A Variant of Higher-Order Anti-Unification*

Alexander Baumgartner[1], Temur Kutsia[1], Jordi Levy[2], and Mateu Villaret[3]

1 **Research Institute for Symbolic Computation (RISC)**
Johannes Kepler University, Linz, Austria
{abaumgar,kutsia}@risc.jku.at
2 **Artificial Intelligence Research Institute (IIIA)**
Spanish Council for Scientific Research (CSIC), Barcelona, Spain
levy@iiia.csic.es
3 **Departament d'Informàtica i Matemàtica Aplicada (IMA)**
Universitat de Girona (UdG), Girona, Spain
villaret@ima.udg.edu

Abstract

We present a rule-based Huet's style anti-unification algorithm for simply-typed lambda-terms in η-long β-normal form, which computes a least general higher-order pattern generalization. For a pair of arbitrary terms of the same type, such a generalization always exists and is unique modulo α-equivalence and variable renaming. The algorithm computes it in cubic time within linear space. It has been implemented and the code is freely available.

1998 ACM Subject Classification F.2.2 Nonnumerical Algorithms and Problems, F.4.1 Mathematical Logic

Keywords and phrases higher-order anti-unification, higher-order patterns.

Digital Object Identifier 10.4230/LIPIcs.RTA.2013.113

1 Introduction

The anti-unification problem of two terms t_1 and t_2 is concerned with finding their generalization, a term t such that both t_1 and t_2 are instances of t under some substitutions. Interesting generalizations are the least general ones. The purpose of anti-unification algorithms is to compute such least general generalizations (lggs).

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. One of such classes is formed by higher-order patterns. These are λ-terms where the arguments of free variables are distinct bound variables. They have been introduced by Miller [25] and gained popularity because of an attractive combination of expressive power and computational costs: There are practical unification algorithms [28, 27, 26] that compute most general unifiers whenever they exist. Pfenning gave the first algorithm for higher-order pattern anti-unification in the Calculus of Constructions [28], with the intention of using it for proof generalization.

Since then, there have been several approaches to higher-order anti-unification, designing algorithms in various restricted cases. Motivated by applications in inductive learning, Feng and Muggleton [14] proposed anti-unification in $M\lambda$, which is essentially an extension of

* This research has been partially supported by the projects HeLo (TIN2012-33042) and TASSAT (TIN2010-20967-C04-01), by the Austrian Science Fund (FWF) with the project SToUT (P 24087-N18) and by the Generalitat de Catalunya with the grant AGAUR 2009-SGR-1434.

higher-order patterns by permitting free variables to apply to object terms, not only to bound variables. Object terms may contain constants, free variables, and variables which are bound outside of object terms. The algorithm has been implemented and was used for inductive generalization.

Anti-unification in a restricted version of $\lambda 2$ (a second-order λ-calculus with type variables [4]) has been studied in [23] with applications in analogical programming and analogical theorem proving. The imposed restrictions guarantee uniqueness of the least general generalization. This algorithm as well as the one for higher-order patterns by Pfenning [28] have influenced the generalization algorithm used in the program transformation technique called supercompilation [24].

There are other fragments of higher-order anti-unification, motivated by analogical reasoning. A restricted version of second-order generalization developed in [15] has an application in the replay of program derivations. A symbolic analogy model, called Heuristic-Driven Theory Projection, uses yet another restriction of higher-order anti-unification to detect analogies between different domains [18].

The last decade has seen a revived interest in anti-unification. The problem has been studied in various theories (e.g., [1, 2, 9, 19]) and from different application points of view (e.g., [3, 8, 18, 23, 31, 22]). A particularly interesting application comes from software code refactoring, to find similar pieces of code, e.g., in Python, Java [6, 7] and Erlang [22] programs. These approaches are based on the first-order anti-unification [29, 30]. To advance the refactoring and clone detection techniques for languages based on λProlog, one needs to employ anti-unification for higher-order terms. This potential application can serve as a motivation to look into the problem of higher-order anti-unification in more detail.

In this paper, we revisit the problem of higher-order anti-unification, permit arbitrary terms as the input and require higher-order patterns in the output, and present an algorithm in the simply-typed setting. The main contributions can be briefly summarized as follows:

1. Designing a rule-based anti-unification algorithm in simply-typed λ-calculus (in Sect. 3). The input of the algorithm are arbitrary terms in η-long β-normal form. The output is a higher-order pattern. The formulation follows Huet's simple and elegant style [17]. The global function for recording disagreements is represented as a store, in the spirit of [1, 2].
2. Proofs of the termination, soundness, and completeness properties of the anti-unification algorithm (in Sect. 4) and its subalgorithm, which computes permuting matchers between patterns (in Sect. 3.2).
3. Complexity analysis (in Sect. 4): The algorithm computes a least general pattern generalization, which always exists and is *unique* modulo α-equivalence, in *cubic* time and requires linear space. As it is done in related work, we assume that symbols and pointers are encoded in constant space, and basic operations on them performed in constant time.
4. Free open-source implementation for both simply-typed and untyped calculi (Sect. 5).

An extended version of this paper appears as the technical report [5].

Related Work

Here we briefly compare our work with the existing results in higher-order anti-unification. The approaches which are closest to us are the following two:

- In [28], Pfenning studied anti-unification in the Calculus of Construction, whose type system is richer than the simple types we consider. Both the input and the output was

required to be higher-order patterns. Some questions have remained open, including the efficiency, applicability, and implementations of the algorithm. Due to the nature of type dependencies in the calculus, the author was not able to formulate the algorithm in Huet's style [17], where a global function is used to guarantee that the same disagreements between the input terms are mapped to the same variable. The complexity has not been studied and the proofs of the algorithm properties have been just sketched.

- Anti-unification in $M\lambda$ [14] is performed on simply-typed terms, where both the input and the output are restricted to a certain extension of higher-order patterns. In this sense it is not comparable to our case, because we do not restrict the input, but require patterns in the output. The paper [14] contains neither the complexity analysis of the $M\lambda$ anti-unification algorithm nor the proofs of its properties.

Some more remotely related / incomparable to us results are listed below:

- Anti-unification studied in [23] is defined in a restricted version of $\lambda 2$. The restriction requires the λ-abstraction not to be used in arguments. The algorithm computes a generalization which is least general with respect to the combination of several orderings defined in the paper. The properties of the algorithm are formally proved, but the complexity has not been analyzed. As the authors point out, the orderings they define are not comparable with the ordering used to compute higher-order pattern generalizations.
- Generalization algorithms in [16] work on second-order terms which contain no λ-abstractions. The output is also restricted: It may contain variables which can be instantiated with multi-hole contexts only. Varying restrictions on the instantiation, various versions of generalizations are obtained. This approach is not comparable with ours.
- The anti-unification algorithm in [18] works on λ-abstraction-free terms as well. It has been developed for analogy making. The application dictates the typical input to be first-order, while their generalizations may contain second-order variables. A certain measure is introduced to compare generalizations, and the algorithm computes those which are preferred by this measure. This approach is not comparable with ours either.
- The approach in [15] is also different from what we do. The anti-unification algorithm there works on a restriction of combinator terms and computes their generalizations (in quadratic time). It has been used for program derivation.

