathcal{R}}$ in Java

We assume that there are two data sources `in1` and `in2` available in form of `Reader` instances, each of them containing one of the hedges to be generalized. Moreover, the variable `eqSys` is of appropriate type and there is a Boolean variable `iterateAll` which corresponds to the option "Iterate all possibilities" of the Web interface. We explain the usage of the library on a code fragment:

```
1  RigidityFnc rFnc=new RigidityFncSubsequence().setMinLen(3);
2  eqSys = new EquationSystem<AntiUnifyProblem>() {
3    public AntiUnifyProblem newEquation() {
4      return new AntiUnifyProblem();
5    }
6  };
7  new InputParser<>(eqSys).parseHedgeEquation(in1, in2);
8  new AntiUnify(rFnc, eqSys, DebugLevel.SILENT) {
9    public void callback(AntiUnifySystem res, Variable var) {
10     System.out.println(res.getSigma().get(var));
11   }
12 }.antiUnify(iterateAll, null);
```

Listing 4.2: Usage example of the algorithm $\mathfrak{G}^{2\mathcal{V}}_{\mathcal{R}}$ in Java

In the first line a certain rigidity function is instantiated and the minimum alignment size is set to the value 3. There are two rigidity functions available from the library. The one which is used in the code fragment computes longest common subsequence alignments. The other one is called `RigidityFncSubstring` and computes longest common substring alignments. It is easy to implement a different rigidity function. One simply has to extend the base class `RigidityFnc` which is provided by the library.

The lines 2 to 6 show the instantiation of an equation system which is of type `AntiUnifyProblem`. It is used in line 7 to instantiate a parser instance. The mentioned input sources are used to create one equation of two hedges, which is added to the equation system. One could add more equations to the system by just calling the method `parseHedgeEquation(in3, in4)` again, where `in3` and `in4` are input sources for the two additional hedges.

After specifying the rigidity function and parsing the equation system, the main algorithm `AntiUnify` is instantiated using this data (line 8). There is one additional argument, which specifies the debug level. For production usage we want to silently compute all the generalizations and process them by a callback function, which is defined in the lines 9 to 11. For debugging, one must also specify a print stream instead of `null` when invoking the algorithm at line 12. The callback function is invoked for each final state and each generalization variable from the input AUEs. The two arguments of the callback function, which are provided to the implementor, correspond to a certain

final state and variable. The first one is of type `AntiUnifySystem` and contains all the data which has been collected during the run: The substitution `getSigma`, the store `getStore` and some additional information. The second argument is the generalization variable. The computed generalization is the value which is associated with this variable in the substitution. Line 10 prints this generalization.

During the anti-unification process, fresh variables are introduced. They are named by a sequence number which is put between a prefix and a suffix. The counter for generating the number sequence is static and can be reset by calling the function `NodeFactory.resetCounter`. The prefix and the suffix for fresh term variables and also for fresh hedge variables can be specified by the user. Therefore the class `NodeFactory` offers four static variables, named `PREFIX_FreshTermVar`, `SUFFIX_Fresh TermVar`, `PREFIX_FreshHedgeVar` and `SUFFIX_FreshHedgeVar`.

4.3 Higher-Order Unranked Anti-Unification $2\mathcal{V}$

In this section we discuss an implementation of the algorithm $\mathfrak{G}_{\mathfrak{a}}^{2\mathcal{V}}$ from section 2.2. Recall the definition of the unranked terms, hedges, and contexts that have been considered there.

Definition 2.18 (Terms, hedges, contexts)**.** *Given pairwise disjoint countable sets of unranked function symbols $\mathcal{F}$ (symbols without fixed arity), hedge variables $\mathcal{V}_{\mathsf{H}}$, context variables $\mathcal{V}_{\mathsf{C}}$, and a special symbol $\circ$ (the hole), we define* terms, hedges, *and* contexts *by the following grammar:*

$$
\begin{array}{lcll}
t & ::= & f(\tilde{s}) & \textit{(term)} \\
s & ::= & t \mid \tilde{x} \mid \tilde{X}(\tilde{s}) & \textit{(hedge element)} \\
\tilde{s} & ::= & s_1, \ldots, s_n & \textit{(hedge)} \\
\tilde{c} & ::= & \tilde{s}_1, \circ, \tilde{s}_2 \mid \tilde{s}_1, f(\tilde{c}), \tilde{s}_2 \mid \tilde{s}_1, \tilde{X}(\tilde{c}), \tilde{s}_2 & \textit{(context)}
\end{array}
$$

where $f \in \mathcal{F}$, $\tilde{x} \in \mathcal{V}_{\mathsf{H}}$, $\tilde{X} \in \mathcal{V}_{\mathsf{C}}$, and $n \geqslant 0$.

In order to get a one-to-one correspondence between an admissible alignment and a generalization (modulo $\simeq$) we introduced the notion of a rigid hedge and restricted the computed generalizations to be rigid hedges. We recall the two definitions of rigid hedges and rigid generalizations from section 2.2.

Definition 2.27 (Rigid hedge)**.** *A hedge $\tilde{s}$ is* rigid *if the following conditions hold:*

1. *No context variable in $\tilde{s}$ applies to the empty hedge.*
2. *$\tilde{s}$ doesn't contain consecutive hedge variables.*
3. *$\tilde{s}$ doesn't contain vertical chains of (context)[†] variables.*
4. *$\tilde{s}$ doesn't contain context variables with a hedge variable as the first or the last argument (i.e., no subterms of the form $\tilde{X}(\tilde{x}, \ldots)$ and $\tilde{X}(\ldots, \tilde{x})$).*

[†]Vertical chains that consist of a context variable and a hedge variable are barred by item 4.

Definition 2.28 (Rigid generalization)**.** *Given two variable-disjoint hedges $\tilde{s}, \tilde{q}$ and their admissible alignment $\mathfrak{a}$, a rigid hedge $\tilde{g}$ is called a* rigid generalization *of $\tilde{s}$ and $\tilde{q}$ with respect to $\mathfrak{a}$, if $\tilde{g}$ is a supporting generalization of $\tilde{s}$ and $\tilde{q}$ with respect to $\mathfrak{a}$ so that:*

5. *There are substitutions σ, ϑ with $\tilde{g}\sigma = \tilde{s}$ and $\tilde{g}\vartheta = \tilde{q}$ such that all the contexts in σ and ϑ are singleton contexts.*

The implemented anti-unification algorithm solves the following problem:

Given: Two variable-disjoint hedges $\tilde{s}$ and $\tilde{q}$ and their admissible alignment $\mathfrak{a}$.

Find: A least general rigid generalization of $\tilde{s}$ and $\tilde{q}$ with respect to $\mathfrak{a}$.

For instance, $\tilde{X}(a, b)$ is a rigid generalization of $f(g(a, b, c))$ and (a, b) with respect to $a\langle 1{\cdot}1{\cdot}1, 1\rangle b\langle 1{\cdot}1{\cdot}2, 2\rangle$, while $\tilde{X}(a, b, \tilde{x})$ and $\tilde{X}(\tilde{Y}(a, b))$ are not.

Web page. The unranked higher-order anti-unification algorithm is available from `http://www.risc.jku.at/projects/stout/software/urauc.php`.

4.3.1 Explanation of the Web Interface for the Algorithm $\mathfrak{G}_{\mathfrak{a}}^{2\mathcal{V}}$

The input form of the Web page of unranked higher-order anti-unification consists of five rows, where the first, the fourth and the last row are equal to those of the unranked first-order anti-unification Web interface.

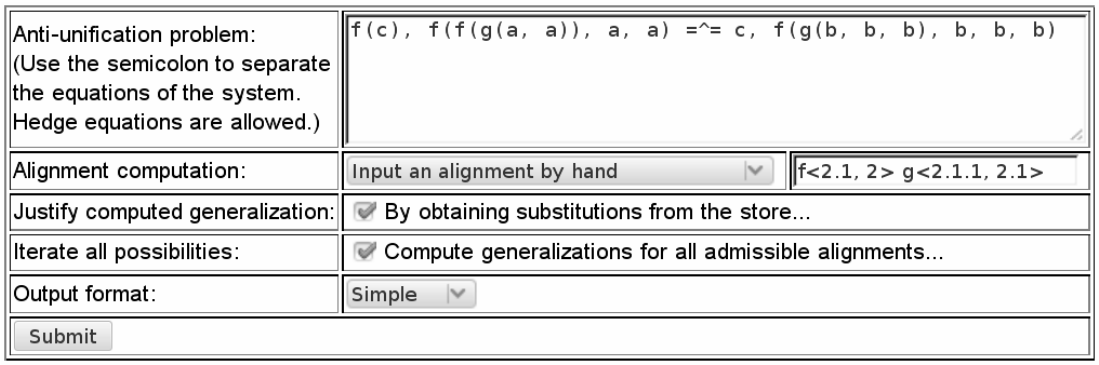

Figure 4.3: The input form of the Web presentation of the algorithm $\mathfrak{G}_{\mathfrak{a}}^{2\mathcal{V}}$.

In the second row, the alignment computation can be chosen. The only two possibilities are longest admissible alignments and the input of an alignment by hand. If the user selects the computation of longest admissible alignments, then the program automatically generates the set of all admissible alignments with maximum length, and the corresponding supporting generalizations are computed. Otherwise, the user has to specify an alignment in the input box next to the drop-down menu.

In the third row one can specify, whether or not to justify the computed generalization. For justification of a generalization $\tilde{g}$, the recorded differences of the input hedges $\tilde{s}, \tilde{q}$ are used to obtain two substitutions σ, ϑ. Then the program tests whether $\tilde{g}\sigma = \tilde{s}$ and $\tilde{g}\vartheta = \tilde{q}$ holds. The justification fails if this is not the case.

4.3.2 Implemented Transformation Strategy for $\mathfrak{G}_{\mathfrak{a}}^{2\mathcal{V}}$

Before we demonstrate the usage of the library on some Java code fragments, we give an overview of the strategy which is used in the implementation.

```
1  // INPUT:
2    E ← Given set of hedge equations {s̃1 ≐ q̃1, ..., s̃n ≐ q̃n}
3    A ← Given alignment computation function
4    H ← Given callback function
5  // COMPUTE:
6    For each equation s̃ ≐ q̃ in E
7      For each admissible alignment a in A(s̃, q̃)
8        State ← (P, S, σ) ← ({x̃: s̃ ≐ q̃; X̃: ∘ ≐ ∘; a}, ∅, {x̃ ↦ X̃(x̃)})
9        While State.P ≠ ∅
10         Fix AUE p_i ← x̃_i: s̃_i ≐ q̃_i; X̃_i: c̃_i ≐ d̃_i; a_i from State.P
11         If a_i = ε
12           Transform State ⟹^{p_i}_{Sol-H} State
13         Else If a_i = a1⟨j1, j2⟩ ... am⟨j1·Im, j2·Jm⟩ with j1, j2 ∈ ℕ
14           Transform State ⟹^{p_i}_{App-A} State
15         Else If a_i = a1⟨j·I1, J1⟩ ... am⟨j·Im, Jm⟩ with I1 ≠ ϵ
16           Transform State ⟹^{p_i}_{Abs-L} State
17         Else If a_i = a1⟨I1, j·J1⟩ ... am⟨Im, j·Jm⟩ with J1 ≠ ϵ
18           Transform State ⟹^{p_i}_{Abs-R} State
19         Else
20           Transform State ⟹^{p_i}_{Spl-H} State
21       For each p_i ← {x̃_i: s̃_i ≐ q̃_i; X̃_i: c̃_i ≐ d̃_i} in State.S
22         If |c̃_i| > 1 or |d̃_i| > 1
23           Transform State ⟹^{p_i}_{Res-C} State
24       For each p_i ← {x̃_i: s̃_i ≐ q̃_i; X̃: c̃_i ≐ d̃_i} in State.S
25         If s̃_i = q̃_i = ϵ_i & c̃_i = d̃_i = ∘
26           Transform State ⟹^{p_i}_{Clr-S} State
27       For each p_i ← {x̃_i: s̃_i ≐ q̃_i; X̃_i: c̃_i ≐ d̃_i} in State.S
28         For each p_j ← {x̃_j: s̃_j ≐ q̃_j; X̃_j: c̃_j ≐ d̃_j} in State.S
29           If p_i ≠ p_j & s̃_i = s̃_j & q̃_i = q̃_j & c̃_i = c̃_j & d̃_i = d̃_j
30             Transform State ⟹^{p_i,p_j}_{Mer-S} State
31 // CALLBACK:
32       H(State, x̃)
```

Listing 4.3: The implemented strategy for $\mathfrak{G}_{\mathfrak{a}}^{2\mathcal{V}}$.

The lines 1–4 in Listing 4.3 define the input assumptions. The alignment computation is a parameter of the algorithm and has to be provided by the user. Two prototype implementations are available from the library. For each AUE in the input set E, the alignment computation function will be invoked (line 6). Its return type is an `Iterator`, from which admissible alignments are pulled one by one (line 7). Line 8 shows how the initial system is constructed from one hedge equation and one alignment. The library

uses a customizable factory pattern, the so called class `NodeFactory`, to generate fresh variables, e.g. $\tilde{x}, \tilde{X}$.