2 Preliminaries

In higher-order signatures we have *types* constructed from a set of *basic types* (typically δ) using the grammar $\tau ::= \delta \mid \tau \to \tau$, where $\to$ is associative to the right. *Variables* (typically $X, Y, Z, x, y, z, a, b, \ldots$) and *constants* (typically $f, c, \ldots$) have an assigned type.

λ-*terms* (typically $t, s, u, \ldots$) 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, and are typed as usual. Terms of the form $(\ldots(h\ t_1)\ldots t_m)$, where h is a constant or a variable, will be written as $h(t_1, \ldots, t_m)$, and terms of the form $\lambda x_1.\cdots.\lambda x_n.t$ as $\lambda x_1, \ldots, x_n.t$. We use $\vec{x}$ as a short-hand for $x_1, \ldots, x_n$.

Other standard notions of the simply typed λ-calculus, like bound and free occurrences of variables, α-conversion, β-reduction, η-long β-normal form, etc. are defined as usual (see [12]). By default, terms are assumed to be written in η-long β-normal form. Therefore, all terms have the form $\lambda x_1, \ldots, x_n.h(t_1, \ldots, t_m)$, where $n, m \geqslant 0$, h is either a constant or a variable, $t_1, \ldots, t_m$ have also this form, and the term $h(t_1, \ldots, t_m)$ has a basic type.

The set of free variables of a term t is denoted by $\mathrm{Vars}(t)$. When we write an equality between two λ-terms, we mean that they are equivalent modulo α, β and η equivalence.

The *depth* of a term t, denoted $\mathrm{Depth}(t)$ is defined recursively as follows: $\mathrm{Depth}(x) = \mathrm{Depth}(c) = 1$, $\mathrm{Depth}(h(t_1, \ldots, t_n)) = 1 + \max_i \mathrm{Depth}(t_i)$, and $\mathrm{Depth}(\lambda x.t) = 1 + \mathrm{Depth}(t)$.

For a term $t = \lambda x_1, \ldots, x_n.h(t_1, \ldots, t_m)$ with $n, m \geqslant 0$, its *head* is defined as $\mathrm{Head}(t) = h$.

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 empty sequence), the occurrence of $\lambda x.$ stands in the position 1, the bound occurrence of y in 1.1.1.1.1.2, the bound occurrence of u in 2.1.1, etc.

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 1.1.1.1.1 in $f(\lambda x.\lambda y.g(\lambda z.h(z, y), x), \lambda u.g(u))$ is $f, \lambda x, \lambda y, g, \lambda z$.

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. 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.

Substitutions are finite sets of pairs $\{X_1 \mapsto t_1, \ldots, X_n \mapsto t_n\}$ where X_i and t_i have the same type and the X's are pairwise distinct variables. They can be extended to type preserving functions from terms to terms as usual, avoiding variable capture. The notions of substitution *domain* and *range* are also standard and are denoted, respectively, by Dom and Ran.

We use postfix notation for substitution applications, writing $t\sigma$ instead of $\sigma(t)$. As usual, the application $t\sigma$ affects only the free occurrences of variables from $\mathrm{Dom}(\sigma)$ in t. 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 $S\sigma = \{t\sigma \mid t \in S\}$. The *composition* of σ and ϑ is written as juxtaposition $\sigma\vartheta$. Yet another standard operation, *restriction* of a substitution σ to a set of variables S, is denoted by $\sigma|_S$.

A substitution σ_1 is *more general* than σ_2, written $\sigma_1 \preceq \sigma_2$, if there exists ϑ such that $X\sigma_1\vartheta = X\sigma_2$ for all $X \in \mathrm{Dom}(\sigma_1) \cup \mathrm{Dom}(\sigma_2)$. The strict part of this relation is denoted by $\prec$. The relation $\preceq$ is a partial order and generates the equivalence relation which we denote by $\simeq$. We overload $\preceq$ by defining $s \preceq t$ if there exists a substitution σ such that $s\sigma = t$.

A term t is called a *generalization* or an *anti-instance* 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* if additionally t is a higher-order pattern. It is the *least general generalization,* (lgg in short), aka a *most specific anti-instance,* of t_1 and t_2, if there is no generalization s of t_1 and t_2 which satisfies $t \prec s$.

An *anti-unification problem* (shortly AUP) is a triple $X(\vec{x}) : t \triangleq s$ where

- $\lambda\vec{x}.X(\vec{x})$, $\lambda\vec{x}.t$, and $\lambda\vec{x}.s$ are terms of the same type,
- t and s are in η-long β-normal form, and
- X does not occur in t and s.

The variable X is called a *generalization variable.* The term $X(\vec{x})$ is called the *generalization term.* The variables that belong to $\vec{x}$, as well as bound variables, are written in the lower case letters $x, y, z, \ldots$. Originally free variables, including the generalization variables, are written with the capital letters $X, Y, Z, \ldots$. This notation intuitively corresponds to the usual convention about syntactically distinguishing bound and free variables.

An *anti-unifier* of an AUP $X(\vec{x}) : t \triangleq s$ is a substitution σ such that $\mathrm{Dom}(\sigma) = \{X\}$ and $\lambda\vec{x}.X(\vec{x})\sigma$ is a term which generalizes both $\lambda\vec{x}.t$ and $\lambda\vec{x}.s$.

An anti-unifier of $X(\vec{x}) : t \triangleq s$ is *least general* (or *most specific*) if there is no anti-unifier ϑ of the same problem that satisfies $\sigma \prec \vartheta$. Obviously, if σ is a least general anti-unifier of an AUP $X(\vec{x}) : t \triangleq s$, then $\lambda\vec{x}.X(\vec{x})\sigma$ is an lgg of $\lambda\vec{x}.t$ and $\lambda\vec{x}.s$.

Here we consider a variant of higher-order anti-unification problem:

Given: Higher-order terms t and s of the same type in η-long β-normal form.
Find: A higher-order pattern generalization r of t and s.

The problem statement means that we are looking for r which is least general among all higher-order patterns which generalize t and s. There can still exist a term which is less general than r, generalizes both s and t, 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, which is an 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.

Below we assume that in the AUPs of the form $X(\vec{x}) : t \triangleq s$, the term $\lambda\vec{x}.X(\vec{x})$ is a higher-order pattern.

3 The Algorithm the Higher-Order Anti-Unification Variant

3.1 The Rules

The higher-order anti-unification algorithm is formulated in a rule-based manner working on triples $A; S; \sigma$ (*systems*). Here A is a set of AUPs of the form $\{X_1(\vec{x_1}) : t_1 \triangleq s_1, \ldots, X_n(\vec{x_n}) : t_n \triangleq s_n\}$ where each X_i occurs in $A \cup S$ only once, S is a set of already solved AUPs (the store), and σ is a substitution (computed so far) mapping variables to patterns.

▸ Remark. One assumption we make on the set $A \cup S$ is that each occurrence of λ binds a distinct name variable (in other words, all names of bound variables are distinct).

Dec: **Decomposition**

$$\{X(\vec{x}) : h(t_1, \ldots, t_m) \triangleq h(s_1, \ldots, s_m)\} \cup A;\ S;\ \sigma \Longrightarrow \{Y_1(\vec{x}) : t_1 \triangleq s_1, \ldots, Y_m(\vec{x}) : t_m \triangleq s_m\} \cup A;\ S;\ \sigma\{X \mapsto \lambda\vec{x}.h(Y_1(\vec{x}), \ldots, Y_m(\vec{x}))\},$$

where h is a constant or $h \in \vec{x}$, and $Y_1, \ldots, Y_n$ are fresh variables of the corresponding types.

Abs: **Abstraction**

$$\{X(\vec{x}) : \lambda y.t \triangleq \lambda z.s\} \cup A;\ S;\ \sigma \Longrightarrow \{X'(\vec{x}, y) : t \triangleq s\{z \mapsto y\}\} \cup A;\ S;\ \sigma\{X \mapsto \lambda\vec{x}, y.X'(\vec{x}, y)\}.$$

where X' is a fresh variable of the appropriate type.

Sol: **Solve**

$$\{X(\vec{x}) : t \triangleq s\} \cup A;\ S;\ \sigma \Longrightarrow A;\ \{Y(\vec{y}) : t \triangleq s\} \cup S;\ \sigma\{X \mapsto \lambda\vec{x}.Y(\vec{y})\},$$

where t and s are of a basic type, $\mathrm{Head}(t) \neq \mathrm{Head}(s)$ or $\mathrm{Head}(t) = \mathrm{Head}(s) = Z \notin \vec{x}$, $\vec{y}$ is a subsequence of $\vec{x}$ consisting of the variables that appear freely in t or in s, and Y is a fresh variable of the corresponding type.

Mer: **Merge**

$$A;\ \{X(\vec{x}) : t_1 \triangleq t_2, Y(\vec{y}) : s_1 \triangleq s_2\} \cup S;\ \sigma \Longrightarrow A;\ \{X(\vec{x}) : t_1 \triangleq t_2\} \cup S;\ \sigma\{Y \mapsto \lambda\vec{y}.X(\vec{x}\pi)\},$$

where $\pi : \{\vec{x}\} \to \{\vec{y}\}$ is a bijection, extended as a substitution, with $t_1\pi = s_1$ and $t_2\pi = s_2$.

One can easily show that a triple obtained from $A; S; \sigma$ by applying any of the rules above to a system is indeed a system: For each expression $X(\vec{x}) : t \triangleq s \in A \cup S$, 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 $A \cup S$ 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 $A \cup S$ except $X'(\vec{x}, y)$ and, maybe, t and s.

Like in the anti-unification algorithms working on triple systems [1, 2, 19], the idea of the store here is to keep track of already solved AUPs in order to reuse in generalizations an existing variable. This is important, since we aim at computing lggs.

The Mer rule requires solving a matching problem $\{t_1 \Rightarrow s_1, t_2 \Rightarrow s_2\}$ with the substitution 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 P, 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 P from D to R. The sets D and R are supposed to have the same cardinality.

Note that a permuted 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 P 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. By $\mathsf{match}(D, R, P)$, we denote such a permuting matcher of P from D to R, when it exists. Otherwise, $\mathsf{match}(D, R, P) = \bot$. An algorithm that computes it is given in Sect. 3.2 below.

To compute generalizations for terms t and s, we start with $\{X : t \triangleq s\}; \varnothing; \varnothing$, where X is a fresh variable, and apply the rules as long as possible. We denote this procedure by $\mathfrak{P}$, to indicate that we compute patterns. The system to which no rule applies has the form $\varnothing; S; \varphi$, where Mer does not apply to S. We call it the final system. When $\mathfrak{P}$ transforms $\{X : t \triangleq s\}; \varnothing; \varnothing$ into a final system $\varnothing; S; \varphi$, we say that *result computed* by $\mathfrak{P}$ is $X\varphi$.

▸ **Example 3.1.** A couple of examples illustrating the generalizations computed by $\mathfrak{P}$:

- 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 $\mathfrak{P}$ performs the following transformations:

$$
\begin{aligned}
& \{X : \lambda x, y.f(U(g(x), y), U(g(y), x)) \triangleq \lambda x', y'.f(h(y', g(x')), h(x', g(y')))\}; \varnothing; \varnothing \\
\Longrightarrow^{2}_{\mathsf{Abs}}\ & \{X'(x, y) : f(U(g(x), y), U(g(y), x)) \triangleq f(h(y, g(x)), h(x, g(y)))\}; \varnothing; \\
& \{X \mapsto \lambda x, y.X'(x, y)\} \\
\Longrightarrow_{\mathsf{Dec}}\ & \{Y_1(x, y) : U(g(x), y) \triangleq h(y, g(x)), Y_2(x, y) : U(g(y), x) \triangleq h(x, g(y))\}; \varnothing; \\
& \{X \mapsto \lambda x, y.f(Y_1(x, y), Y_2(x, y)), \dots\} \\
\Longrightarrow_{\mathsf{Sol}}\ & \{Y_2(x, y) : U(g(y), x) \triangleq h(x, g(y))\}; \{Y_1(x, y) : U(g(x), y) \triangleq h(y, g(x))\}; \\
& \{X \mapsto \lambda x, y.f(Y_1(x, y), Y_2(x, y)), \dots\} \\
\Longrightarrow_{\mathsf{Sol}}\ & \varnothing; \{Y_1(x, y) : U(g(x), y) \triangleq h(y, g(x)), Y_2(x, y) : U(g(y), x) \triangleq h(x, g(y))\}; \\
& \{X \mapsto \lambda x, y.f(Y_1(x, y), Y_2(x, y)), \dots\} \\
\Longrightarrow_{\mathsf{Mer}}\ & \varnothing; \{Y_1(x, y) : U(g(x), y) \triangleq h(y, g(x))\} \\
& \{X \mapsto \lambda x, y.f(Y_1(x, y), Y_1(y, x)), \dots, Y_2 \mapsto \lambda x, y.Y_1(y, x)\}
\end{aligned}
$$

  The computed result is $r = \lambda x, y.f(Y_1(x, y), Y_1(y, x))$. It generalizes the input terms t and s: $r\{Y_1 \mapsto \lambda x, y.U(g(x), y)\} = t$ and $r\{Y_1 \mapsto \lambda x, y.h(y, g(x))\} = s$. These substitutions can be read from the final store.
- For $\lambda x, y, z.g(f(x, z), f(y, z), f(y, x))$ and $\lambda x', y', z'.g(h(y', x'), h(x', y'), h(z', y'))$, $\mathfrak{P}$ computes their generalization $\lambda x, y, z.f(Y_1(x, y, z), Y_1(y, x, z), Y_1(y, z, x))$

- For $\lambda x, y.f(\lambda z.U(z, y, x), U(x, y, x))$ and $\lambda x', y'.f(\lambda z'.h(y', z', x'),\ h(y', x', x'))$, $\mathfrak{P}$ computes their generalization $\lambda x, y.f(\lambda z.Y_1(x, y, z), Y_2(x, y))$.