Afterwards the rules Sol-H, App-A, Abs-L, Abs-R, and Spl-H are applied exhaustively, as shown at the lines 9–20. For each iteration, an arbitrary AUE from P is fixed and a rule which is applicable for this AUE is determined.

For the following transformation of the store (lines 21–30), the rules Res-C, Clr-S, and Mer-S, are applied exhaustively, one after another. (Note that Res-C might introduce empty AUEs. Therefore Clr-S applications have to follow.)

Line 31 shows how the callback function, which is provided by the user, is invoked, following an exhaustive system transformation. The user is free to process this result further as he/she needs. Obviously, `State.`σ applied to x gives the computed generalization, but also the store is available via `State.`S.

4.3.3 Using the Algorithm $\mathfrak{G}_{\mathfrak{a}}^{2\mathcal{V}}$ in Java

To embed $\mathfrak{G}_{\mathfrak{a}}^{2\mathcal{V}}$ into another Java program, the first thing to do is to create an `EquationSystem` of type `AntiUnifyProblem`. Listing 4.4 shows how to create a new equation system.

```
1 eqSys = new EquationSystem<AntiUnifyProblem>() {
2   public AntiUnifyProblem newEquation() {
3     return new AntiUnifyProblem();
4   }
5 };
```

Listing 4.4: Creation of an equation system of type AntiUnifyProblem.

After instantiating the equation system, some equations should be added to the empty system. A convenient way to do this is by using the class `InputParser`, as shown in Listing 4.5. It takes two arbitrary `Reader` instances, e.g. `in1` and `in2`, each representing one input hedge, creates a new `Equation`, and adds the equation to a given set of equations, e.g. `eqSys`. Depending on the context of the library integration, those hedges may also be composed manually.

```
1 new InputParser<>(eqSys).parseHedgeEquation(in1,in2);
```

Listing 4.5: Parsing two given hedges in1 and in2, and putting them into eqSys.

For invoking the main algorithm $\mathfrak{G}_{\mathfrak{a}}^{2\mathcal{V}}$, we assume that the variable `aFnc` is an alignment computation function. All alignment computation functions are of the abstract type `AlignFnc`. The library offers two such functions: The first one, called `AlignFncLAA`, computes longest admissible alignments (see section 2.2.8). The other one is `AlignFncInput` and can be used to specify a certain admissible alignment for one given anti-unification equation. `AlignFncInput` only works for a singleton equation system. Listing 4.6 shows how the anti-unification algorithm can be invoked using the built equation system.

```
1 new AntiUnify(aFnc, eqSys, DebugLevel.SILENT) {
2   public void callback(AntiUnifySystem sy, Variable x){
3     System.out.println(sy.getSigma().get(x));
```

```
4     }
5   }.antiUnify(true, false, null);
```

Listing 4.6: Sample code for calling the algorithm $\mathfrak{G}_{\mathfrak{a}}^{2\mathcal{V}}$.

The third argument of `AntiUnify` (line 1) defines the verbosity of the computation. For production use, normally one wants to silently compute the generalization. Whoever, if the debug level differs from `SILENT`, then also a `PrintStream` has to be provided in line 5 as third argument instead of `null`. By the first argument of the method `antiUnify`, the alignment iteration can be turned off, such that only one generalization is computed for each equation. The second argument in line 5 specifies, whether or not to justify the computed generalization. The justification works in the following way:

Let $\tilde{g}$ be a generalization of an input equation $\tilde{s} \triangleq \tilde{q}$ obtained by our anti-unification algorithm, and let S be the corresponding store. (In the demonstration code the store can be obtained by `sy.getStore()`.) The recorded differences in S are used to obtain two substitutions $\sigma_{\mathrm{L}}(S)$ and $\sigma_{\mathrm{R}}(S)$. The justification fails if either $\tilde{g}\sigma_{\mathrm{L}}(S) \neq \tilde{s}$ or $\tilde{g}\sigma_{\mathrm{R}}(S) \neq \tilde{q}$.

Last but not least, the lines 2–4 show a very simple implementation of a callback function, which prints the generalization $x\sigma$ to the standard output stream, for each generalization variable x which corresponds to one of the input equations.

4.4 Higher-Order Unranked Anti-Unification 4$\mathcal{V}$

Here, we aim at extending the algorithm $\mathfrak{G}_{\mathfrak{a}}^{2\mathcal{V}}$ with the two additional transformation rules Nar-FO and Nar-HO that introduce term variables and function variables. Before we discuss the implementation of the new rules, we recall the definition of the considered term language as well as the relaxed notion of a rigid hedge from section 2.3:

Definition 2.35 (Terms, hedges, contexts)**.** *Given pairwise disjoint countable sets of unranked function symbols $\mathcal{F}$ (symbols without fixed arity), term variables $\mathcal{V}_{\mathsf{T}}$, function variables $\mathcal{V}_{\mathsf{F}}$, hedge variables $\mathcal{V}_{\mathsf{H}}$, context variables $\mathcal{V}_{\mathsf{C}}$, and a special symbol $\circ$ (the hole), we define* terms*,* hedges*, and* contexts *by the following grammar:*

$$
\begin{array}{lll}
t & ::= & x \mid f(\tilde{s}) \mid F(\tilde{s}) \qquad \textit{(term)} \\
s & ::= & t \mid \tilde{x} \mid \tilde{X}(\tilde{s}) \qquad \textit{(hedge element)} \\
\tilde{s} & ::= & s_1, \ldots, s_n \qquad \textit{(hedge)} \\
\mathring{c} & ::= & f(\tilde{s}_1, \circ, \tilde{s}_2) \mid F(\tilde{s}_1, \circ, \tilde{s}_2) \qquad \textit{(bounded context)} \\
\tilde{c} & ::= & \tilde{s}_1, \circ, \tilde{s}_2 \mid \tilde{s}_1, f(\tilde{c}), \tilde{s}_2 \mid \tilde{s}_1, F(\tilde{c}), \tilde{s}_2 \mid \tilde{s}_1, \tilde{X}(\tilde{c}), \tilde{s}_2 \qquad \textit{(context)}
\end{array}
$$

where $f \in \mathcal{F}$, $x \in \mathcal{V}_{\mathsf{T}}$, $F \in \mathcal{V}_{\mathsf{F}}$, $\tilde{x} \in \mathcal{V}_{\mathsf{H}}$, $\tilde{X} \in \mathcal{V}_{\mathsf{C}}$, and $n \geqslant 0$.

Definition 2.38 (Rigid hedge)**.** *A hedge $\tilde{s}$ is* rigid *if the following conditions hold:*

1. *No higher-order variable in $\tilde{s}$ applies to the empty hedge.*
2. *If $\tilde{s}$ contains consecutive first-order variables, then both of them are term variables.*
3. *If $\tilde{s}$ contains a vertical chain of variables, then both of them are function variables.*
4. *$\tilde{s}$ doesn't contain higher-order variables with a first-order variable as the first or the last argument.*

The problem we intend to solve remains the same as in the previous section except for the new kinds of variables considered in the generalization.

4.4.1 Implementation of the Anti-Unification Algorithm $\mathfrak{G}_{\mathfrak{a}}^{4\mathcal{V}}$

First of all, we need to introduce term variables and hedge variables. For the sake of simplicity, we extend the class `HedgeVariable` which is provided by the library in order to obtain a class `TermVariable` that has the same behavior. Similarly, we extend the class `ContextVariable` in order to obtain a class `FunctionVariable` with the same behavior. Hence, `TermVariable` is a subclass of `HedgeVariable` and `FunctionVariable` is a subclass of `ContextVariable`. It follows that the implementation of the rule Mer-S works for all kinds of variables, as well as other classes (e.g., `Substitution`) which expect the variables to be of the types `HedgeVariable` and `ContextVariable`. One can restrict the usage of `TermVariable` and `FunctionVariable` by throwing an exception for misuse. This can be done by extending the class `Substitution` and checking for ill-typed mappings. Notice that this implementations of term variables and function variables are *already available* from the library.

Now we have to extend the anti-unification algorithm, that is, we need to extend the class `AntiUnifySystem`. It already offers two methods `narFO` and `narHO` for two rules Nar-FO and Nar-HO. They are dummy methods and need to be implemented. Recall the definitions of Nar-FO and Nar-HO:

Nar-FO: **Narrowing First-Order Variables**

$$P;\, \{\tilde{x}\colon (t_1,\ldots,t_n) \triangleq (s_1,\ldots,s_n);\, \tilde{X}\colon \circ \triangleq \circ\} \cup S;\, \sigma \implies$$
$$P;\, \{y_i\colon t_i \triangleq s_i;\, \tilde{Y}_i\colon \circ \triangleq \circ \mid 1 \leqslant i \leqslant n\} \cup S;\, \sigma\{\tilde{x} \mapsto (y_1,\ldots,y_n)\},$$

where $n \geqslant 1$ and y_i's, $\tilde{Y}_i$'s are fresh. t_i's and s_i's are terms.

Nar-HO: **Narrowing Higher-Order Variables**

$$P;\{\tilde{x}\colon \epsilon \triangleq \epsilon;\, \tilde{X}\colon \mathcal{H}_1(\tilde{s}_1,\ldots \mathcal{H}_n(\tilde{s}_n,\circ,\tilde{s}'_n)\ldots,\tilde{s}'_1) \triangleq \mathcal{G}_1(\tilde{q}_1,\ldots \mathcal{G}_n(\tilde{q}_n,\circ,\tilde{q}'_n)\ldots,\tilde{q}'_1)\} \cup S;\sigma \implies$$
$$P;\{\tilde{y}_i\colon \epsilon \triangleq \epsilon;\, F_i\colon \mathcal{H}_i(\tilde{s}_i,\circ,\tilde{s}'_i) \triangleq \mathcal{G}_i(\tilde{q}_i,\circ,\tilde{q}'_i) \mid 1 \leqslant i \leqslant n\} \cup S;\sigma\{\tilde{X} \mapsto F_1(\ldots F_n(\circ)\ldots)\},$$

where $n \geqslant 1$ and $\tilde{y}_i$'s, F_i's are fresh. $\mathcal{H}_i$'s and $\mathcal{G}_i$'s are from $\mathcal{F} \cup \mathcal{V}_{\mathsf{F}}$.

The class `AntiUnifySystem` already contains a Boolean field called `narrowVariables` to enable the two additional rules. If this variable is set to `true`, then, at the very end of the transformation sequence (line 31 in Listing 4.3), the two methods `narFO` and `narHO` are called. Both of them should return a Boolean value. If one methods returns `true`, then the rule Mer-S will be applied again exhaustively. Listing 4.7 shows the class `AntiUnifySystem4V` that extends the algorithm $\mathfrak{G}_{\mathfrak{a}}^{2\mathcal{V}}$.

```
1 public class AntiUnifySystem4V extends AntiUnifySystem {
2   private Substitution sigma;
3   public AntiUnifySystem4V(EquationSystem<AntiUnifyProblem>
4       problemSet, Substitution sigma) {
5     super(problemSet, sigma);
6     this.sigma = sigma;
7     narrowVariables = true;
8   }
```

```
 9    protected boolean narFO(List<VariableWithHedges> storeS,
10        DebugLevel debugLevel, PrintStream debugOut) {
11      boolean storeUpdated = false;
12      // TODO: implement Nar-FO
13      return storeUpdated;
14    }
15    protected boolean narHO(List<VariableWithHedges> storeQ,
16        DebugLevel debugLevel, PrintStream debugOut) {
17      boolean storeUpdated = false;
18      // TODO: implement Nar-HO
19      return storeUpdated;
20    }
21  }
```

Listing 4.7: Extending the algorithm $\mathfrak{G}_{\mathfrak{a}}^{2\mathcal{V}}$ in order to obtain $\mathfrak{G}_{\mathfrak{a}}^{4\mathcal{V}}$.

Before we turn to discussing the implementation for `narFO` and `narHO` we update the class `AntiUnify` which is the entry point of the anti-unification algorithm $\mathfrak{G}_{\mathfrak{a}}^{2\mathcal{V}}$. It should use the class `AntiUnifySystem4V` instead of `AntiUnifySystem`. This can be archived easily, since the instantiation of `AntiUnifySystem` may be overridden by extending the class `AntiUnify` and overriding the method `createSystem`. Listing 4.8 illustrates the implementation of the new entry point for the algorithm $\mathfrak{G}_{\mathfrak{a}}^{4\mathcal{V}}$.

```
1  public class AntiUnify4V extends AntiUnify {
2    public AntiUnify4V(AlignFnc aFnc, EquationSystem
3        <AntiUnifyProblem> sys, DebugLevel debugLevel) {
4      super(aFnc, sys, debugLevel);
5    }
6    protected AntiUnifySystem createSystem(
7        EquationSystem<AntiUnifyProblem> eqSys) {
8      return new AntiUnifySystem4V(eqSys,new Substitution());
9    }
10 }
```

Listing 4.8: Entry point for the algorithm $\mathfrak{G}_{\mathfrak{a}}^{4\mathcal{V}}$.