As one can see, the computed results are higher-order pattern generalizations of the input terms. Below we will prove it formally, when we establish soundness of $\mathfrak{P}$. The computed results are, in fact, pattern lggs. The Completeness Theorem in the Section 4 states this.

From the examples one can notice yet another advantage of using the store (besides helping in the merging): In the final system, it contains AUPs from which one can get the substitutions that show how the original terms can be obtained from the computed result.

3.2 Computation of Permuting Matchers

In this section we describe the algorithm $\mathfrak{M}$ to compute permuting matchers. It is a rule-based algorithm working on quintuples of the form $D;\ R;\ P;\ \rho;\ \pi$ (also called systems) where D is a set of domain variables, R is a set of range variables, D and R have the same cardinality and are disjoint, P is a set of matching problems of the form $\{s_1 \Rightarrow t_1, \ldots, s_m \Rightarrow t_m\}$, and ρ and π are substitutions (computed so far) mapping variables to variables. Here 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. The rules are the following:

Dec-M: **Decomposition**

$$D;\ R;\ \{h_1(t_1, \ldots, t_m) \Rightarrow h_2(s_1, \ldots, s_m)\} \cup P;\ \rho;\ \pi \Longrightarrow$$
$$D;\ R;\ \{t_1 \Rightarrow s_1, \ldots, t_m \Rightarrow s_m\} \cup P;\ \rho;\ \pi,$$

where each of h_1 and h_2 is a constant or a variable, and $h_1 \notin D$ or $h_2 \notin R$, and $h_1\pi = h_2\rho$. These conditions make this rule disjoint from the Per-M rule.

Abs-M: **Abstraction**

$$D;\ R;\ \{\lambda x.t \Rightarrow \lambda y.s\} \cup P;\ \rho;\ \pi \Longrightarrow D;\ R;\ \{t \Rightarrow s\} \cup P;\ \rho\{y \mapsto x\};\ \pi.$$

Per-M: **Permuting**

$$\{x\} \cup D;\ \{y\} \cup R;\ \{x(t_1, \ldots, t_m) \Rightarrow y(s_1, \ldots, s_m)\} \cup P;\ \rho;\ \pi \Longrightarrow$$
$$D;\ R;\ \{t_1 \Rightarrow s_1, \ldots, t_m \Rightarrow s_m\} \cup P;\ \rho;\ \pi\{x \mapsto y\},$$

where x and y have the same type.

Like in the rules for anti-unification above, also here each occurrence of λ binds a unique variable. The input for $\mathfrak{M}$ is initialized in the Mer rule, which needs to compute $\mathsf{match}(D, R, \{t_1 \Rightarrow s_1, t_2 \Rightarrow s_2\})$. The algorithm has the following steps:

1. Domain/range separation: To make sure that they do not share elements, we rename the domain variables with fresh ones, if necessary. It is not a restriction: If ν is such a renaming substitution, then μ is a permuting matcher of $\{s_1\nu \Rightarrow t_1, s_2\nu \Rightarrow t_2\}$ from $D\nu$ to R iff $(\nu\mu)|_D$ is a permuting matcher of $\{s_1 \Rightarrow t_1, s_2 \Rightarrow t_2\}$ from D to R.
2. Next, we create the initial system $D\nu;\ R;\ \{s_1\nu \Rightarrow t_1, s_2\nu \Rightarrow t_2\};\ \varnothing;\ \varnothing$ and apply the rules Dec-M, Abs-M and Per-M exhaustively. If no rule applies to a system $D;\ R;\ P;\ \rho;\ \pi$ with $P \neq \varnothing$, then it is transformed into $\bot$, called the *failure state*. The system $D;\ R;\ \varnothing;\ \rho;\ \pi$ is called the *success state*. No rule applies to it either.
3. When $\mathfrak{M}$ reaches the success state, we say that $\mathfrak{M}$ computes π. From it, we can return the permuting matcher $(\nu\pi)|_D$. When $\mathfrak{M}$ reaches the failure state, we say that it fails.

▸ **Example 3.2.** To compute the permuting matcher of $\{x(y,z) \Rightarrow x(z,y), X(y,\lambda u.u) \Rightarrow X(z,\lambda v.v)\}$ from $\{x,y,z\}$ to $\{x,y,z\}$ by $\mathfrak{M}$, first, we separate the domain and the range with $\nu = \{x \mapsto x', y \mapsto y', z \mapsto z'\}$, obtaining the initial system $\{x',y',z'\};\{x,y,z\};$ $\{x'(y',z') \Rightarrow x(z,y),\ X(y',\lambda u.u) \Rightarrow X(z,\lambda v.v)\};\varnothing;\varnothing$. Applying the rules of $\mathfrak{M}$, we obtain the success state $\varnothing;\varnothing;\varnothing;\{v \mapsto u\};\{x' \mapsto x, y' \mapsto z, z' \mapsto y\}$. Composing ν and the computed substitution we obtain $\{x \mapsto x, y \mapsto z, z \mapsto y\}$, which is the permuting matcher we were looking for.

The algorithm $\mathfrak{M}$ maintains the following invariants: (Justifications can be found in [5].)

Invariant 1: For each tuple D; R; P; $\rho;\pi$ in a derivation performed by $\mathfrak{M}$, the sets D and R are disjoint and have the same number of elements.

Invariant 2: For each tuple D; R; $\{t_1 \Rightarrow s_1, \ldots, t_m \Rightarrow s_m\}$; ρ; π in a derivation performed by $\mathfrak{M}$, $D \subseteq \cup_{i=1}^{m}\mathrm{Vars}(t_i)$ and $R \subseteq \cup_{i=1}^{m}\mathrm{Vars}(s_i)$.

Invariant 3: For each tuple D_i; R_i; P_i; ρ_i; π_i in a derivation performed by $\mathfrak{M}$ starting from D; R; P; ρ; π, the following equalities hold: $D_i \cup \mathrm{Dom}(\pi_i) = D$ and $R_i \cup \mathrm{Ran}(\pi_i) = R$.

▸ **Theorem 3.3.** *$\mathfrak{M}$ is terminating, sound, and complete.*

Proof. *Termination.* Termination of $\mathfrak{M}$ is straightforward: Each rule strictly reduces the multiset of sizes of matching problems in the tuples it operates on. Since each tuple D; R; P; ρ; π with $P \neq \varnothing$ 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.

Soundness. Soundness of $\mathfrak{M}$ means that if for a given tuple D; R; P; $\varnothing$; $\varnothing$ it computes a substitution π, then π is a permuting matcher of P from D to R. Obviously, π maps variables from D to R. It follows from the way how the Per-M rule constructs π. The fact that π is a matcher is straightforward: $\mathrm{Dom}(\pi) \cap \mathrm{Ran}(\pi) = \varnothing$, the differences between t and s for $t \Rightarrow s \in P$ 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 bijection property is more involved: The Per-M rule (namely, the fact that it removes x and y from D and R) and the first invariant guarantee 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 P (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 $\mathfrak{M}$ is sound.

Completeness. Recall that for each D, R, and P, if there exists a permuting matcher of P from D to R, then it is unique. Since we have already proved soundness of $\mathfrak{M}$, we have only to show that if there exists a permuting matcher of P from D to R, then $\mathfrak{M}$ does not fail for D; R; P; $\varnothing$; $\varnothing$. Let μ be such a matcher. Then $t\mu = s$ for all $t \Rightarrow s \in P$. This means that, if t has a form $h_1(t_1,\ldots,t_n)$, then s should be $h_2(s_1,\ldots,s_n)$ and $h_1\mu = h_2$, $t_i\mu = s_i$ 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 $\mathfrak{M}$ fails. That means that there exists the system D_k; R_k; $\{t \Rightarrow s\} \cup P_k$; ρ_k; π_k to which no rule applies. Since the steps performed by $\mathfrak{M}$ 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 $\mathfrak{M}$ applies to the system is that t and s should be, respectively, of the form $h_1(t_1,\ldots,t_n)$

and $h_2(s_1, \ldots, s_m)$ with $h_1\pi_k \neq h_2\rho_k$, where $h_1 \notin D_k$ or $h_2 \notin R_k$. Because of the uniqueness of the matcher, $\pi_k = \mu|_{D\setminus D_k}$. On the other hand, $h_1\mu = h_2\rho_k$, because μ matches t to $s\rho_k$.

Hence, we have $h_1\mu|_{D\setminus D_k} \neq h_2\rho_k$ where $h_1 \notin D_k$ or $h_2 \notin R_k$, and $h_1\mu = h_2\rho_k$. The latter means that either $h_1 \in D$ and $h_2 \in R$, or $h_1 \notin D$ and $h_2 \notin R$, because D and R are disjoint, the permuting matcher μ bijectively maps D to R, and ρ_k does not affect R.

Case 1: $h_1 \in D$ and $h_2 \in R$. Because of $h_1\mu|_{D\setminus D_k} \neq h_2\rho_k$, we have $h_1 \in D_k$. If $h_2 \notin R_k$, then there exists some $x \in D$, such that $x \neq h_1$ and $x\mu = h_2$, which contradicts the fact that μ is injective. If $h_2 \in R_k$, we get a contradiction with the condition $h_1 \notin D_k$ or $h_2 \notin R_k$. Hence, the case with $h_1 \in D$ and $h_2 \in R$ is impossible.

Case 2: $h_1 \notin D$ and $h_2 \notin R$. Then $h_1 = h_2\rho_k$ should hold, because $h_1\mu = h_2\rho_k$ and $h_1 \notin \mathrm{Dom}(\mu) = D$. We again get the contradiction, this time with $h_1\mu|_{D\setminus D_k} \neq h_2\rho_k$.

The obtained contradictions show that if there exists a permuting matcher of P from D to R, then $\mathfrak{M}$ does not fail for D; R; P; $\varnothing$; $\varnothing$, which implies completeness of $\mathfrak{M}$. ◂

▸ **Theorem 3.4.** *The algorithm $\mathfrak{M}$ has linear space and time complexity.*

Proof. For the input consisting of the sets of domain variables D, range variables R, and matching equations P, the size is the cardinality of $D \cup R$ plus the number of symbols in P.

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. At the initial step, renaming all variables in D with fresh ones can take only linear time with the help of the hash table for the renaming substitution.

After that, we perform the following linear time steps: Collecting the set of bound variables V_r appearing in the right sides of matching equations in P, constructing the initial hash tables T_D and T_R for (the renamed) D and R (we can assume that the hash functions are perfect), and constructing two hash tables for substitutions. The one for permuting substitutions 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 substitutions. 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: (Note that the substitution compositions in the rules, due to the disjointness of D and R, amounts to only adding a new pair to the existing substitution.)

By Dec-M: First, look up the value for h_1 in T_D, to make sure that $h_1 \notin D$. If D contains the entry for h_1, then look up the value for h_2 in T_R, to make sure that $h_2 \notin R$. If the latter test fails, the rule is not applicable.

Next, if either $h_1 \notin D$ or $h_2 \notin R$, then look up the value for h_1 in T_π, look up the value for h_2 in T_ρ, and compare them with each other. If the values of h_1 or h_2 are not found in the tables, then just use the corresponding h (i.e., h_1 or h_2) in the comparison.

By Abs-M: Modifying an entry in T_ρ: For a renaming substitution $\{y \mapsto x\}$, 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 substitution $\{x \mapsto y\}$, 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 $\mathfrak{M}$ at each node of the input trees are done in constant time. It implies that $\mathfrak{M}$ has linear time complexity. ◂

4 Properties of the Anti-Unification Algorithm

▸ **Theorem 4.1** (Termination). *The procedure $\mathfrak{P}$, which uses $\mathfrak{M}$ to compute permuting matchers, terminates for all input terms t and s.*

Proof. We define the measure of $A; S; \sigma$ as a pair of multisets $(M(A), M(S))$, where the multiset $M(L) = \{\min(\mathrm{Depth}(t), \mathrm{Depth}(s)) \mid X(\vec{x}) : t \triangleq s \in L\}$ for any L. Measures are compared lexicographically. Obviously, each rule in $\mathfrak{P}$ strictly reduces it. The ordering is well-founded. The procedure $\mathfrak{M}$ in the rule Mer is terminating. Hence, $\mathfrak{P}$ terminates. ◂

▸ **Theorem 4.2** (Soundness). *If $\{X : t \triangleq s\}; \varnothing; \varnothing \Longrightarrow^* \varnothing; S; \sigma$ is a derivation in $\mathfrak{P}$, 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 $A_1; S_1; \varphi \Longrightarrow A_2; S_2; \varphi\vartheta$ 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 AUP whose generalization term is not in this form. It finishes the (sketch of the) proof of (4.2).