Recall that we mentioned the possibility of extending the class `Substitution` in order to check for ill-typed mappings. In line 8, the identity substitution σ is instantiated. This is the substitution which holds the generalization. An extended version can be passed to `AntiUnifySystem4V` easily.

Now we turn to implementing Nar-FO and Nar-HO. We do not discuss the two arguments `debugLevel` and `debugOut` because their sole purpose is to provide logging mechanisms. The argument `storeS` is a list of tuples of a hedge variable and two hedges. Similarly, `storeQ` is a list of tuples of a context variable and two contexts. For convenience reasons, the algorithm's store is split into two parts in the implementation.

In `narFO` we have to iterate over the `storeS`, check whether the length of both hedges is the same, and ensure that all the elements in both hedges are terms, i.e., there are neither hedge variables nor context variables. If this is the case, then we remove the current element from the list and create new AUEs with term variables. We add those

newly created AUEs to `storeS` and instantiate the hedge variable in `sigma` with the sequence of term variables. (Notice that `sigma` is a field of `AntiUnifySystem4V`.)

It remains to implement `narHO`. Again, one has to iterate over the store `storeQ` and check for applicability of the rule. The position of the hole of a context $\mathrm{Pos}_\circ(\tilde{c})$ can be obtained by calling the method $\tilde{c}$`.getHoleIdxRec()` on a certain context object $\tilde{c}$. This method returns an object of type `IntList` which is an integer sequence. With this information, it is trivial to check if the two holes appear at the same level at the contexts. For computational reasons, the sequence in `IntList` is in reverse order to the position of the hole.

```
 1 protected boolean narHO(List<VariableWithHedges> storeQ,
 2     DebugLevel debugLevel, PrintStream debugOut) {
 3   boolean updated = false;
 4   for (int i = storeQ.size() - 1; i >= 0; i--) {
 5     VariableWithHedges eq = storeQ.get(i);
 6     Hedge left  = eq.getLeft();
 7     Hedge right = eq.getRight();
 8     IntList idxL = left.getHoleIdxRec();
 9     IntList idxR = right.getHoleIdxRec();
10     if (idxL.size == idxR.size) {
11       boolean conditionOK = true;
12       for (int j = idxL.size; conditionOK && --j > 0;) {
13         TermNode leftTerm =  left.get((int)idxL.values[j]);
14         TermNode rightTerm= right.get((int)idxR.values[j]);
15         conditionOK =
16               !( leftTerm.getAtom() instanceof ContextVar)
17            && !(rightTerm.getAtom() instanceof ContextVar);
18         left  = leftTerm.getHedge();
19         right = rightTerm.getHedge();
20       }
21       if (conditionOK) {
22         // TODO: Decompose context
23         updated = true;
24       }
25     }
26   }
27   return updated;
28 }
```

Listing 4.9: Following the integer sequence to the hole.

Listing 4.9 shows the source code of checking the conditions for applicability of Nar-HO. In line 4 we iterate over the store so that removing and adding elements at the end of the list `storeQ` does not effect the iteration process. In the lines 8 and 9 we obtain the indexes of the holes in reverse order. If both are of the same length (line 10), then we check the following condition of the rule Nar-HO: $\mathcal{H}_i$'s and $\mathcal{G}_i$'s are from $\mathcal{F} \cup \mathcal{V}_\mathsf{F}$. This is done by the `for` loop in the lines 12–20. Decomposition (line 22) can be done in a similar manner, by using such a `for` loop. Afterwards the AUE that

has been decomposed can be removed from `storeQ`, new AUEs with function variables should be added, and the context variable has to be instantiated in the substitution `sigma` by a chain of function variables.

Merging of function variables and term variables is done automatically after returning `true` from `narFO` or `narHO`.

4.5 Anti-Unification for Nominal Terms

Before we turn to discussing the implementation, we recall the major definitions of permutations, nominal terms, freshness contexts, and $\mathcal{A}$-based terms-in-context. Under the assumption that the set of atoms permitted in generalizations is finite, there is a unique lgg modulo variable renaming and α-equivalence.

Definition 3.2 (Nominal term)**.** *Given disjoint sets of countably infinite* term variables $\mathcal{V}$[†]*, countably infinite* atoms $\mathcal{A}$[‡]*, and a signature* Σ*.*

A swapping $(\mathsf{a}\,\mathsf{b})$ *is a pair of atoms* $\mathsf{a}, \mathsf{b} \in \mathcal{A}$ *of the same sort. A* permutation *is a (possibly empty) sequence of swappings.* Nominal terms *are given by the grammar:*

$$t ::= f(t_1, \ldots, t_n) \mid \mathsf{a} \mid \mathsf{a}.t \mid \pi \cdot x$$

where f is an n-ary function symbol, a *is an atom,* π *is a permutation, and x is a variable. They are called respectively* application, atom, abstraction, *and* suspension.

A *freshness context* ∇ is a finite set of pairs of the form $\mathsf{a}\#x$ stating that the instantiation of x cannot contain free occurrences of a. A *term-in-context* is a pair $\langle \nabla, t \rangle$ of a freshness context ∇ and a term t. A term-in-context $\langle \nabla, t \rangle$ is *based* on a set of atoms $\mathcal{A}$, if all the atoms which occur in t and ∇ are elements of $\mathcal{A}$.

The implemented anti-unification algorithm solves the following problem:

Given: Two nominal terms t_1 and t_2 of the same sort, a freshness context ∇, and a *finite* set of atoms A such that $\langle \nabla, t_1 \rangle$ and $\langle \nabla, t_2 \rangle$ are based on A.

Find: A term-in-context $\langle \Gamma, t \rangle$ which is also based on A, such that $\langle \Gamma, t \rangle$ is a least general generalization of $\langle \nabla, t_1 \rangle$ and $\langle \nabla, t_2 \rangle$.

For instance, for $t_1 = f(b, a)$, $t_2 = f(X, (a\,b) \cdot X)$, $\nabla = \{b\#X\}$, and $A = \{a, b\}$, the algorithm computes the lgg of $\langle \nabla, t_1 \rangle$ and $\langle \nabla, t_2 \rangle$, which is $\langle \varnothing, f(Y, (a\,b) \cdot Y) \rangle$.

Web page. The nominal anti-unification algorithm is available from
`http://www.risc.jku.at/projects/stout/software/nau.php`.

4.5.1 Explanation of the Web Interface for the Algorithm $\mathfrak{G}_{\mathsf{N}}$

The input form of the Web interface to the nominal anti-unification algorithm consists of five rows shown below, where the first, the fourth and the fifth row are similar to the first, third and fifth explained in subsection 4.3.1.

[†]We assume that $\mathcal{V}$ contains countably infinite variables of each sort of atoms and sort of data.
[‡]We assume that $\mathcal{A}$ contains countably infinite atoms of each sort of atoms.

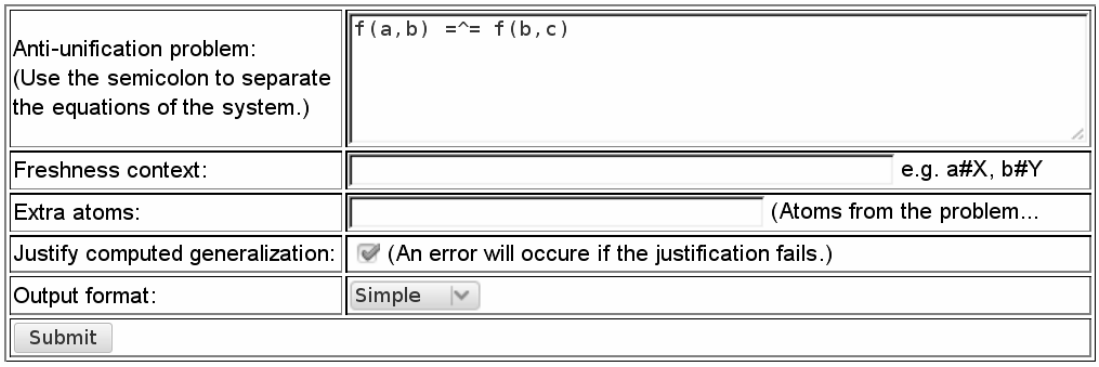

Figure 4.4: The input form of the Web presentation of the algorithm $\mathfrak{G}_{\mathsf{N}}$.

All the anti-unification equations share the same freshness context ∇, which can be specified in the second row. The computed term-in-context is a generalization of $\langle \nabla, t \rangle$ and $\langle \nabla, s \rangle$ for every anti-unification equation `t =^= s`.

As all the terms-in-context $\langle \nabla, t \rangle$ and $\langle \nabla, s \rangle$ obtained by anti-unification equations `t =^= s` have to be based on the same set of atoms A, all the atoms which appear in the anti-unification problem as well as those from ∇ are assumed to be elements of A. In the third row, the user may specify some additional atoms which are in A.

4.5.2 Implemented Transformation Strategy for $\mathfrak{G}_{\mathsf{N}}$

We illustrate the strategy that is used by the implementation in order to exhaustively transform a given set of AUEs P with respect to a given freshness context ∇ into a set of final states. A set $\mathcal{A}$ of additional atoms may be provided as input. Notice that all the atoms from P and ∇ are automatically collected. The implementation invokes a given callback function $\mathcal{H}$ as soon as a final state is reached.

```
1  // INPUT:
2    P ← Given set of AUEs
3    ∇ ← Given freshness context
4    A ← Given set of additional atoms
5    H ← Given callback function
6  // COMPUTE:
7    A' ← A(P) ∪ A(∇) ∪ A
8    V_P ← Set of generalization variables from P
9    State ← (P, S, Γ, σ) were S ← ∅ and Γ ← ∅ and σ ← Id
10   While State.P ≠ ∅
11     Fix AUE p ← x: t ≜ s from State.P
12     If Top(t) = Top(s) = a and a ∈ A' ∪ F
13       Transform State ⟹^p_Dec State
14     Else If t = a.t' and s = a.s'
15       Transform State ⟹^{p, a}_Abs State
16     Else If t = a.t', s = b.s', ∇ ⊢ c#a.t', ∇ ⊢ c#b.s', and c ∈ A'
17       Transform State ⟹^{p, c}_Abs State
```

```
18        Else
19          Transform State ⟹^p_Sol State
20      For each p1 in State.S
21        For each p2 in State.S\{p1}
22          π ← Compute permutation of p1 and p2
23          If π ≠ ⊥
24            Transform State ⟹^{p1,p2,π}_Mer State
25  // CALLBACK:
26      For each x in V_P
27        H(State, x);
```

Listing 4.10: The implemented strategy for $\mathfrak{G}_{\mathsf{N}}$.

The lines 1–5 in listing 4.10 define the input assumptions. All the atoms and generalization variables are collected in, respectively, line 7 and line 8. Those variables are later used in the CALLBACK: block (lines 25–27), which invokes the given callback function $\mathcal{H}$ for every collected variable x, such that $x\,\mathrm{State}.\sigma$ is the resulting generalization for the input AUE corresponding to x. Line 9 shows how the state is initialized. The first transformation iteration starts at line 10, where an arbitrary AUE is fixed (line 11) as long as there is an AUE in the problem set P. We know that, for a fixed AUE from P, either Abs, Dec, or Sol is applicable. The lines 12–19 show the entire procedure, which computes $\Longrightarrow^*_{\mathsf{Abs,Dec,Sol}}$ to transform the state so that $P = \varnothing$. The sole purpose of the statements in line 14–15 is to increase the performance of Abs. The lines 20–24 show the merging process $\Longrightarrow^*_{\mathsf{Mer}}$, which invokes the subalgorithm $\mathcal{E}$ for each pair of AUEs in the store S (line 22). The strategy used to perform the subalgorithm is demonstrated in subsection 4.5.4. It returns $\bot$ iff the permutation computation fails.

4.5.3 Using the Algorithm $\mathfrak{G}_{\mathsf{N}}$ in Java

To explain the library usage on a code example, we again assume the existence of two `Reader` instances `in1` and `in2` which contain the nominal terms to be generalized. Furthermore, we assume that there is a `Reader` instance `inA` for reading atoms and `inN` for the freshness context. Both of them are assumed to be comma separated sets, e.g., `inN = {a#X,b#Y,...}` and `inA = {c,d,...}`, where the braces are optional. The data source `inA` only specifies extra atoms, which do nor appear in `in1`, `in2` and `inN`.

```
1  final NodeFactory factory = new NodeFactory();
2  eqSys = new EquationSystem<AntiUnifyProblem>() {
3    public AntiUnifyProblem newEquation(NominalTerm t,
       NominalTerm s) {
4      return new AntiUnifyProblem(t, s, factory);
5    }
6  };
7  FreshnessCtx nablaIn = new InputParser(factory)
8      .parseEquationAndCtx(in1, in2, inA, inN, eqSys);
9  new AntiUnify(eqSys, nablaIn, DebugLevel.SILENT, factory) {
10   public void callback(AntiUnifySystem res, Variable var) {
11     System.out.println(res.getNablaGen());
```

```
12        System.out.println(res.getSigma().get(var));
13      }
14  }.antiUnify(false, null);
```

Listing 4.11: Usage example of the algorithm $\mathfrak{G}_{\mathsf{N}}$ in Java

In contrast to the other libraries, an instance of `NodeFactory` is needed, which we create in line 1. The lines 2 to 6 demonstrate the creation of an equation system.

All the input sources are parsed at the lines 7 and 8. The new equation is added to `eqSys` and the parsed freshness context is returned. Moreover, the factory instance remembers all the parsed atoms regardless of the input source they come from. More equations may be added `eqSys` by calling the method `parseEquation(in1, in2, eqSys)` from `InputParser`. Atoms and freshness contexts can also be parsed separately.

Line 11 shows that, additionally to the substitution and store, the generated freshness context is provided by the instance `res` of the class `AntiUnifySystem`.

Again, one can specify how fresh variables and fresh atoms are named. In contrast to the other three libraries, this functionality is implemented by private instance variables of `NodeFactory` and appropriate getter and setter methods.

4.5.4 Implementation of $\mathcal{E}$ for Deciding Equivariance

The nominal equivariance algorithm tests whether two terms differ from each other only by a permutation and a renaming of bound atom, i.e., if they are equivariant. Equivariance problem arises, for instance, in the course of generalization of the terms-in-context $p_1 = \langle \varnothing, f(a,b)\rangle$ and $p_2 = \langle \varnothing, f(b,c)\rangle$, where the atoms permitted in the generalization are a, b, and c, then the term-in-context $\langle \{c\#X, a\#Y\}, f(X,Y)\rangle$ generalizes p_1 and p_2, but it is not least general. To compute the latter, we need to reflect the fact that generalizations of the atoms are related to each other: One can be obtained from the other by the permutation $(b\,c)(c\,a)$. This leads to a least general generalization $\langle \{c\#X\}, f(X, (b\,c)(c\,a)\cdot X)\rangle$.

The equivariance decision algorithm solves the following problem:

Given: A set of equations of the form $t \sim s$, a freshness context ∇, and a finite set of atoms A such that all $\langle \nabla, t\rangle$ and $\langle \nabla, s\rangle$ are based on A.

Find: A permutation π of variables from A such that for all equations $t \sim s$, $\pi{\cdot}t$ is α-equivalent to s with respect to ∇, if such a π exists. Otherwise report failure.

For instance, in the example above, the permutation $(b\,c)(c\,a)$ was computed by the equivariance algorithm for $\{a \sim b, b \sim c\}$, $A = \{a, b, c\}$, and $\nabla = \varnothing$.

Web page. The equivariance decision algorithm is available from
`http://www.risc.jku.at/projects/stout/software/nequiv.php`.

4.5.4.1 Explanation of the Web Interface for the Algorithm $\mathcal{E}$

The input form is nearly the same as the one for nominal anti-unification. Figure 4.5 shows the Web input form.

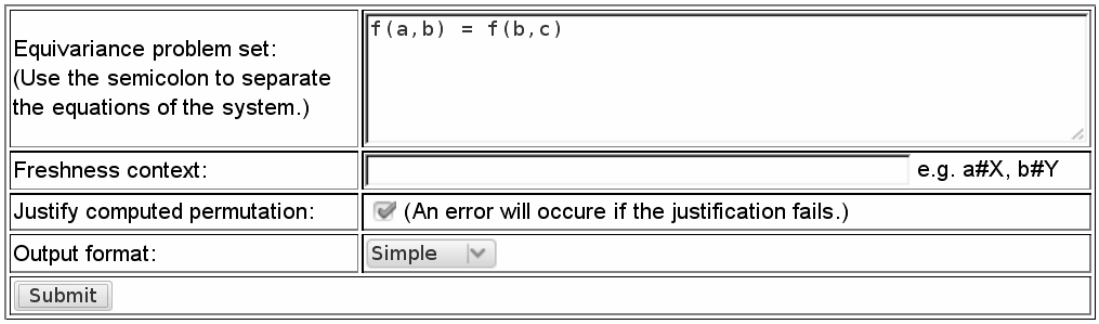

Figure 4.5: The input form of the Web presentation of the algorithm $\mathcal{E}$.

There are two differences: The row to specify extra atoms is missing, because the computed permutation must only permute atoms which appear in the problem set and further on, terms of an equivariance equation are separated by `=` instead of `=^=`.

4.5.4.2 Implemented Transformation Strategy for $\mathcal{E}$

We discuss the rule application strategy which is used by the implementation of $\mathcal{E}$. The input is a set of equivariance equations E and a freshness context ∇. It is split into two phases where the second phase starts at line 20 of Listing 4.12.

```
1  // INPUT:
2    E ← Given set of equivariance equations
3    ∇ ← Given freshness context
4  // COMPUTE:
5    E′ ← ∅ set of atom equations
6    State ← (E, ∇, 𝒜, π) were 𝒜 ← 𝒜(E) ∪ 𝒜(∇) and π ← Id
7    While State.E ≠ ∅
8      Fix equivariance problem p ← t ∼ s from State.E
9      If t = a.t′ and s = b.s′
10       Fix fresh atom ć ∉ 𝒜
11       Transform State ⟹^{p, ć}_{Alp-E} State
12     Else If Top(t) = Top(s) = f and f ∈ ℱ
13       Transform State ⟹^{p}_{Dec-E} State
14     Else If Top(t) = Top(s) = x and x ∈ 𝒱
15       Transform State ⟹^{p}_{Sus-E} State
16     Else If t = a and s = b
17       Move p from State.E to E′
18     Else
19       Return ⊥
20   State ← (E′, ∇, 𝒜, π)
21   While State.E′ ≠ ∅
22     Fix atom equation p ← a ∼ b from State.E′
23     If π•a = b
24       Transform State ⟹^{p}_{Rem-E} State
```

```
25        Else If π•a, b ∈ 𝒜
26          Transform State ⟹^p_Sol-E State
27        Else
28          Return ⊥
29      Return State.π
```

Listing 4.12: The implemented strategy for $\mathcal{E}$.

In line 5, an empty set of atom equations is initialized. This set is used for the second phase of the algorithm. In the first phase (line 9–15) the equivariance problems are reduced to atom equations. The latter ones are moved into a separate set, in the lines 16 and 17 of the first phase. If an equivariance equation is not an atom equation and none of the rules Alp-E, Dec-E, and Sus-E is applicable, then it is irreducible. Therefore, $\bot$ is returned in this case (line 18–19).

In line 20, the second phase is initialized with the set of atom equations that have been collected during phase one. Afterwards the rules Rem-E and Sol-E are applied exhaustively (line 23–26). If none of the two rules is applicable for an atom equation, then there is no appropriate permutation and $\bot$ is returned (line 27–28). If `State.`E' can be transformed into $\varnothing$, then the computation of π succeeded.

4.5.4.3 Using the Algorithm $\mathcal{E}$ in Java

We assume to have data sources for two nominal terms `in1` and `in2`, and another one for a freshness context, called `inN`, similarly to the nominal anti-unification algorithm. Moreover, we assume that an equation system `eqSys` has already been instantiated and that a `NodeFactory` instance, called `factory`, exists. We explain the usage of the library on the following code fragment:

```
1 InputParser parser = new InputParser(factory);
2 parser.parseEquation(in1, in2, eqSys);
3 FreshnessCtx nablaIn = parser.parseNabla(inN);
4 Collection<? extends Atom> atomSet = factory
5     .getAllByType(factory.classAtom);
6 Permutation pi = new Equivariance(eqSys, atomSet, nablaIn)
7     .compute(factory, false, DebugLevel.SILENT, null);
8 System.out.println(pi);
```

Listing 4.13: Usage example of the algorithm $\mathcal{E}$ in Java

In line 1 the parser instance is created, which afterwards is used to parse the equation and the freshness context from the input sources. The lines 4 and 5 demonstrate how one can obtain the collected set of atoms from the `NodeFactory` instance.

Later in line 6 this set is needed to instantiate a class named `Equivariance`, which encapsulates the computation of a permutation `pi`. The computation returns `null`, if no permutation exists for the input. The class `Permutation` contains two mappings from atoms to atoms (`Map<Atom, Atom>`): The permutation itself can be obtained by calling `getPerm` and the inverse permutation, which can be obtained by `getInverse`. Furthermore the class `Permutation` provides some methods to work with permutations and swappings.

4.6 Anti-Unification for Lambda Terms

We demonstrate the implementation of the algorithm $\mathfrak{G}_\mathsf{P}$ from section 3.2 which works on simply-typed λ-terms: It takes as input two such terms of the same type, in η-long β-normal form, and returns their least general pattern generalization. Patterns here mean higher-order patterns à la Miller [59]. The input terms are not required to be patterns. Again, we start with recalling the most important definitions.

Definition 3.22 (λ-terms)**.** *Given a countably infinite set of* term variables $\mathcal{V}^\dagger$ *and a signature* Σ*.* λ-terms *are built using the grammar*

$$t ::= x \mid c \mid \lambda x.t \mid t_1\ t_2$$

where x *is a variable and* c *is a constant.*

Terms like $(\ldots(h\ t_1)\ldots t_m)$ are written as $h(t_1,\ldots,t_m)$ and terms of the form $\lambda x_1.\cdots.\lambda x_n.t$ as $\lambda x_1,\ldots,x_n.t$.

Definition 3.26 (Higher-order pattern)**.** *A* higher-order pattern *is a* λ*-term where, when written in* η*-long* β*-normal form, all free variable occurrences are applied to lists of pairwise distinct (*η*-long forms of) bound variables.*

The implemented algorithm solves the following problem:

Given: Higher-order terms t_1 and t_2 of the same type in η-long β-normal form.

Find: A least general higher-order pattern generalization of t_1 and t_2.

For instance, if $t_1 = \lambda x, y.f(h(x,x,y), h(x,y,y))$ and $t_2 = \lambda x, y.f(g(x,x,y), g(x,y,y))$, then $t = \lambda x, y.f(X(x,y), Y(x,y))$ is a higher-order pattern lgg of t_1 and t_2.

Web page. The implementation of the anti-unification algorithm is available from `http://www.risc.jku.at/projects/stout/software/hoau.php`.

4.6.1 Explanation of the Web Interface for the Algorithm $\mathfrak{G}_\mathsf{P}$

The implementation slightly differs from the theoretical algorithm: In addition to simply-typed terms, it can also take untyped input. It has an advantage that the user does not necessarily have to supply types, but has a disadvantage that the terms may not be typeable or normalizable. Figure 4.6 shows the input form of the Web interface that consists of four rows.

In the first row, the anti-unification problem should be given. The problems consist of one or more anti-unification equations, separated by semicolon. Each such equation consists of two λ-terms, with `=^=` in between. The backslash \ is used instead of λ.

In the second row, the maximum recursion depth of the β-reduction can be specified. This is to avoid infinite chain of reductions for terms like $(\lambda x.(x\ x))(\lambda x.(x\ x))$.

As in subsection 4.3.1, one can choose to justify the computed lgg in the third row.

In the last row, the output format can be specified. One can choose form a drop-down box between `simple`, `verbose`, `progress`, and `progress-origin`. The first three

[†]We assume that $\mathcal{V}$ contains countably infinite term variables of each type.

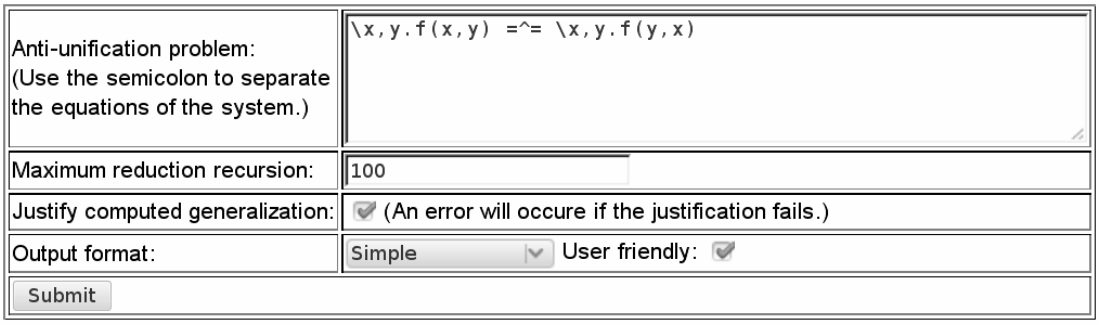

Figure 4.6: The input form of the Web presentation of the algorithm $\mathfrak{G}_{\mathsf{P}}$.

of them are like those described in subsection 4.2.1. By choosing the output format `progress-origin`, all the debug information will be shown to the user, but the original names of bound variables are used. This is useful for debugging, as all the bound variables are renamed by the parser, giving them unique names.