Proving (4.2) is more involved. First, we prove that if $A_1; S_1; \varphi \Longrightarrow A_2; S_2; \varphi\vartheta$ is one step, then for any $X(\vec{x}) : t \triangleq s \in A_1 \cup S_1$, we have $X(\vec{x})\vartheta \preceq t$ and $X(\vec{x})\vartheta \preceq s$. Note that if $X(\vec{x}) : t \triangleq s$ was not transformed at this step, then this property trivially holds for it. Therefore, we assume that $X(\vec{x}) : t \triangleq s$ 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 ψ_1 and ψ_2 be substitutions defined, respectively, by $Y_i\psi_1 = \lambda\vec{x}.t_i$ and $Y_i\psi_2 = \lambda\vec{x}.s_i$ for all $1 \leqslant i \leqslant m$. Such substitutions obviously exist since the Y's introduced by the Dec rule are fresh. Then $X(\vec{x})\vartheta\psi_1 = h(t_1, \ldots, t_m)$, $X(\vec{x})\vartheta\psi_2 = h(s_1, \ldots, s_m)$ 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 $\psi_1 = \{X' \mapsto \lambda\vec{x}, y.t'\}$ and $\psi_2 = \{X' \mapsto \lambda\vec{x}, y.s'\}$. Then $X(\vec{x})\vartheta\psi_1 = \lambda y.t' = t$, $X(\vec{x})\vartheta\psi_2 = \lambda y.s' = s$, 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 $\psi_1 = \{Y \mapsto \lambda\vec{y}.t\}$ and $\psi_2 = \{Y \mapsto \lambda\vec{y}.s\}$. Then $X(\vec{x})\vartheta\psi_1 = t$, $X(\vec{x})\vartheta\psi_2 = s$, and, hence, $X(\vec{x})\vartheta \preceq t$ and $X(\vec{x})\vartheta \preceq s$.

If Mer applies, then there exists $Y(\vec{y}) : t' \triangleq s' \in S_1$ such that $\mathsf{match}(\{\vec{x}\}, \{\vec{y}\}, t \Rightarrow t', s \Rightarrow s')$ 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 $Y(\vec{y}) : t' \triangleq s'$, let $\psi_1 = \{X \mapsto \lambda\vec{x}.t\}$ and $\psi_2 = \{X \mapsto \lambda\vec{x}.s\}$. Then $Y(\vec{y})\vartheta\psi_1 = (\lambda\vec{x}.t)(\vec{x}\pi) = t\pi = t'$, $Y(\vec{y})\vartheta\psi_2 = (\lambda\vec{x}.s)(\vec{x}\pi) = s\pi = s'$, 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 $A_0; S_0; \vartheta_0 \Longrightarrow^* \varnothing; S_n; \vartheta_0\vartheta_1 \cdots \vartheta_n$ is a derivation in $\mathfrak{P}$, then for any $X(\vec{x}) : t \triangleq s \in A_0 \cup S_0$ 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 $A_0; S_0; \vartheta_0 \Longrightarrow A_1; S_1; \vartheta_0\vartheta_1 \Longrightarrow^* \varnothing; S_n; \vartheta_0\vartheta_1 \cdots \vartheta_n$ 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 $X(\vec{x}) : t \triangleq s$ be an AUP selected for transformation at the current step. (Again, the property trivially holds for the AUPs 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 $U \in \mathrm{Vars}(\mathrm{Ran}(\vartheta_2^n))$, then there is an AUP of the form $U(\vec{u}) : t' \triangleq s' \in S_n$. 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 A_1 contains the AUP $X'(\vec{x}, y) : t'\{y_1 \mapsto y\} \triangleq s'\{y_2 \mapsto y\}$. 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 $Y(\vec{y}) : t' \triangleq s' \in S_0$ such that $\mathsf{match}(\{\vec{x}\}, \{\vec{y}\}, t \Rightarrow t', s \Rightarrow s')$ 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 $\mathfrak{P}$ for $X : t \triangleq s$ generalizes both t and s. We call $X\sigma$, a *generalization of t and s computed by $\mathfrak{P}$*. Moreover, given a derivation $\{X : t \triangleq s\}; \varnothing; \varnothing \Longrightarrow^* \varnothing; S; \sigma$ in $\mathfrak{P}$, we say that

- σ is a *substitution computed by $\mathfrak{P}$ for $X : t \triangleq s$;*
- the restriction of σ on X, $\sigma|_X$, is an *anti-unifier of $X : t \triangleq s$ computed by $\mathfrak{P}$.*

▸ **Theorem 4.3** (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 $\lambda\vec{x}.s \preceq \lambda\vec{x}.X(\vec{x})\sigma$, where σ is an anti-unifier of $X : \lambda\vec{x}.t_1 \triangleq \lambda\vec{x}.t_2$ computed by $\mathfrak{P}$.*

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 $\mathrm{Head}(t_1) \neq \mathrm{Head}(t_2)$, or $\mathrm{Head}(t_1) = \mathrm{Head}(t_2) \notin \vec{x}$. 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 system $\{X'(\vec{x}, x) : t_1' \triangleq t_2'\{x \mapsto y\}\}; \varnothing; \sigma_1$, 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 $\mathrm{Head}(t_1) \neq \mathrm{Head}(t_2)$, or $\mathrm{Head}(t_1) = \mathrm{Head}(t_2) = h$. 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 $\mathrm{Head}(t_1) = \mathrm{Head}(t_2)$. 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 $\{Y_i(\vec{x}) : t_i^1 \triangleq t_i^2 \mid 1 \leqslant i \leqslant n\}; \varnothing; \sigma_0$, 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 $\Delta = \{X(\vec{x}) : t_1 \triangleq t_2\}; \varnothing; \sigma_0 \Longrightarrow \{Y_i(\vec{x}) : t_i^1 \triangleq t_i^2 \mid 1 \leqslant i \leqslant n\}; \varnothing; \sigma_0 \Longrightarrow^* \varnothing; S_n; \sigma_0\sigma_1 \cdots \sigma_n$.

Let for any term t, $t|_p$ denote the subterm of t at position p. 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 p_1 and p_2, it indicates that

(a) $t_1|_{p_1}$ and $t_1|_{p_2}$ differ from each other by a permutation of variables bound in t_1,
(b) $t_2|_{p_1}$ and $t_2|_{p_2}$ differ from each other by the same (modulo variable renaming) permutation of variables bound in t_2,
(c) the path to p_1 is the same (modulo bound variable renaming) in t_1 and t_2. It equals (modulo bound variable renaming) the path to p_1 in s, and
(d) the path to p_2 is the same (modulo bound variable renaming) in t_1 and t_2. It equals (modulo bound variable renaming) the path to p_2 in s.