4.6.2 Implemented Transformation Strategy for $\mathfrak{G}_{\mathsf{P}}$

Before we demonstrate the usage of the library on some Java code fragments, we give an overview of the strategy which is used in the implementation. The implementation is parametric by a callback function $\mathcal{H}$ that is invoked when the final state is reached.

```
1  // INPUT:
2    P ← Given set of AUEs
3    H ← Given callback function
4  // COMPUTE:
5    V_P ← Set of generalization variables from P
6    State ← (P, S, σ) were S ← ∅ and σ ← ∅
7    While State.P ≠ ∅
8      Fix AUE p ← X(x⃗) : t ≟ s from State.P
9      If t = λy.t' or s = λz.s'
10       Transform State ⟹^p_Abs State
11     Else If Top(t) = Top(s)
12       Transform State ⟹^p_Dec State
13     Else
14       Transform State ⟹^p_Sol State
15   For each p1 in State.S
16     For each p2 in State.S\{p1}
17       π ← Compute permuting matcher of p1 and p2
18       If π ≠ ⊥
19         Transform State ⟹^{p1,p2,π}_Mer State
20 // CALLBACK:
21   For each X in V_P
22     H(State, X);
```

Listing 4.14: The implemented strategy for $\mathfrak{G}_{\mathsf{P}}$.

The lines 1–3 in listing 4.14 define the input assumptions. In line 5, the generalization variables from all the input AUEs are collected. Those variables are later used in the CALLBACK: block (lines 20–22), which invokes the given callback function $\mathcal{H}$ for every collected variable X, such that $X\,\mathrm{State}.\sigma$ is the resulting generalization for the input AUE corresponding to X. Line 6 shows how the state is initialized. The first transformation iteration starts at line 7, where an arbitrary AUE is fixed (line 8) as long as there is an AUE in the problem set P. We know that, for a fixed AUE from P, either Abs, Dec, or Sol is applicable. The lines 7–14 show the entire procedure, which computes $\Longrightarrow^*_{\mathsf{Abs,Dec,Sol}}$ to exhaustively transform the state. The lines 15–19 show the merging process $\Longrightarrow^*_{\mathsf{Mer}}$, which invokes the subalgorithm $\mathcal{M}$ for each pair of AUEs in the store S (line 17). The strategy used to perform the subalgorithm is demonstrated in subsection 4.6.4. It returns $\bot$ iff the merging fails, or a mapping $\pi : D \to R$ otherwise.

4.6.3 Using the Algorithm $\mathfrak{G}_\mathsf{P}$ in Java

To embed $\mathfrak{G}_\mathsf{P}$ or $\mathcal{M}$ into another Java program, the first thing to do is to create an `EquationSystem`. The class `EquationSystem` is implemented in a generic way and it is used to represent either a set of AUEs or a set of matching problems. An AUE is represented by the Java class `AntiUnifyProblem` and a matching problem by a class named `PermEquivProblem`. Listing 4.15 shows how to create a new equation system of a certain `TYPE`.

```
1 EquationSystem<TYPE> eqSys = new EquationSystem<TYPE>(){
2   public TYPE newEquation() {
3     return new TYPE();
4   } };
```

Listing 4.15: Creation of an equation system of a certain TYPE.

By replacing `TYPE` with `PermEquivProblem` or `AntiUnifyProblem` an equation system of appropriate type may be instantiated. One more thing to mention is, that the standard implementation of `AntiUnifyProblem` automatically retrieves a unique generalization variable for each instance. The user may extend the class `AntiUnifyProblem` to create a new class with different behavior, and use her/his own type instead of `TYPE`.

After instantiating an equation system of appropriate type, some equations of the respective type can be added to the empty system. A convenient way to do this is by using the class `InputParser`, as shown in listing 4.16.

```
new InputParser().parseEquation(in1, in2, eqSys, 1000);
```

Listing 4.16: Parsing two given λ-terms in1 and in2, and putting them into eqSys.

We assume that there are two data sources `in1` and `in2` available in form of `Reader` instances, each of them containing one of the input λ-terms. The implementation does neither require the input terms to be typed, nor to be in β-normal form. During the parsing process β-reduction is performed exhaustively. To avoid infinite chain of reductions for terms like $(\lambda x.(x\ x))(\lambda x.(x\ x))$, the fourth argument defines an upper bound, to limit the recursive calls of β-reduction. Listing 4.17 shows how the anti-unification algorithm $\mathfrak{G}_\mathsf{P}$ can be invoked using the built equation system.

```
1 new AntiUnify(eqSys, 1000, DebugLevel.SILENT) {
2   public void callback(AntiUnifySystem res, Variable var) {
3     System.out.println(res.getSigma().get(var));
4   } }.antiUnify(false, null);
```

Listing 4.17: Sample code for calling the algorithm $\mathfrak{G}_\mathsf{P}$.

Like above, the second argument of `AntiUnify` (line 1) restricts the number of recursions for β-reduction. As, by our rule definitions, β-reduction only performs variable renaming, the upper bound should never be reached, if the input is already in β-normal form. The third argument in line 1 defines the verbosity of the computation. For production use, normally one wants to silently compute the generalization. Whoever, if the debug level differs from `SILENT`, then also a `PrintStream` has to be provided in line 4 as second argument instead of `null`. The first argument in line 4 specifies, whether or not to justify the computed generalization. For justification of a generalization r, the recorded differences of the input terms s, t are used to obtain two substitutions ϑ, φ, respectively. The justification fails if either $r\vartheta \neq_\alpha s$ or $r\varphi \neq_\alpha t$. Last but not least, the lines 2 and 3 show a very simple implementation of a callback function, which prints the generalization $x\sigma$ to the standard output stream, for each generalization variable x which corresponds to one of the input AUEs.

4.6.4 Implementation of $\mathcal{M}$ for Computing Permuting Matchers

The algorithm $\mathfrak{G}_\mathsf{P}$ requires to decide the existence problem of a variable renaming for two given input terms so that the terms become alpha equivalent. It is needed to ensure that the computed generalization is least general. Such a problem arises, e.g., in the course of generalization of the terms $t_1 = \lambda x, y, z.f(x(y, z), x(z, y))$ and $t_2 = \lambda x, y, z.f(X(y, \lambda u.u), X(z, \lambda v.v))$. To see if the same variable can be used in the generalization of the arguments of t_1 and t_2, we have to check whether there exists a variable renaming π such that $x(y, z)\pi = x(z, y)$ and $X(y, \lambda u.u)\pi \triangleq X(z, \lambda v.v)$.

The algorithm that performs such a test is integrated in the implementation of $\mathfrak{G}_\mathsf{P}$, but we provide access to it separately as well, due to the fact that the problem is interesting, may appear in various contexts, and having a tool to solve it is useful.

The algorithm solves the following problem:

Given: A set of equations of the form $t \sim s$ where t and s are λ-terms, and two sets of variables, the domain D and the range R.

Find: A variable renaming $\pi\colon D \to R$, such that $t\pi$ is α-equivalent to s for all equations $t \sim s$, if it exists. Otherwise report failure.

The generalization problem for t_1 and t_2 above creates the set of equations $\{x(y, z) \sim x(z, y), X(y, \lambda u.u) \sim X(z, \lambda v.v)\}$, the domain $D = \{x, y, z\}$ and the range $R = \{x, y, z\}$. Then the decision algorithm $\mathcal{M}$ constructs and returns the mapping $\pi = \{x \mapsto x, y \mapsto z, z \mapsto y\}$. Afterwards, this renaming can be used to answer the original question of generalization of t_1 and t_2, obtaining the lgg $\lambda x, y, z.f(Y(x, y, z), Y(x, z, y))$ where, indeed, the variable Y appears twice.

Web page. The permuting matcher decision algorithm is available from `http://www.risc.jku.at/projects/stout/software/hoequiv.php`.

4.6.4.1 Explanation of the Web Interface for the Algorithm $\mathcal{M}$

The input form of the Web interface to the permuting matcher algorithm consists of four rows shown below, like illustrated in Figure 4.7.

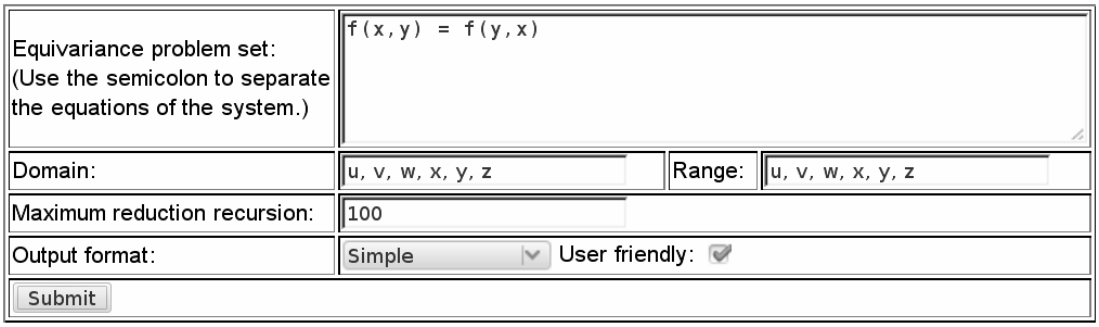

Figure 4.7: The input form of the Web presentation of the algorithm $\mathcal{M}$.

The first, the third and the fourth row are equivalent, respectively, to the first, the second and the fourth ones in the interface of $\mathfrak{G}_{\mathsf{P}}$, described above. (The terms of an equivariance equation are separated by `=` instead of `=^=`.) In the second row, the two sets of variables which specify the domain and the range should be given.

4.6.4.2 Implemented Transformation Strategy for $\mathcal{M}$

In contrast to the web interface, where the user may enter an arbitrary (untyped) lambda term that should be reduced to β-normal form if possible, here the input is assumed to be a lambda term which is already in β-normal form. Therefore, the argument that defines the maximum recursion depth is not needed.

```
1  // INPUT:
2    D ← Given set of domain variables
3    R ← Given set of range variables
4    M ← Given set of matching problems
5  // COMPUTE:
6    State ← (D, R, M, ρ, π) were ρ ← ∅ and π ← ∅
7    While State.M ≠ ∅
8      Fix matching problem p ← t ~ s from State.P
9      If t = λy.t' or s = λz.s'
10       Transform State ⟹^p_Abs-M State
11     Else If Top(t) ∈ D and Top(s) ∈ R
12       Transform State ⟹^p_Per-M State
13     Else If Top(t) ∉ D and Top(s) ∉ R and Top(t)π = Top(s)ρ
14       Transform State ⟹^p_Dec-M State
15     Else
16       Return ⊥
17   Return State.π
```

Listing 4.18: The implemented strategy for $\mathcal{M}$.

Like in listing 4.14, the block INPUT: of listing 4.18 defines the input assumptions. In line 6 the state is initialized. Afterwards the rules are applied exhaustively. If no rule is applicable to a certain matching problem, then a permuting matcher does not exist and $\perp$ is returned. In the Java implementation $\perp$ is represented by `null`. If `State.`M can be transformed into $\varnothing$, then the final state has been reached and the implementation returns the mapping `State.`π

4.6.4.3 Using the Algorithm $\mathcal{M}$ in Java

We explain the usage on a code fragment and assume that there are two data sources `in1` and `in2` available in form of `Reader` instances, each of them contains one of the λ-terms. There is also an integer variable `maxReduce` which specifies the maximum recursion depth of β-reduction.

To use the algorithm $\mathcal{M}$ separately, the sets of domain and range variables must be provided in addition to the equation system. This is an easy task and may be done as shown in listing 4.19 lines 1–4.

```
1 Set<Variable> dom = DataStructureFactory.$.newSet();
2 Set<Variable> ran = DataStructureFactory.$.newSet();
3 dom.add(new Variable("x", null));
4 ...
5 Map<Variable,Variable> pi = new PermEquiv(eqSys, dom, ran)
6    .compute(DebugLevel.SILENT, null);
7 System.out.println(pi);
```

Listing 4.19: Usage example of the algorithm $\mathcal{M}$ in Java

The second argument of the `Variable`-constructor specifies the type of the variable. (`null` is used for untyped variables.) We assume that the set of equations `eqSys` exists (e.g., it can be created like illustrated in Listing 4.15). In the lines 4 and 5, the main algorithm `PermEquiv` is invoked. In case of failure `null` is returned and in the success case `pi` is a (possibly empty) mapping.

Chapter 5

Conclusion

This thesis discussed the anti-unification problem for various term languages. We developed algorithms that solve such problems, illustrated their usage on some examples, analyzed their properties, and studied their implementation. The work contributes in the following three major ways to the community:

- Anti-unification algorithms for unranked terms, simply-typed λ-terms, and nominal terms have been designed. Some of them require to constructively solve a decision problem in order to merge variables in the generalization. (Merging of variables is required to compute generalizations that are *least general*.) We also designed subalgorithms that solve those problems. We proved the properties and the complexity bounds of all the developed algorithms, which all are polynomial in time and space.
- We showed that under certain conditions the matching problem and the joinability problem can be solved by a *coherent* anti-unification algorithm. The algorithms discussed in this work compute either a unique lgg for two input terms or they solve the matching problem themselves so that the minimization step can be performed easily. This has the advantage that, when implementing our algorithms, one does not need to implement an extra algorithm that solves the matching problem.
- We implemented all the discussed algorithms in Java and developed a robust and well engineered open-source software library that is freely available online. We discussed the strategies that are used in our implementations and illustrated how those algorithms can be integrated in users projects.

The higher-order anti-unification problem for unranked terms is highly nondeterministic. Therefore the computation has been split into two parts, the skeleton computation, and the computation of a generalization with respect to a given skeleton. It turned out that, by imposing a few natural restrictions, the generalization becomes unique for a given skeleton, and we proved that it can be computed in quadratic time. Since the skeleton for a given pair of hedges is not unique, a minimization step is needed. However, the developed algorithm can solve the matching problem as needed to perform the minimization. The anti-unification algorithms for unranked terms are *self-minimizing*.

Even though, the generalization problem we consider for nominal terms belongs to the first-order setting, we showed that without any restrictions, a minimal complete set of generalizations does not always exist. Therefore we suggest a restriction under which the problem becomes unitary.

For simply-typed λ-terms, we compute so called higher-order pattern generalizations. This setting imposes restricted higher-order power and yields a single unique generalization for two simply-typed λ-terms. The algorithm can be used to develop refactoring and clone detection techniques for languages based on λProlog.

5.1 Discussion of Future Research Directions

Naturally, our results give rise to various research directions that may be considered in the future. In order to discuss some of the possible directions for future research, we illustrate in Figure 5.1 the subsumption hierarchy of various anti-unification (AU) problems. The rectangular nodes indicate problems that have not yet been studied. At the bottom left side of Figure 5.1 there is the syntactic first-order case (SYN-AU). Following the arrows, it can be generalized, by syntactic higher-order AU (SYN-AU^2), or by order-sorted first-order AU (OS-AU). Both of the latter ones are subsumed by order-sorted higher-order AU (OS-AU^2). On the other side there is the word anti-unification problem (W-AU). It generalizes to unranked fist-order AU (UR-AU), which itself is subsumed by unranked higher-order AU (UR-AU^2). Unranked and order-sorted theories can be generalized by regular-expresion order-sorted (REOS) theories, for both, the first-order case (REOS-AU), and the higher-order case (REOS-AU^2).

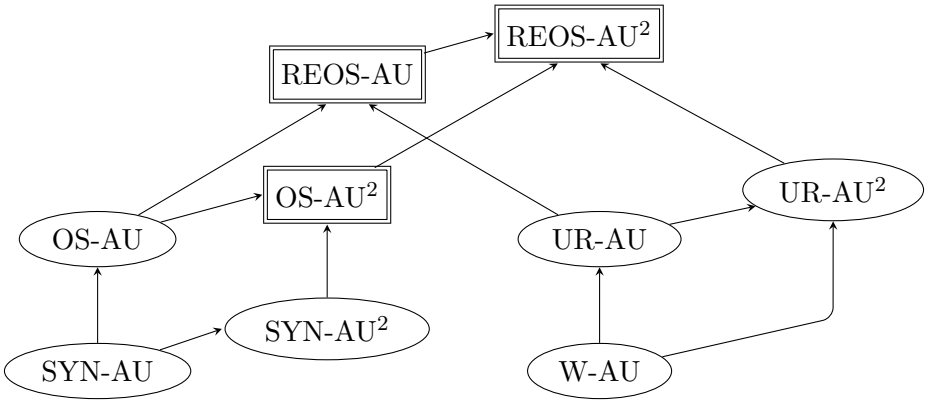

Figure 5.1: Subsumption hierarchy of various anti-unification problems.

As illustrated in Figure 5.1, there are some cases of the anti-unification problem that have not been investigated yet (the rectangular nodes). For the dual unification problem, the first-order regular-expresion order-sorted case has been studied in [52]. First, those results are interesting from the theoretical point of view, as they give new insights in the theory itself and establish connections between other theoretical results. Second, implementations of AU algorithms might be generalized by exploiting this new insights. It is known, that abstraction and reusability greatly encourage stability and maintainability of software packages. Therefore the homogenization of various theories is of great value for the community, from both points of view, the theoretical one, and the application point of view.

This leads to one possible direction for future research: the investigation of the anti-unification problem in the theories REOS-AU and REOS-AU^2. Since the additional typing information in the REOS theories can be seen as a specialization of the unranked theories, it seems natural to try the following approach:

1. Cut off all the sort information from the input terms.
2. Run one of the algorithms UR-AU or UR-AU^2, to obtain an unranked generalization term of most general sort.
3. Specialize the sort information towards the input terms as long as possible.

As the computational behavior of this modular approach cannot be foreseen, it might also be interesting to develop a more direct algorithm.

Another direction for future research within this area could be the investigation of the above AU problems modulo some equational theories. In contrast to the dual unification problem, there is little work on anti-unification modulo theories. One of the rare works considers the theory of OS-AU modulo associativity, commutativity, and unity [4]. There are actually some application areas, where UR-AU modulo theories are requested. For this reason, investigating UR-AU modulo commutativity would be of great value. (Associativity and unity is trivial in this theory.)

Besides developing new algorithms, also the extraction of well-behaving fragments is important from the application point of view. This becomes more and more important, as the size of the data increases. For instance, one could try to apply a fragment of our novel UR-AU^2 algorithm (which runs in $O(n^2)$ time in the worst case, where n is the size of the input,) to Big-Data, while other fragments might not be applicable because of their poor runtime complexity. Since the skeleton computation function is the parameter of our algorithm, studying and experimenting with different skeleton computation functions could also be work for the future. There is strong evidence which suggests that there are even better behaving fragments of UR-AU^2, where the runtime is bound by $O(n \log(n))$ in the worst case [67].

A different direction for future research would be to consider the anti-unification problem for *permissive* nominal logic. We suspect that for permissive nominal terms (PNT), one can use our higher-order anti-unification algorithm from section 3.2 after translating PNTs into higher-order pattern. The back-translation from higher-order pattern to PNTs might be easier than the translation back to the nominal terms we considered in section 3.1.

Yet another direction for future research could be the introduction of skeletons for λ-terms such that the skeleton computation can be decoupled from the computation of the generalization, like done in section 2.2. This approach could lead to more accurate results than restricting generalizations to be higher-order pattern. Then, one could also think about considering a richer type theory like the calculus of constructions, allowing higher-order types.

To conclude the discussion about future research directions, we would like to emphasize that the possibilities within any research area are infinite. This discussion might be seen as motivation for some possible small steps towards a certain direction.

List of Figures

Listings

Notations

$\preceq_{\equiv}$	instantiation quasi ordering modulo $\equiv$
$\preceq$	instantiation quasi ordering where the concrete instance of $\equiv$ is either unimportant or clear from the context
$\simeq_{\equiv}$	equivalence relation induced by $\preceq_{\equiv}$
$\simeq$	equivalence relation induced by $\preceq$
$\prec_{\equiv}$	$\preceq_{\equiv}$ minus $\simeq_{\equiv}$
$\prec$	$\preceq$ minus $\simeq$
$=_{\alpha}$	alpha equivalence
$\#$	freshness predicate
i, j	integer (position)
I, J	string of positive integers (position)
ϵ	empty sequence (position or hedge)
a, b, c, d, e, f, g, h	function symbol
$\mathsf{a}, \mathsf{b}, \mathsf{c}, \mathsf{d}, \mathsf{e}$	atom
x, y, z, u, X, Y, Z, U	term variable
$\vec{x}, \vec{y}, \vec{z}$	sequence of term variables
$\tilde{x}, \tilde{y}, \tilde{z}$	hedge variable
F, G, H	function variable
$\tilde{X}, \tilde{Y}, \tilde{Z}, \tilde{U}$	context variable
ξ	first-order variable (term variable or hedge variable)
Ξ	higher-order variable (function variable or context variable)
$\mathcal{H}, \mathcal{G}$	function symbol or function variable
$\hbar, \mathcal{g}$	function symbol or higher-order variable
s	function symbol or (any type of) variable
$\hbar$	function symbol or atom
λ	abstraction, i.e., a variable binding
ν, δ	basic sorts, i.e., basic types
τ	composite sorts, i.e., composite types
$\mathcal{F}$	set of unranked function symbols
Σ	Signature of types and ranked function symbols
$\mathcal{A}$	set of atoms
$\mathcal{V}$	set of all variables
$\mathcal{V}_{\mathsf{T}}$	set of term variables (individual variables)
$\mathcal{V}_{\mathsf{H}}$	set of hedge variables
$\mathcal{V}_{\mathsf{F}}$	set of function variables
$\mathcal{V}_{\mathsf{C}}$	set of context variables
$\mathcal{T}(\mathcal{F}, \mathcal{V})$	set of hedges constructed over $\mathcal{F}$ and $\mathcal{V}$
$\mathcal{T}(\Sigma, \mathcal{A}, \mathcal{V})$	set of nominal terms constructed over $\Sigma, \mathcal{A}$, and $\mathcal{V}$
$\mathcal{T}$	set of terms where the concrete instances of basic components

	are either unimportant or clear from the context
$\mathcal{V}(\cdot)$	set of all occurring variables
$\mathcal{V}_{\mathsf{T}}(\cdot)$	set of occurring term variables
$\mathcal{V}_{\mathsf{H}}(\cdot)$	set of occurring hedge variables
$\mathcal{V}_{\mathsf{F}}(\cdot)$	set of occurring function variables
$\mathcal{V}_{\mathsf{C}}(\cdot)$	set of occurring context variables
$\mathcal{A}(\cdot)$	set of occurring atoms
t, s, u, q, r	term
$\tilde{s}, \tilde{q}, \tilde{r}, \tilde{g}, \tilde{h}$	hedge
$\circ$	hole (unit context)
$\tilde{c}, \tilde{d}$	context (hedge with hole)
$\mathring{c}, \mathring{d}$	bounded context
$\dot{c}, \dot{d}$	singleton context
π, ρ, μ, τ	permutation of atoms
∇, Γ	freshness context
term-in-context	pair of a freshness context and a term, written $\langle \nabla, t \rangle$
p	term-in-context
$\sigma, \vartheta, \varphi$	substitution
π, ρ, μ	mapping from variables to variables
Id	identity (e.g., substitution, permutation, mapping)
w	string of symbols (word)
$\lvert \cdot \rvert$	cardinality of a set or length of a sequence
$\lVert \cdot \rVert$	size of a term, hedge, or context
$\lVert \cdot \rVert_{\mathsf{Abs}}$	number of abstractions in a term
$\tilde{s}\lvert_I$	term at position I in a hedge $\tilde{s}$
$\tilde{s}\lvert_i^j$	subhedge of $\tilde{s}$ from position i to j (inclusive $\tilde{s}\lvert_i$ and $\tilde{s}\lvert_j$)
$\mathrm{Top}(\cdot)$	string of top symbols (symbols at level 1)
$\mathrm{Pos}(\cdot)$	set of all positions of a term or hedge
$\mathrm{Pos}_s(\cdot)$	set of positions of occurrences of s in a term or hedge
$\mathrm{Depth}(\cdot)$	depth of a term
$\mathrm{Dom}(\cdot)$	domain of a substitution
$\mathrm{Ran}(\cdot)$	range of a substitution
$\mathrm{FV}(\cdot)$	free variables
$\mathrm{FA}(\cdot)$	free atoms
$\mathrm{FA}^{-\mathrm{s}}(\cdot)$	free atoms without considering suspensions
$\mathrm{Fresh}(\cdot)$	subset of fresh atoms
$\mathfrak{a}$	alignment
$\mathfrak{e}$	empty alignment
$\mathcal{R}$	rigidity function
$\mathcal{A}$	higher-order alignment computation function
$\mathcal{H}$	callback function
$\mathcal{C}s$	set of alignments of a certain length
$\mathcal{C}t\mathcal{X}$	minimal freshness context or $\perp$ if it does not exist
$\mathcal{E}$	algorithm to compute equivariance
$\mathcal{M}$	algorithm to compute permuting matcher

$\mathfrak{G}_{\mathcal{R}}$	anti-unification algorithm for hedges $\mathcal{T}(\mathcal{F}, \mathcal{V}_{\mathsf{H}})$
$\mathfrak{G}_{\mathfrak{a}}^{2\mathcal{V}}$	anti-unification algorithm for hedges $\mathcal{T}(\mathcal{F}, \mathcal{V}_{\mathsf{H}} \cup \mathcal{V}_{\mathsf{C}})$
$\mathfrak{G}_{\mathfrak{a}}^{4\mathcal{V}}$	anti-unification algorithm for hedges $\mathcal{T}(\mathcal{F}, \mathcal{V}_{\mathsf{H}} \cup \mathcal{V}_{\mathsf{C}} \cup \mathcal{V}_{\mathsf{T}} \cup \mathcal{V}_{\mathsf{F}})$
$\mathfrak{G}_{\mathsf{N}}$	anti-unification algorithm for nominal terms $\mathcal{T}(\Sigma, \mathcal{A}, \mathcal{V})$
$\mathfrak{G}_{\mathsf{P}}$	anti-unification algorithm for simply-typed λ-terms $\mathcal{T}(\Sigma, \mathcal{V})$
AUE	anti-unification equation
lgg	least general generalization
laa	longest admissible alignment
mcg	minimal complete set of generalizations
dag	directed acyclic graph

Bibliography

[1] Hassan Aït-Kaci and Yutaka Sasaki. An axiomatic approach to feature term generalization. In Luc De Raedt and Peter A. Flach, editors, *ECML*, volume 2167 of *Lecture Notes in Computer Science*, pages 1–12. Springer, 2001. ISBN 3-540-42536-5. doi: 10.1007/3-540-44795-4_1.