Then, because of (c) and (d), we should have two AUPs in S_n: One, between (renamed variants of) $t_1|_{p_1}$ and $t_2|_{p_1}$, and the other one between (renamed variants of) $t_1|_{p_2}$ and $t_2|_{p_2}$. The possible renaming of variables is caused by the fact that Abs might have been applied to obtain the AUPs. Let those AUPs be $Z(\vec{z_1}) : r_1^1 \triangleq r_1^2$ and $Z'(\vec{z_2}) : r_2^1 \triangleq r_2^2$. The conditions (a) and (b) make sure that $\mathsf{match}(\{\vec{z_1}\}, \{\vec{z_2}\}, \{r_1^1 \Rightarrow r_2^1, r_1^2 \Rightarrow r_2^2\})$ 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 $\Delta' = \{X(\vec{x}) : t_1 \triangleq t_2\}; \varnothing; \sigma_0 \Longrightarrow \{Y_i(\vec{x}) : t_i^1 \triangleq t_i^2 \mid 1 \leqslant i \leqslant n\}; \varnothing; \sigma_0 \Longrightarrow^* \varnothing; S_n; \sigma_0\sigma_1 \cdots \sigma_n \Longrightarrow^* \varnothing; S_{n+m}; \sigma_0\sigma_1 \cdots \sigma_n\sigma_1' \cdots \sigma_m'$, 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 $\lambda\vec{x}.s \preceq \lambda\vec{x}.X(\vec{x})\sigma$, which finishes the proof. ◂

Depending which AUP is selected to perform a step, there can be different derivations in $\mathfrak{P}$ starting from the same AUP, leading to different generalizations. The next theorem states that all those generalizations are equivalent.

▸ **Theorem 4.4** (Uniqueness Modulo $\simeq$). *Let* $\{X : t \triangleq s\}; \varnothing; \varnothing \Longrightarrow^* \varnothing; S_1; \sigma_1$ *and* $\{X : t \triangleq s\}; \varnothing; \varnothing \Longrightarrow^* \varnothing; S_2; \sigma_2$ *be two maximal derivations in* $\mathfrak{P}$ *from* $X : t \triangleq s$. *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 AUPs for each rule) then the result remains the same: If $\Delta_1 = A_1; S_1; \sigma_1 \Longrightarrow_{\mathsf{R1}} A_2; S_2; \sigma_1\vartheta_1 \Longrightarrow_{\mathsf{R2}} A_3; S_3; \sigma_1\vartheta_1\vartheta_2$ and $\Delta_2 = A_1; S_1; \sigma_1 \Longrightarrow_{\mathsf{R2}}$

$A'_2; S'_2; \sigma_1\vartheta_2 \Longrightarrow_{\mathsf{R1}} A'_3; S'_3; \sigma_1\vartheta_2\vartheta_1$ are two two-step derivations, where R1 and R2 are (not necessarily different) rules and each of them transforms the same AUP(s) in both Δ_1 and Δ_2, then $A_3 = A'_3$, $S_3 = S'_3$, 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 AUP in a unique way. We show that it is irrelevant in which order we perform matching in the Merge rule.

Let $A; \{Z(\vec{z}) : t_1 \triangleq s_1, Y(\vec{y}) : t_2 \triangleq s_2\} \cup S; \sigma \Longrightarrow A; \{Z(\vec{z}) : t_1 \triangleq s_1\} \cup S; \sigma\{Y \mapsto \lambda\vec{y}.Z(\vec{z}\pi)\}$ be the merging step with $\pi = \mathsf{match}(\{\vec{z}\}, \{\vec{y}\}, \{t_1 \Rightarrow t_2, s_1 \Rightarrow s_2\})$. If we do it in the other way around, we would get the step $A; \{Z(\vec{z}) : t_1 \triangleq s_1, Y(\vec{y}) : t_2 \triangleq s_2\} \cup S; \sigma \Longrightarrow A; \{Y(\vec{y}) : t_2 \triangleq s_2\} \cup S; \sigma\{Z \mapsto \lambda\vec{z}.Y(\vec{y}\mu)\}$, where $\mu = \mathsf{match}(\{\vec{y}\}, \{\vec{z}\}, \{t_2 \Rightarrow t_1, s_2 \Rightarrow s_1\})$. 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:

$$\lambda\vec{y}.Z(\vec{z}\pi)\rho_2 = \lambda\vec{y}.Y(\vec{y}). \tag{1}$$

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 (1) is proved.

Next, starting from $X\vartheta_1\rho_2$, we can transform it as $X\vartheta_1\rho_2 = X\sigma\rho_1\rho_2 = X\sigma\{Y \mapsto \lambda\vec{y}.Z(\vec{z}\pi)\rho_2, Z \mapsto \lambda\vec{z}.Y(\vec{y}\pi^{-1})\} =_{\text{by (1)}} X\sigma\{Y \mapsto \lambda\vec{y}.Z(\vec{z}\pi)\rho_2, Z \mapsto \lambda\vec{z}.Y(\vec{y}\pi^{-1})\} = X\sigma\{Y \mapsto \lambda\vec{y}.Y(\vec{y}), Z \mapsto \lambda\vec{z}.Y(\vec{y}\pi^{-1})\} = X\sigma\{Y \mapsto \lambda\vec{y}.Y(\vec{y})\}\{Z \mapsto \lambda\vec{z}.Y(\vec{y}\pi^{-1})\}$. 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 $\mathfrak{P}$ computes their generalization, a higher-order pattern, which is less general than any other higher-order pattern which generalizes t and s. The next theorem is about its complexity:

▸ **Theorem 4.5** (Complexity of $\mathfrak{P}$). *The algorithm $\mathfrak{P}$, when using $\mathfrak{M}$ to compute permuting matchers, has space complexity $O(n)$ and time complexity $O(n^3)$, where n is the size (the number of symbols) of input.*

Proof. We can keep the substitutions in the systems in triangular form. Then the size of systems is linear in the size of input. Only at the end we will apply the computed anti-unifier to the corresponding generalization variable to return the generalization: Having the substitution $[X \mapsto t_0, Y_1 \mapsto t_1, \ldots, Y_n \mapsto t_n]$, we need to compute $t_0\{Y_1 \mapsto t_1\} \cdots \{Y_n \mapsto t_n\}$. Its size does not exceed the size on the input. Hence, the space complexity is linear.

For proving the cubic time complexity, we can assume that the applications of the Mer rule are postponed till the end. The number of application of the other rules is bounded by the size of the input. Abs involves renaming which can be done in linear time. Sol requires selection of variables that occur freely in terms, which also needs linear time. Composition of substitutions is just appending a new binding at the end of the existing triangular substitution. As for the Mer rule, it can be called at most quadratic number of times. At each application it calls $\mathfrak{M}$ which itself requires linear time. Hence, the cubic complexity of applications of Mer dominates the complexity of applications of the other rules. The

last step, constructing the generalization $t_0\{Y_1 \mapsto t_1\}\cdots\{Y_n \mapsto t_n\}$ from the computed triangular substitution, requires linear number of substitution applications. Each application traverses the term, replaces all occurrences of Y_i with t_i, and performs β-reduction (i.e, bound variable permutation). Traversal, replacement, and β-reduction can take at most quadratic time. Therefore, the complexity of this last step is also cubic. It implies that $\mathfrak{P}$ has the $O(n^3)$ time complexity.