[2] María Alpuente, Santiago Escobar, José Meseguer, and Pedro Ojeda. A modular equational generalization algorithm. In Michael Hanus, editor, *LOPSTR*, volume 5438 of *Lecture Notes in Computer Science*, pages 24–39. Springer, 2008. ISBN 978-3-642-00514-5. doi: 10.1007/978-3-642-00515-2_3.

[3] María Alpuente, Santiago Escobar, José Meseguer, and Pedro Ojeda. Order-sorted generalization. *Electr. Notes Theor. Comput. Sci.*, 246:27–38, 2009. doi: 10.1016/j.entcs.2009.07.013.

[4] María Alpuente, Santiago Escobar, Javier Espert, and José Meseguer. A modular order-sorted equational generalization algorithm. *Information and Computation*, 235(0):98 – 136, 2014. ISSN 0890-5401. doi: 10.1016/j.ic.2014.01.006. Special issue on Functional and (Constraint) Logic Programming.

[5] Amihood Amir, Tzvika Hartman, Oren Kapah, B. Riva Shalom, and Dekel Tsur. Generalized LCS. *Theor. Comput. Sci.*, 409(3):438–449, 2008. doi: 10.1016/j.tcs.2008.08.037.

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

[7] Elaheh Asghari and MohammadReza KeyvanPour. Xml document clustering: techniques and challenges. *Artificial Intelligence Review*, 43(3):417–436, 2015. ISSN 0269-2821. doi: 10.1007/s10462-012-9379-2.

[8] Franz Baader. Characterizations of unification type zero. In *Rewriting Techniques and Applications*, pages 2–14. Springer, 1989.

[9] Franz Baader. Unification, weak unification, upper bound, lower bound, and generalization problems. In Ronald V. Book, editor, *RTA*, volume 488 of *Lecture Notes in Computer Science*, pages 86–97. Springer, 1991. ISBN 3-540-53904-2. doi: 10.1007/3-540-53904-2_88.

[10] Franz Baader and Wayne Snyder. Unification theory. *Handbook of automated reasoning*, 1:445–532, 2001.

[11] Henk Barendregt. Introduction to generalized type systems. *J. Funct. Program.*, 1(2):125–154, 1991.

[12] Alexander Baumgartner and Temur Kutsia. Unranked second-order anti-unification. In Ulrich Kohlenbach, Pablo Barceló, and Ruy J. G. B. de Queiroz, editors, *Logic, Language, Information, and Computation - 21st International Workshop, WoLLIC 2014, Valparaíso, Chile, September 1-4, 2014. Proceedings*, volume 8652 of *Lecture Notes in Computer Science*, pages 66–80. Springer, 2014. ISBN 978-3-662-44144-2. doi: 10.1007/978-3-662-44145-9_5.

[13] Alexander Baumgartner and Temur Kutsia. A library of anti-unification algorithms. In Eduardo Fermé and João Leite, editors, *Logics in Artificial Intelligence*, volume 8761 of *Lecture Notes in Computer Science*, pages 543–557. Springer International Publishing, 2014. ISBN 978-3-319-11557-3. doi: 10.1007/978-3-319-11558-0_38.

[14] Alexander Baumgartner, Temur Kutsia, Jordi Levy, and Mateu Villaret. A variant of higher-order anti-unification. In Femke van Raamsdonk, editor, *24th International Conference on Rewriting Techniques and Applications (RTA 2013)*, volume 21 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 113–127, Dagstuhl, Germany, 2013. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik. ISBN 978-3-939897-53-8. doi: 10.4230/LIPIcs.RTA.2013.113.

[15] Alexander Baumgartner, Temur Kutsia, Jordi Levy, and Mateu Villaret. Nominal anti-unification. In Maribel Fernández, editor, *26th International Conference on Rewriting Techniques and Applications, RTA 2015, June 29 to July 1, 2015, Warsaw, Poland*, volume 36 of *LIPIcs*, pages 57–73. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2015. ISBN 978-3-939897-85-9. doi: 10.4230/LIPIcs.RTA.2015.57.

[16] Ira D. Baxter, Andrew Yahin, Leonardo Mendonça de Moura, Marcelo Sant'Anna, and Lorraine Bier. Clone detection using abstract syntax trees. In *1998 International Conference on Software Maintenance, ICSM 1998, Bethesda, Maryland, USA, November 16-19, 1998*, pages 368–377. IEEE Computer Society, 1998. ISBN 0-8186-8779-7. doi: 10.1109/ICSM.1998.738528.

[17] Geert Jan Bex, Sebastian Maneth, and Frank Neven. A formal model for an expressive fragment of xslt. In *Proceedings of the First International Conference on Computational Logic*, CL '00, pages 1137–1151, London, UK, UK, 2000. Springer-Verlag. ISBN 3-540-67797-6.

[18] Armin Biere. Normalisation, unification and generalisation in free monoids. Master's thesis, University of Karlsruhe, 1993. (in German).

[19] Peter Bulychev. Duplicate code detection using Clone Digger. *Python Mag.*, 9: 18–24, 2008.

[20] Peter E. Bulychev, Egor V. Kostylev, and Vladimir A. Zakharov. Anti-unification algorithms and their applications in program analysis. In Amir Pnueli, Irina Virbitskaite, and Andrei Voronkov, editors, *Perspectives of Systems Informatics, 7th International Andrei Ershov Memorial Conference, PSI 2009, Novosibirsk, Russia, June 15-19, 2009. Revised Papers*, volume 5947 of *Lecture Notes in Computer Science*, pages 413–423. Springer, 2009. ISBN 978-3-642-11485-4. doi: 10.1007/978-3-642-11486-1_35.

[21] Petr Bulychev and Marius Minea. An evaluation of duplicate code detection using anti-unification. In *Proc. 3rd International Workshop on Software Clones*, 2009.

[22] J. Burghardt and B. Heinz. *Implementing Anti-unification Modulo Equational Theory*. Arbeitspapiere der GMD. GMD-Forschungszentrum Informationstechnik, 1996.

[23] Jochen Burghardt. E-generalization using grammars. *Artif. Intell.*, 165(1):1–35, 2005. doi: 10.1016/j.artint.2005.01.008.

[24] James Cheney. Relating higher-order pattern unification and nominal unification. In *Proc. 19th International Workshop on Unification, UNIF'05*, pages 104–119,

2005.

[25] James Cheney. Equivariant unification. *J. Autom. Reasoning*, 45(3):267–300, 2010. doi: 10.1007/s10817-009-9164-3.

[26] James Cheney and Christian Urban. alpha-prolog: A logic programming language with names, binding and a-equivalence. In Bart Demoen and Vladimir Lifschitz, editors, *Logic Programming, 20th International Conference, ICLP 2004, Saint-Malo, France, September 6-10, 2004, Proceedings*, volume 3132 of *Lecture Notes in Computer Science*, pages 269–283. Springer, 2004. ISBN 3-540-22671-0. doi: 10.1007/978-3-540-27775-0_19.

[27] Alonzo Church. The calculi of lambda-conversion. *Bull. Amer. Math. Soc. 50 (1944), 169-172*, pages 0002–9904, 1944. doi: 10.1090/S0002-9904-1944-08090-7.

[28] Ilyas Cicekli and Nihan Kesim Cicekli. Generalizing predicates with string arguments. *Appl. Intell.*, 25(1):23–36, 2006. doi: 10.1007/s10489-006-8864-1.

[29] Manuel Clavel, Francisco Durán, Steven Eker, Patrick Lincoln, Narciso Martí-Oliet, José Meseguer, and Carolyn L. Talcott, editors. *All About Maude - A High-Performance Logical Framework, How to Specify, Program and Verify Systems in Rewriting Logic*, volume 4350 of *Lecture Notes in Computer Science*, 2007. Springer. ISBN 978-3-540-71940-3.

[30] Arthur L. Delcher and Simon Kasif. Efficient parallel term matching and anti-unification. *J. Autom. Reasoning*, 9(3):391–406, 1992.

[31] Gilles Dowek. Higher-order unification and matching. In John Alan Robinson and Andrei Voronkov, editors, *Handbook of Automated Reasoning*, pages 1009–1062. Elsevier and MIT Press, 2001. ISBN 0-444-50813-9, 0-262-18223-8.

[32] Gilles Dowek, Murdoch James Gabbay, and Dominic P. Mulligan. Permissive nominal terms and their unification: an infinite, co-infinite approach to nominal techniques. *Logic Journal of the IGPL*, 18(6):769–822, 2010. doi: 10.1093/jigpal/jzq006.

[33] William S. Evans, Christopher W. Fraser, and Fei Ma. Clone detection via structural abstraction. *Software Quality Journal*, 17(4):309–330, 2009. doi: 10.1007/s11219-009-9074-y.

[34] Cao Feng and Stephen Muggleton. Towards inductive generalization in higher order logic. In Derek H. Sleeman and Peter Edwards, editors, *ML*, pages 154–162. Morgan Kaufmann, 1992. ISBN 1-55860-247-X.

[35] A. Fornells, E. Armengol, E. Golobardes, S. Puig, and J. Malvehy. Experiences using clustering and generalizations for knowledge discovery in melanomas domain. In Petra Perner, editor, *Advances in Data Mining. Medical Applications, E-Commerce, Marketing, and Theoretical Aspects*, volume 5077 of *Lecture Notes in Computer Science*, pages 57–71. Springer Berlin Heidelberg, 2008. ISBN 978-3-540-70717-2. doi: 10.1007/978-3-540-70720-2_5.

[36] Koichi Furukawa, Mutumi Imai, and Randy Goebel. Hyper least general generalization and its application to higher-order concept learning. *manuscript draft*, 1996.

[37] Murdoch Gabbay and Andrew M. Pitts. A new approach to abstract syntax with variable binding. *Formal Asp. Comput.*, 13(3-5):341–363, 2002. doi: 10.1007/s001650200016.

[38] Murdoch J. Gabbay. *A Theory of Inductive Definitions with alpha-Equivalence*. PhD thesis, University of Cambridge, UK, 2000.

[39] Murdoch J. Gabbay and Andrew M. Pitts. A new approach to abstract syntax involving binders. In *LICS*, pages 214–224. IEEE Computer Society, 1999. ISBN 0-7695-0158-3.

[40] Boris A. Galitsky, Josep Lluis De La Rosa, and GáBor Dobrocsi. Mapping syntactic to semantic generalizations of linguistic parse trees. In R. Charles Murray and Philip M. McCarthy, editors, *Proceedings of the Twenty-Fourth International Florida Artificial Intelligence Research Society Conference, May 18-20, 2011, Palm Beach, Florida, USA*. AAAI Press, 2011.

[41] Jeremy J Gray and Karen Hunger Parshall. *Episodes in the history of modern algebra (1800-1950)*, volume 32. American Mathematical Soc., 2011.

[42] Masami Hagiya. Generalization from partial parametrization in higher-order type theory. *Theor. Comput. Sci.*, 63(2):113–139, 1989. doi: 10.1016/0304-3975(89)90074-1.

[43] Robert W. Hasker. *The Replay of Program Derivations*. PhD thesis, University of Illionois at Urbana-Champaign, 1995.

[44] Birgit Heinz. *Anti-Unifikation modulo Gleichungstheorie und deren Anwendung zur Lemmagenerierung*. PhD thesis, Muenchen, 1996. Diss.Berlin, 1995.

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

[46] Ming-Yang Kao, Tak Wah Lam, Wing-Kin Sung, and Hing-Fung Ting. An even faster and more unifying algorithm for comparing trees via unbalanced bipartite matchings. *J. Algorithms*, 40(2):212–233, 2001. doi: 10.1006/jagm.2001.1163.

[47] Emanuel Kitzelmann and Ute Schmid. Inductive synthesis of functional programs: An explanation based generalization approach. *Journal of Machine Learning Research*, 7:429–454, 2006.

[48] Rainer Koschke, Raimar Falke, and Pierre Frenzel. Clone detection using abstract syntax suffix trees. In *13th Working Conference on Reverse Engineering (WCRE 2006), 23-27 October 2006, Benevento, Italy*, pages 253–262. IEEE Computer Society, 2006. ISBN 0-7695-2719-1. doi: 10.1109/WCRE.2006.18.

[49] Ulf Krumnack, Angela Schwering, Helmar Gust, and Kai-Uwe Kühnberger. Restricted higher-order anti-unification for analogy making. In Mehmet A. Orgun and John Thornton, editors, *Australian Conference on Artificial Intelligence*, volume 4830 of *Lecture Notes in Computer Science*, pages 273–282. Springer, 2007. ISBN 978-3-540-76926-2. doi: 10.1007/978-3-540-76928-6_29.