Note that if the input does not satisfy the condition each bound variable to be unique (on which both $\mathfrak{P}$ and $\mathfrak{M}$ rely), we can rename the variables before calling $\mathfrak{P}$. 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. ◂

5 Final Remarks

One can observe that $\mathfrak{P}$ can be adapted with a relatively little effort to work on untyped terms (cf. the formulation of the unification algorithm both for untyped and simply-typed patterns in [27]). One thing to be added is lazy η-expansion: The AUP of the form $X(\vec{x}) : \lambda y.t \triangleq h(s_1, \ldots, s_m)$ should be transformed into $X(\vec{x}) : \lambda y.t \triangleq \lambda z.h(s_1, \ldots, s_m, z)$ for a fresh z. (Dually for abstractions in the right hand side.) The expansion should be performed both in $\mathfrak{P}$ and $\mathfrak{M}$. In addition, Sol needs an extra condition for the case when $\mathrm{Head}(t) = \mathrm{Head}(s)$ but the terms have different number of arguments such as, e.g., in $f(a, x)$ and $f(b, x, y)$.

The anti-unification algorithm has been implemented (both for simply-typed and untyped terms, without perfect hashing) in Java. It can be used online or can be downloaded freely from `http://www.risc.jku.at/projects/stout/software/hoau.php`.

As for the related topics, we would mention nominal anti-unification. Several authors explored relationship between nominal terms and higher-order patterns (see, e.g., [11, 13, 20, 21] among others), proposing translations between them in the context of unification. However, it is not immediately clear how to reuse those translations for anti-unification, in particular, how to get nominal generalizations from pattern generalizations.

Studying anti-unification in the calculi with more complex type systems, such as the extension of the system F with subtyping $F_{<:}$ [10], would be a very interesting direction of future work, because it may have applications in clone detection and refactoring for the functional programming languages in the ML family.

References

1 M. Alpuente, S. Escobar, J. Meseguer, and P. Ojeda. A modular equational generalization algorithm. In *LOPSTR*, volume 5438 of *LNCS*, pages 24–39. Springer, 2008.

2 M. Alpuente, S. Escobar, J. Meseguer, and P. Ojeda. Order-sorted generalization. *Electr. Notes Theor. Comput. Sci.*, 246:27–38, 2009.

3 E. Armengol and E. Plaza. Bottom-up induction of feature terms. *Machine Learning*, 41(3):259–294, 2000.

4 H. Barendregt. Introduction to generalized type systems. *J. Funct. Program.*, 1(2):125–154, 1991.

5 A. Baumgartner, T. Kutsia, J. Levy, and M. Villaret. A variant of higher-order anti-unification. Technical Report 12-19, RISC, Johannes Kepler University Linz, 2012. `http://www.risc.jku.at/publications/download/risc_4675/hoau.pdf`.

6 P. Bulychev. Duplicate code detection using Clone Digger. *Python Mag.*, 9:18–24, 2008.

7 P. Bulychev and M. Minea. An evaluation of duplicate code detection using anti-unification. In *Proc. 3rd International Workshop on Software Clones*, 2009.

8 P. E. Bulychev, E. V. Kostylev, and V. A. Zakharov. Anti-unification algorithms and their applications in program analysis. In *Ershov Memorial Conference*, volume 5947 of *LNCS*, pages 413–423. Springer, 2009.

9 J. Burghardt. E-generalization using grammars. *Artif. Intell.*, 165(1):1–35, 2005.

10 L. Cardelli, S. Martini, J. C. Mitchell, and A. Scedrov. An extension of System F with subtyping. *Inf. Comput.*, 109(1/2):4–56, 1994.

11 J. Cheney. Relating higher-order pattern unification and nominal unification. In *UNIF'05*, pages 104–119, 2005.

12 G. Dowek. Higher-order unification and matching. In *Handbook of Automated Reasoning*, pages 1009–1062. Elsevier and MIT Press, 2001.

13 G. Dowek, M. J. Gabbay, and D. 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.

14 C. Feng and S. Muggleton. Towards inductive generalization in higher order logic. In *ML*, pages 154–162. Morgan Kaufmann, 1992.

15 R. W. Hasker. *The Replay of Program Derivations*. PhD thesis, University of Illionois at Urbana-Champaign, 1995.

16 K. Hirata, T. Ogawa, and M. Harao. Generalization algorithms for second-order terms. In *ILP*, volume 3194 of *LNCS*, pages 147–163. Springer, 2004.

17 G. Huet. *Résolution d'équations dans des langages d'ordre* $1, 2, \ldots, \omega$. PhD thesis, Université Paris VII, September 1976.

18 U. Krumnack, A. Schwering, H. Gust, and K.-U. Kühnberger. Restricted higher-order anti-unification for analogy making. In *AUS-AI*, volume 4830 of *LNCS*, pages 273–282. Springer, 2007.

19 T. Kutsia, J. Levy, and M. Villaret. Anti-unification for unranked terms and hedges. In *RTA*, volume 10 of *LIPIcs*, pages 219–234, 2011.

20 J. Levy and M. Villaret. Nominal unification from a higher-order perspective. In *RTA*, volume 5117 of *LNCS*, pages 246–260. Springer, 2008.

21 J. Levy and M. Villaret. Nominal unification from a higher-order perspective. *ACM Trans. Comput. Log.*, 13(2):10, 2012.

22 H. Li and S. J. Thompson. Similar code detection and elimination for Erlang programs. In *PADL*, volume 5937 of *LNCS*, pages 104–118. Springer, 2010.

23 J. Lu, J. Mylopoulos, M. Harao, and M. Hagiya. Higher order generalization and its application in program verification. *Ann. Math. Artif. Intell.*, 28(1-4):107–126, 2000.

24 G. Mendel-Gleason. *Types and Verification for Infinite State Systems*. PhD thesis, Dublin City University, 2012.

25 D. Miller. A logic programming language with lambda-abstraction, function variables, and simple unification. *J. Log. Comput.*, 1(4):497–536, 1991.

26 G. Nadathur and N. Linnell. Practical higher-order pattern unification with on-the-fly raising. In *ICLP*, volume 3668 of *LNCS*, pages 371–386. Springer, 2005.

27 T. Nipkow. Functional unification of higher-order patterns. In *LICS*, pages 64–74. IEEE Computer Society, 1993.

28 F. Pfenning. Unification and anti-unification in the calculus of constructions. In *LICS*, pages 74–85. IEEE Computer Society, 1991.

29 G. D. Plotkin. A note on inductive generalization. *Machine Intel.*, 5(1):153–163, 1970.

30 J. C. Reynolds. Transformational systems and the algebraic structure of atomic formulas. *Machine Intel.*, 5(1):135–151, 1970.

31 U. Schmid. *Inductive Synthesis of Functional Programs, Universal Planning, Folding of Finite Programs, and Schema Abstraction by Analogical Reasoning*, volume 2654 of *LNCS*. Springer, 2003.