[50] S. Kuo and G. R. Cross. An improved algorithm to find the length of the longest common subsequence of two strings. *SIGIR Forum*, 23(3-4):89–99, April 1989. ISSN 0163-5840. doi: 10.1145/74697.74702.

[51] Temur Kutsia. Solving equations with sequence variables and sequence functions. *J. Symb. Comput.*, 42(3):352–388, 2007. doi: 10.1016/j.jsc.2006.12.002.

[52] Temur Kutsia and Mircea Marin. Regular expression order-sorted unification and matching. *Journal of Symbolic Computation*, 67(0):42 – 67, 2015. ISSN 0747-7171. doi: 10.1016/j.jsc.2014.08.002.

[53] Temur Kutsia, Jordi Levy, and Mateu Villaret. On the relation between context and sequence unification. *J. Symb. Comput.*, 45(1):74–95, January 2010. ISSN

0747-7171. doi: 10.1016/j.jsc.2009.07.001.

[54] Temur Kutsia, Jordi Levy, and Mateu Villaret. Anti-unification for unranked terms and hedges. *J. Autom. Reasoning*, 52(2):155–190, 2014. doi: 10.1007/s10817-013-9285-6.

[55] Jordi Levy and Mateu Villaret. Nominal unification from a higher-order perspective. *ACM Trans. Comput. Log.*, 13(2):10, 2012. doi: 10.1145/2159531.2159532.

[56] Huiqing Li and Simon J. Thompson. Similar code detection and elimination for erlang programs. In Manuel Carro and Ricardo Peña, editors, *Practical Aspects of Declarative Languages, 12th International Symposium, PADL 2010, Madrid, Spain, January 18-19, 2010. Proceedings*, volume 5937 of *Lecture Notes in Computer Science*, pages 104–118. Springer, 2010. ISBN 978-3-642-11502-8. doi: 10.1007/978-3-642-11503-5_10.

[57] Jianguo Lu, John Mylopoulos, Masateru Harao, and Masami Hagiya. Higher order generalization and its application in program verification. *Ann. Math. Artif. Intell.*, 28(1-4):107–126, 2000.

[58] Gavin Mendel-Gleason. *Types and Verification for Infinite State Systems.* PhD thesis, Dublin City University, 2012.

[59] Dale Miller. A logic programming language with lambda-abstraction, function variables, and simple unification. *J. Log. Comput.*, 1(4):497–536, 1991. doi: 10.1093/logcom/1.4.497.

[60] Dale Miller and Gopalan Nadathur. A logic programming approach to manipulating formulas and programs. In *Proceedings of the 1987 Symposium on Logic Programming, San Francisco, California, USA, August 31 - September 4, 1987*, pages 379–388. IEEE-CS, 1987. ISBN 0-8186-0799-8.

[61] Makoto Murata. Transformation of documents and schemas by patterns and contextual conditions. In *Proceedings of the Third International Workshop on Principles of Document Processing*, PODP '96, pages 153–169, London, UK, UK, 1997. Springer-Verlag. ISBN 3-540-63620-X.

[62] Makoto Murata. Data model for document transformation and assembly. In *Proceedings of the 4th International Workshop on Principles of Digital Document Processing*, PODDP '98, pages 140–152, London, UK, UK, 1998. Springer-Verlag. ISBN 3-540-65086-5.

[63] Gopalan Nadathur and Dale Miller. An overview of lambda-prolog. In Robert A. Kowalski and Kenneth A. Bowen, editors, *Logic Programming, Proceedings of the Fifth International Conference and Symposium, Seattle, Washington, August 15-19, 1988 (2 Volumes)*, pages 810–827. MIT Press, 1988. ISBN 0-262-61056-6.

[64] Maxwell Herman Alexander Newman. On theories with a combinatorial definition of "equivalence". *Annals of mathematics*, pages 223–243, 1942.

[65] Tobias Nipkow. Functional unification of higher-order patterns. In *LICS*, pages 64–74. IEEE Computer Society, 1993. ISBN 0-8186-3140-6. doi: 10.1109/LICS.1993.287599.

[66] Srinivas Padmanabhuni, Randy Goebel, and Koichi Furukawa. Curried least general generalization: A framework for higher order concept learning. In *PRICAI Workshops*, pages 45–60, 1996. doi: 10.1007/3-540-64413-X_27.

[67] Zeshan Peng and Hingfung Ting. An $O(n \log n)$-time algorithm for the maximum constrained agreement subtree problem for binary trees. In Rudolf Fleischer and

Gerhard Trippen, editors, *Algorithms and Computation*, volume 3341 of *Lecture Notes in Computer Science*, pages 754–765. Springer Berlin Heidelberg, 2005. ISBN 978-3-540-24131-7. doi: 10.1007/978-3-540-30551-4_65.

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

[69] Brigitte Pientka. Higher-order term indexing using substitution trees. *ACM Trans. Comput. Log.*, 11(1), 2009. doi: 10.1145/1614431.1614437.

[70] Enric Plaza. Cases as terms: A feature term approach to the structured representation of cases. In Manuela M. Veloso and Agnar Aamodt, editors, *ICCBR*, volume 1010 of *Lecture Notes in Computer Science*, pages 265–276. Springer, 1995. ISBN 3-540-60598-3. doi: 10.1007/3-540-60598-3_24.

[71] Gordon D. Plotkin. A note on inductive generalization. *Machine Intel.*, 5(1): 153–163, 1970.

[72] RJ Popplestone. An experiment in automatic induction. *Machine Intelligence*, 5: 203–215, 1970.

[73] Loic Pottier. Generalisation de termes en theorie equationelle: Cas associatif-commutatif., 1989.

[74] John C. Reynolds. Transformational systems and the algebraic structure of atomic formulas. *Machine Intel.*, 5(1):135–151, 1970.

[75] Chanchal K. Roy, James R. Cordy, and Rainer Koschke. Comparison and evaluation of code clone detection techniques and tools: A qualitative approach. *Science of Computer Programming*, 74(7):470 – 495, 2009. ISSN 0167-6423. doi: 10.1016/j.scico.2009.02.007. Special Issue on Program Comprehension (ICPC 2008).

[76] Chanchal Kumar Roy and James R. Cordy. A survey of software clone detection research. Technical report, School of Computing, Queen's University at Kingston, Ontario, Canada, 2007.

[77] Ute Schmid. *Inductive Synthesis of Functional Programs, Universal Planning, Folding of Finite Programs, and Schema Abstraction by Analogical Reasoning*, volume 2654 of *Lecture Notes in Computer Science*. Springer, 2003. ISBN 3-540-40174-1.

[78] Martin Schmidt, Helmar Gust, Kai-Uwe Kühnberger, and Ulf Krumnack. Refinements of restricted higher-order anti-unification for heuristic-driven theory projection. In Joscha Bach and Stefan Edelkamp, editors, *KI*, volume 7006 of *Lecture Notes in Computer Science*, pages 289–300. Springer, 2011. ISBN 978-3-642-24454-4. doi: 10.1007/978-3-642-24455-1_28.

[79] Mark R. Shinwell, Andrew M. Pitts, and Murdoch James Gabbay. Freshml: programming with binders made simple. *SIGPLAN Notices*, 38(9):263–274, 2003. doi: 10.1145/944746.944729.

[80] Bernd Thomas. Learning t-wrappers for information extraction. In *Workshop on Machine Learning in Human Language Technology*, 1999.

[81] Christian Urban. Nominal techniques in isabelle/hol. *J. Autom. Reasoning*, 40(4): 327–356, 2008. doi: 10.1007/s10817-008-9097-2.

[82] Christian Urban, Andrew M. Pitts, and Murdoch Gabbay. Nominal unificaiton. In Matthias Baaz and Johann A. Makowsky, editors, *CSL*, volume 2803 of *Lecture Notes in Computer Science*, pages 513–527. Springer, 2003. ISBN 3-540-40801-0.

doi: 10.1007/978-3-540-45220-1_41.

[83] Christian Urban, Andrew M. Pitts, and Murdoch Gabbay. Nominal unification. *Theor. Comput. Sci.*, 323(1-3):473–497, 2004. doi: 10.1016/j.tcs.2004.06.016. URL `http://dx.doi.org/10.1016/j.tcs.2004.06.016`.

[84] Vera Wahler, Dietmar Seipel, Jürgen Wolff von Gudenberg, and Gregor Fischer. Clone detection in source code by frequent itemset techniques. In *SCAM*, pages 128–135. IEEE Computer Society, 2004. ISBN 0-7695-2144-4.

[85] Akihiro Yamamoto, Kimihito Ito, Akira Ishino, and Hiroki Arimura. Modelling semi-structured documents with hedges for deduction and induction. In Céline Rouveirol and Michèle Sebag, editors, *ILP*, volume 2157 of *Lecture Notes in Computer Science*, pages 240–247. Springer, 2001. ISBN 3-540-42538-1.

[86] Wuu Yang. Identifying syntactic differences between two programs. *Softw., Pract. Exper.*, 21(7):739–755, 1991. doi: 10.1002/spe.4380210706.

[87] Kaizhong Zhang. Algorithms for the constrained editing problem between ordered labeled trees and related problems. *Pattern Recognition*, 28:463–474, 1995.

Curriculum Vitae

Alexander Baumgartner

http://www.risc.jku.at/home/abaumgar/

Personal Data

Gender: Male
Date of birth: 14th of September, 1976 (Salzburg, Austria)
Citizenship: Austria

Affiliation

Research Institute for Symbolic Computation (RISC)
Johannes Kepler University Linz
Altenbergerstraße 69
A-4040 Linz, Austria

Office: Schloß Hagenberg, -2.12
Tel: +43 (0)732 2468 9967
e-Mail: abaumgar@risc.jku.at

Education

2012–present PhD Studies in Technical Sciences (Doktoratsstudium der technischen Wissenschaften) at the Research Institute for Symbolic Computation, Johannes Kepler University, A-4040 Linz, Austria.
Thesis Advisor: Priv.-Doz. Dr. Temur Kutsia

2006–2011 Diploma Studies in Computer Science at the Paris Lodron University of Salzburg, A-5020 Salzburg, Austria.
Thesis Advisor: Ao.Univ.-Prof. Dr. Helge Hagenauer

June 1997 Matura (higher education entrance qualification) at the HBLA Ursprung, A-5161 Elixhausen, Austria.

Working Experience

2012–present Research Associate at the Research Institute for Symbolic Computation, Johannes Kepler University, A-4040 Linz, Austria.

2001–2012 Software Engineer at Land Salzburg (Government of the Austrian state Salzburg), A-5020 Salzburg, Austria.

1998–2001 Programmer at Wüstenrot AG (Financial institution), A-5020 Salzburg, Austria.

Teaching Experience

Summer 2015	Practical Software Technology; Lecture 4 hours per week; Course language: English.
Summer 2015	Formal Foundations in Business Informatics; Exercise class 1 hour per week; Course language: German.
Winter 2014	Mathematical Analysis; Exercise class 2 hours per week; Course language: German.
Winter 2013	Mathematical Analysis; Exercise class 2 hours per week; Course language: German.

Publications

[1] Alexander Baumgartner, Temur Kutsia, Jordi Levy, and Mateu Villaret. Nominal anti-unification. In Maribel Fernández, editor, *26th International Conference on Rewriting Techniques and Applications, RTA 2015, June 29 to July 1, 2015, Warsaw, Poland*, volume 36 of *LIPIcs*, pages 57–73. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2015.

[2] Alexander Baumgartner and Temur Kutsia. A library of anti-unification algorithms. In Eduardo Fermé and João Leite, editors, *Logics in Artificial Intelligence*, volume 8761 of *Lecture Notes in Computer Science*, pages 543–557. Springer International Publishing, 2014.

[3] Alexander Baumgartner and Temur Kutsia. Unranked second-order anti-unification. In Ulrich Kohlenbach, Pablo Barceló, and Ruy J. G. B. de Queiroz, editors, *Logic, Language, Information, and Computation - 21st International Workshop, WoLLIC 2014, Valparaíso, Chile, September 1-4, 2014. Proceedings*, volume 8652 of *Lecture Notes in Computer Science*, pages 66–80. Springer, 2014.

[4] Alexander Baumgartner, Temur Kutsia, Jordi Levy, and Mateu Villaret. Nominal anti-unification. In Temur Kutsia and Christophe Ringeissen, editors, *Proceedings of the 28th International Workshop on Unification, UNIF 2014*, pages 62–68, 2014.

[5] Alexander Baumgartner and Temur Kutsia. Unranked anti-unification with hedge and context variables. In Konstantin Korovin and Barbara Morawska, editors, *27th International Workshop on Unification, UNIF 2013, Eindhoven, Netherlands, June 26, 2013*, volume 19 of *EPiC Series*, pages 13–21. EasyChair, 2013.

[6] Alexander Baumgartner, Temur Kutsia, Jordi Levy, and Mateu Villaret. A variant of higher-order anti-unification. In Femke van Raamsdonk, editor, *24th International Conference on Rewriting Techniques and Applications (RTA 2013)*, volume 21 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 113–127, Dagstuhl, Germany, 2013. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik.