A Variant of Higher-Order Anti-Unification

Alexander Baumgartner¹, Temur Kutsia¹, Jordi Levy², and Mateu Villaret³

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 $\lambda$-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 $MA$, 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 $\lambda$-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 $\lambda$Prolog, 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 $\lambda$-calculus (in Sect. 3). The input of the algorithm are arbitrary terms in $\eta$-long $\beta$-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 $\alpha$-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 $\lambda$-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
  $\lambda$-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 $\lambda$-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.

\section{Preliminaries}

In higher-order signatures we have types constructed from a set of basic types (typically $\delta$)
using the grammar $\tau ::= \delta | \tau \to \tau$, 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.

$\lambda$-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 $\overline{t}$ as a short-hand for $x_1, \ldots, x_n$.

Other standard notions of the simply typed $\lambda$-calculus, like bound and free occurrences
of variables, $\alpha$-conversion, $\beta$-reduction, $\eta$-long $\beta$-normal form, etc. are defined as usual
(see [12]). By default, terms are assumed to be written in $\eta$-long $\beta$-normal form. Therefore,
al l terms have the form $\lambda x_1, \ldots, x_n . h(t_1, \ldots, t_m)$, where $n, m \geq 0$, $h$ 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.
A Variant of Higher-Order Anti-Unification

The set of free variables of a term $t$ is denoted by $\text{Vars}(t)$. When we write an equality between two $\lambda$-terms, we mean that they are equivalent modulo $\alpha$, $\beta$ and $\eta$ equivalence.

The depth of a term $t$, denoted $\text{Depth}(t)$ is defined recursively as follows: $\text{Depth}(x) = 1$, $\text{Depth}(t_1 \ldots t_n) = 1 + \text{max} \{ \text{Depth}(t_i) \}$, and $\text{Depth}(\lambda x.t) = 1 + \text{Depth}(t)$. For a term $t = \lambda x_1, \ldots, x_n. h(t_1, \ldots, t_m)$ with $n, m \geq 0$, its head is defined as $\text{Head}(t) = h$.

Positions in $\lambda$-terms are defined with respect to their tree representation in the usual way, as string of integers. For instance, in the term $\lambda x.\lambda y.(\lambda z. h(z, y), x), \lambda u.g(u))$, the symbol $f$ stands in the position $e$ (the empty sequence), the occurrence of $\lambda x$. stands in the position 1, the bound occurrence of $y$ in 1.1.1.1.2, the bound occurrence of $u$ in 2.1.1, etc.

The path to a position in a $\lambda$-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 $\lambda$-term where, when written in $\eta$-long $\beta$-normal form, all free variable occurrences are applied to lists of pairwise distinct ($\eta$-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 $\lambda x.f(X(X(x))), f(X(c), c)$ 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 $\text{Dom}(\sigma)$ in $t$. We write $x_1\sigma, \ldots, x_n\sigma$ for $x_1, \ldots, x_n$. Similarly, for a set of terms $S$, we define $S\sigma = \{t\sigma \mid t \in S\}$. The composition of $\sigma$ and $\vartheta$ is written as juxtaposition $\sigma \vartheta$. Yet another standard operation, restriction of a substitution $\sigma$ to a set of variables $S$, is denoted by $\sigma|_S$.

A substitution $\sigma_1$ is more general than $\sigma_2$, written $\sigma_1 \preceq \sigma_2$, if there exists $\vartheta$ such that $X\sigma_1\vartheta = X\sigma_2$ for all $X \in \text{Dom}(\sigma_1) \cup \text{Dom}(\sigma_2)$. 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 $\sim$. We overload $\preceq$ by defining $s \preceq t$ if there exists a substitution $\sigma$ 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 $t_1 \preceq t$ 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 $s \prec t$.

An anti-unification problem (shortly AUP) is a triple $X(\overline{x}) : t \not\equiv s$ where

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

The variable $X$ is called a generalization variable. The term $X(\overline{x})$ is called the generalization term. The variables that belong to $\overline{x}$, 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 $X(\overline{x}) : t \not\equiv s$ is a substitution $\sigma$ such that $\text{Dom}(\sigma) = \{X\}$ and $\lambda \overline{x}.X(\overline{x})\sigma$ is a term which generalizes both $\lambda \overline{x}.t$ and $\lambda \overline{x}.s$.

An anti-unifier of $X(\overline{x}) : t \not\equiv s$ is least general (or most specific) if there is no anti-unifier $\vartheta$ of the same problem that satisfies $\sigma \prec \vartheta$. Obviously, if $\sigma$ is a least general anti-unifier of an AUP $X(\overline{x}) : t \not\equiv s$, then $\lambda \overline{x}.X(\overline{x})\sigma$ is an lgg of $\lambda \overline{x}.t$ and $\lambda \overline{x}.s$. 
Here we consider a variant of higher-order anti-unification problem:

**Given:** Higher-order terms $t$ and $s$ of the same type in $\eta$-long $\beta$-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 $X(\vec{x}) : t \neq s$, 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 $\{X_i(\vec{x}_i) : t_i \neq s_1, \ldots, X_n(\vec{x}_n) : t_n \neq s_n\}$ where each $X_i$ occurs in $A \cup S$ only once, $S$ is a set of already solved AUPs (the store), and $\sigma$ 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 $\lambda$ binds a distinct name variable (in other words, all names of bound variables are distinct).

**Dec:** Decomposition

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

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

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

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

**Sol:** Solve

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

where $t$ and $s$ are of a basic type, $\text{Head}(t) \neq \text{Head}(s)$ or $\text{Head}(t) = \text{Head}(s) = Z \notin \vec{x}$, $\vec{y}$ 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

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

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 $X(\vec{x}) : t \neq s \in A \cup S$, 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 $\eta$-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 \(\lambda\) 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 \(X'(\bar{t}, y)\) 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 \(\alpha\)-lgs.

The Mer rule requires solving a matching problem \(\{t_1 \Rightarrow s_1, t_2 \Rightarrow s_2\}\) with the substitution \(\pi\) which bijectively maps the variables from \(\bar{t}\) to the variables from \(\bar{s}\). 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 permutable 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 \(\text{match}(D, R, P)\), we denote such a permutable matcher of \(P\) from \(D\) to \(R\), when it exists. Otherwise, \(\text{match}(D, R, P) = \bot\). An algorithm that computes it is given in Sect. 3.2 below.

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

» Example 3.1. A couple of examples illustrating the generalizations computed by \(\Psi\):
For $\lambda x, y.f(\lambda z. U(z, y, x), U(x, y, x))$ and $\lambda x'. y'. f(\lambda z'. h(y', z', x'), h(y', x', x'))$, $\Psi$ 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 $\Psi$. The computed results are, in fact, pattern lgs. 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 $\rho$ and $\pi$ are substitutions (computed so far) mapping variables to variables. Here $\rho$ is supposed to keep bound variable renamings to deal with abstractions, while in $\pi$ we compute the permuting matcher to be returned in case of success. The rules are the following:

- **Dec-M: Decomposition**
  
  $\begin{align*}
  D; R; \{h_1(t_1, \ldots, t_m) \Rightarrow h_2(s_1, \ldots, s_m)\} & \ni P; \rho; \pi \\
  D; R; \{t_1 \Rightarrow s_1, \ldots, t_m \Rightarrow s_m\} & \cup P; \rho; \pi
  \end{align*}$

  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**
  
  $\begin{align*}
  D; R; \{\lambda x.t \Rightarrow \lambda y.s\} & \ni P; \rho; \pi \\
  D; R; \{t \Rightarrow s\} & \cup P; \rho(y \mapsto x); \pi
  \end{align*}$

- **Per-M: Permuting**
  
  $\begin{align*}
  \{x\} \ni D; \{y\} \ni R; \{x(t_1, \ldots, t_m) \Rightarrow y(s_1, \ldots, s_m)\} & \ni P; \rho; \pi \\
  D; R; \{t_1 \Rightarrow s_1, \ldots, t_m \Rightarrow s_m\} & \cup P; \rho; \pi[x \mapsto y],
  \end{align*}$

  where $x$ and $y$ have the same type.

Like in the rules for anti-unification above, also here each occurrence of $\lambda$ binds a unique variable. The input for $\mathfrak{M}$ is initialized in the Mer rule, which needs to compute $\text{match}(D, R, \{t_1 \Rightarrow s_1, t_2 \Rightarrow s_2\})$. 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 $\mu$ is such a renaming substitution, then $\mu$ is a permuting matcher of $\{s_1\mu \Rightarrow t_1, s_2\mu \Rightarrow t_2\}$ from $D\mu$ 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 $D\nu; R; \{s_1\nu \Rightarrow t_1, s_2\nu \Rightarrow t_2\}; \emptyset; \emptyset$ 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 $P \neq \emptyset$, then it is transformed into $\bot$, called the failure state. The system $D; R; \emptyset; \emptyset; \rho; \pi$ 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 $\pi$. 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 \( \{x', y', z'; [x, y, z]; x'(y', z') \Rightarrow x(z, y), X(y', \lambda u. u) = X(z, \lambda v. v)\}; \mathcal{Q}; \emptyset \). Applying the rules of \( \mathfrak{M} \), we obtain the success state \( \emptyset; \emptyset; \emptyset; [v \mapsto u]; \{x' \mapsto x, y' \mapsto z, z' \mapsto y\} \). Composing \( \nu \) 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 \( D; R; \{t_1 \mapsto s_1, \ldots, t_m \mapsto s_m\}; \rho; \pi \) in a derivation performed by \( \mathfrak{M} \), \( D \subseteq \bigcup_{i=1}^{m} \text{Vars}(t_i) \) and \( R \subseteq \bigcup_{i=1}^{m} \text{Vars}(s_i) \).

Invariant 3: For each tuple \( D_i; R; P; \rho_i; \pi_i \) in a derivation performed by \( \mathfrak{M} \) starting from \( D; R; P; \rho; \pi \), the following equalities hold: \( D_i \cup \text{Dom}(\pi_i) = D \) and \( R_i \cup \text{Ran}(\pi_i) = R \).

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 \( D; R; P; \rho; \pi \) with \( P \neq \emptyset \) 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 \( D; R; P; \emptyset; \emptyset \) it computes a substitution \( \pi \), then \( \pi \) is a permuting matcher of \( P \) from \( D \) to \( R \). Obviously, \( \pi \) maps variables from \( D \) to \( R \). It follows from the way how the \( \text{Per-M} \) rule constructs \( \pi \). The fact that \( \pi \) is a matcher is straightforward: \( \text{Dom}(\pi) \cap \text{Ran}(\pi) = \emptyset \), the differences between \( t \) and \( s \) for \( t = s \in P \) are either repaired by the bindings from \( \pi \) constructed by \( \text{Per-M} \), or the differences are \( \alpha \)-equivalences repaired by the bindings from \( \rho \) constructed by \( \text{Abs-M} \), or the failure occurs since no rule can be applied. The bijection property is more involved: The \( \text{Per-M} \) 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 \( \emptyset \) by applications of \( \text{Per-M} \) (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 \( \pi \). Hence, \( \pi \) 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 \( D; R; \emptyset; \emptyset \) and \( \rho \). Let \( \mu \) be such a matcher. Then \( t_\mu = s \) for all \( t = s \in P \). 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 \( h_1 \mu = h_2, t_\mu = s_i \) for all \( 1 \leq i \leq n \). 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 \( D_k; R_k; \{t = s\} \cup P_k; \rho_k; \pi_k \) 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, \(\pi_k = \mu |_{D_k,D_k}\). On the other hand, \(h_1 \mu = h_2 \rho_k\), because \(\mu\) matches \(t\) to \(s\rho_k\).

Hence, we have \(h_1 |_{D_k,D_k} \neq h_2 \rho_k\) 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 \(\mu\) bijectively maps \(D\) to \(R\), and \(\rho_k\) does not affect \(R\).

Case 1: \(h_1 \in D\) and \(h_2 \in R\). Because of \(h_1 |_{D_k,D_k} \neq h_2 \rho_k\), 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 \(\mu\) 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 \(h_1 \notin \text{Dom}(\mu) = D\). We again get the contradiction, this time with \(h_1 |_{D_k,D_k} \neq h_2 \rho_k\).

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

**Theorem 3.4.** The algorithm \(\mathcal{M}\) 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 \(T_\pi\). Its set of keys is \(D\). We can reuse the same hash function as for \(T_D\). Each address in \(T_\pi\) is initialized with null. Another table, \(T_\rho\), 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 \(T_\pi\), look up the value for \(h_2\) in \(T_\rho\), 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 \(T_\rho\): For a renaming substitution \(\{ y \mapsto x \}\), we put \(x\) in the table at the address corresponding to the hash index of \(y\): \(T_\rho[\text{hash}(y)] = x\). Since all bound variables are distinct, we will not have to modify the same entry in \(T_\rho\) again.

- **By Per-M:** Modifying an entry for \(x\) in \(T_\pi\): For a substitution \(\{ x \mapsto y \}\), we put \(y\) in the address corresponding to the hash index of \(x\): \(T_\pi[\text{hash}(x)] = y\). As we destroy the entries for \(x\) in \(T_D\) and for \(y\) in \(T_R\), we will not modify the same entry again.
Therefore, we assume that \( \text{Abs} \) if \( \text{Sol} \) of patterns under substitution application and from the fact that substitutions in the rules pattern, then \( \text{order pattern and, second, at each step} \)

\[
\text{Theorem 4.2}
\]

To prove that \( \text{X} \sigma \) is a higher-order pattern in \( \eta \)-long \( \beta \)-normal form, then \( \text{X} \varphi \theta \) 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 \( \text{X} \sigma \) being in \( \eta \)-long \( \beta \)-normal form, this is guaranteed by the series of applications of the \( \text{Abs} \) rule, even if \( \text{Dec} \) introduces an \( \text{AUP} \) 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 \( \text{A}_1; \text{S}_1; \varphi \Rightarrow \text{A}_2; \text{S}_2; \varphi \theta \) is one step, then for any \( \text{X} (\overline{x}) : t \neq s \in \text{A}_1 \cup \text{S}_1 \), we have \( \text{X} (\overline{x}) \varphi \theta \leq t \) and \( \text{X} (\overline{x}) \varphi \theta \leq s \). Note that if \( \text{X} (\overline{x}) : t \neq s \) was not transformed at this step, then this property trivially holds for it. Therefore, we assume that \( \text{X} (\overline{x}) : t \neq s \) is selected and prove the property for each rule:

\[
\text{Dec: Here } t = \text{h}(t_1, \ldots, t_m), s = \text{h}(s_1, \ldots, s_m), \text{ and } \vartheta = \{ \text{X} \mapsto \lambda \overline{x}. \text{h}(\text{Y}_1(\overline{x}), \ldots, \text{Y}_m(\overline{x})) \}.
\]

Then \( \text{X} (\overline{x}) \varphi \theta = \text{h}(Y_1(\overline{x}), \ldots, Y_m(\overline{x})) \). Let \( \psi_1 \) and \( \psi_2 \) be substitutions defined, respectively, by \( Y_1 \psi_1 = \lambda \overline{x}. t_1 \) and \( Y_i \psi_2 = \lambda \overline{x}. s_i \) for all \( 1 \leq i \leq m \). Such substitutions obviously exist since the \( \text{Y} \)’s introduced by the \( \text{Dec} \) rule are fresh. Then \( \text{X} (\overline{x}) \varphi \psi_1 = \text{h}(t_1, \ldots, t_m) \), \( \text{X} (\overline{x}) \psi_2 = \text{h}(s_1, \ldots, s_m) \) and, hence, \( \text{X} (\overline{x}) \varphi \theta \leq t \) and \( \text{X} (\overline{x}) \varphi \theta \leq s \).

\[
\text{Abs: Here } t = \lambda y_1. t', s = \lambda y_2. s', \text{ and } \vartheta = \{ \text{X} \mapsto \lambda \overline{x}. \text{Y}(\overline{x}) \}. \text{ Then } \text{X} (\overline{x}) \varphi \theta = \lambda y. \text{X}'(\overline{x}). y. \text{Y}(\overline{x}) \}. \text{ Let } \psi_1 = \{ \text{X}' \mapsto \lambda \overline{x}. y. t' \} \text{ and } \psi_2 = \{ \text{X}' \mapsto \lambda \overline{x}. y. s' \}. \text{ Then } \text{X} (\overline{x}) \psi_1 = \lambda y. t' = t, \text{X} (\overline{x}) \psi_2 = \lambda y. s' = s, \text{ and, hence, } \text{X} (\overline{x}) \varphi \theta \leq t \text{ and } \text{X} (\overline{x}) \varphi \theta \leq s.
\]

\[
\text{Sol: We have } \vartheta = \{ \text{X} \mapsto \lambda \overline{x}. \text{Y}(\overline{x}) \}, \text{ where } \overline{y} \text{ is the subsequence of } \overline{x} \text{ consisting of the variables that appear freely in } t \text{ or } s. \text{ Let } \psi_1 = \{ \text{Y} \mapsto \lambda \overline{y}. t \} \text{ and } \psi_2 = \{ \text{Y} \mapsto \lambda \overline{y}. s \}. \text{ Then } \text{X} (\overline{x}) \psi_1 = t, \text{X} (\overline{x}) \psi_2 = s, \text{ and, hence, } \text{X} (\overline{x}) \varphi \theta \leq t \text{ and } \text{X} (\overline{x}) \varphi \theta \leq s.
\]

If \( \text{Mer} \) applies, then there exists \( \text{Y}(\overline{y}) : t' \neq s' \in \text{S}_1 \) such that \( \text{match(} \{ \overline{x} \}, \{ \overline{y} \}, t = t', s = s' \) is a permuting matcher \( \pi \), and \( \vartheta = \{ \text{Y} \mapsto \lambda \overline{y}. \text{X}(\overline{x}) \pi \} \). Then \( \text{X} (\overline{x}) \varphi \theta \leq t \) and \( \text{X} (\overline{x}) \varphi \theta \leq s \) obviously hold. As for the \( \text{Y}(\overline{y}) : t' \neq s' \), let \( \psi_1 = \{ \text{X} \mapsto \lambda \overline{x}. t \} \) and \( \psi_2 = \{ \text{X} \mapsto \lambda \overline{x}. s \}. \text{ Then } \text{Y}(\overline{y}) \psi_1 = (\lambda \overline{x}. t)(\overline{x}) \pi = t\pi = t', \text{Y}(\overline{y}) \psi_2 = (\lambda \overline{x}. s)(\overline{x}) \pi = s\pi = s', \text{ and, hence, } \text{Y}(\overline{y}) \varphi \theta \leq t' \text{ and } \text{Y}(\overline{y}) \varphi \theta \leq s' \).

Now, we proceed by induction on the length of derivation \( l \). In fact, we will prove a more general statement: If \( \text{A}_0; \text{S}_0; \vartheta_0 \Rightarrow \ast \emptyset; \text{S}_0; \vartheta_0 \varphi_1 \cdots \varphi_n \) is a derivation in \( \mathcal{P} \), then for any \( \text{X} (\overline{x}) : t \neq s \in \text{A}_0 \cup \text{S}_0 \) we have \( \text{X} (\overline{x}) \varphi_1 \cdots \varphi_n \varphi_1 \cdots \varphi_n \leq t \) and \( \text{X} (\overline{x}) \varphi_1 \cdots \varphi_n \varphi_1 \cdots \varphi_n \leq s \).
Hence, we have occurring freely in In the second case, either $s$ is an lgg of $X$. By the induction hypothesis, $Y_i(X)(\sigma)^i | t_i$ and $Y_i(X)(\sigma)^2 | s_i$ for all $1 \leq i \leq m$. By construction of $\sigma$, if there is $U \in \text{Vars}(\text{Ran}(\sigma^2))$, then there is an AUP of the form $U(t') : t' \neq s' \in S_n$. Let $\sigma$ (resp. $\varphi$) be a substitution which maps each such $U$ to the corresponding $t'$ (resp. $s'$). Then $Y_i(X)(\sigma)^2 \sigma = t_i$ and $Y_i(X)(\sigma)^2 \varphi = s_i$. Since $X(X)(\sigma)^1 = h(Y_i(X), \ldots, Y_m(X))$, we get that $X(X)(\sigma)^1 \sigma = t$ and $X(X)(\sigma)^1 \varphi = s$, and, hence, $X(X)(\sigma)^1 | t$ and $X(X)(\sigma)^1 | s$.

Abs: Here $t = \lambda y_1.t'$, $s = \lambda y_2.s'$, $X(X)(\sigma)^1 = \lambda y X'(X, y)$, and $A_i$ contains the AUP $X'(X, y) : t'y_1 \rightarrow y' \neq s'y_2 \rightarrow y$. By the induction hypothesis, $X'(X, y)(\sigma)^2 | t'y_1 \rightarrow y$ and $X'(X, y)(\sigma)^2 | s'y_2 \rightarrow y$. Since $X(X)(\sigma)^1 = \lambda y X'(X, y)(\sigma)^2$ and due to the way how $y$ was chosen, we finally get $X(X)(\sigma)^1 | t = t$ and $X(X)(\sigma)^1 | s = s$.

Sol: We have $X(X)(\sigma)^1 = Y(\overline{y})$ where $Y$ is in the store. By the induction hypothesis, $Y(\overline{y})(\sigma)^2 | t$ and $Y(\overline{y})(\sigma)^2 | s$. Therefore, $X(X)(\sigma)^1 | t$ and $X(X)(\sigma)^1 | s$.

For Mer, there exists $Y(\overline{y}) : t \neq s \in S_0$ such that $\text{match}(\{X\}, \{\overline{y}\}, t \Rightarrow t', s \Rightarrow s')$ is a permuting matcher $\pi$, and $\sigma^1 = \{Y \rightarrow X(X)(\pi)\}$. By the induction hypothesis, $X(X)(\sigma)^1 = X(X)(\sigma)^2 | t$ and $X(X)(\sigma)^1 = X(X)(\sigma)^2 | s$. These imply that $X(X)(\pi)(\sigma)^2 | t'$ and $X(X)(\pi)(\sigma)^2 | s'$, which, together, yields $X(X)(\sigma)^1 \pi | t'$ and $X(X)(\sigma)^1 \pi | s'$.

Hence, the result computed by $\Psi$ for $X : t \neq s$ generalizes both $t$ and $s$. We call $X$, a generalization of $t$ and $s$ computed by $\Psi$. Moreover, given a derivation $\{X : t \neq s\} : \mathcal{O} : \mathcal{O} \Rightarrow^{*} \mathcal{O} : \mathcal{S} : \sigma$, we say that $\sigma$ is a substitution computed by $\Psi$ for $X : t \neq s$;

the restriction of $\sigma$ on $X$, $\sigma|_X$, is an anti-unifier of $X : t \neq s$ computed by $\Psi$.

> **Theorem 4.3 (Completeness).** Let $\lambda X.t_1$ and $\lambda X.t_2$ be higher-order terms and $\lambda X.s$ be a higher-order pattern such that $\lambda X.s$ is a generalization of both $\lambda X.t_1$ and $\lambda X.t_2$. Then $\lambda X.s \leq \lambda X.X(\sigma)$, where $\sigma$ is an anti-unifier of $X : \lambda X.t_1 \Rightarrow \lambda X.t_2$ computed by $\Psi$.

**Proof.** By structural induction on $s$. We can assume without loss of generality that $\lambda X.s$ is an lgg of $\lambda X.t_1$ and $\lambda X.t_2$. We also assume that it is in the $\eta$-long $\beta$-normal form.

If $s$ is a variable, then there are two cases: Either $s \in X$, or $s \notin X$. In the first case, we have $s = t_1 = t_2$. The Dec rule gives $\sigma = \{X \rightarrow \lambda X.s\}$ and, hence, $\lambda X.s \leq \lambda X.X(\sigma) = s$.

In the second case, either $\text{Head}(t_1) \neq \text{Head}(t_2)$, or $\text{Head}(t_1) = \text{Head}(t_2) \notin X$. Sol is supposed to give us $\sigma = \{X \rightarrow \lambda X.X'(x')\}$ where $x'$ is a subsequence of $X$ consisting of variables occurring freely in $t_1$ or in $t_2$. But $x'$ should be empty, because otherwise $s$ would not be just a variable (remember that $\lambda X.s$ is an lgg of $\lambda X.t_1$ and $\lambda X.t_2$ in the $\eta$-long $\beta$-normal form).

Hence, we have $\sigma = \{X \rightarrow \lambda X.X'\}$ and $\lambda X.s \leq \lambda X.X(\sigma)$, because $s, s \Rightarrow X'$ is $X(\sigma)$.

If $s$ is a constant $c$, then $t_1 = t_2 = c$. We can apply the Dec rule, obtaining $\sigma = \{X \rightarrow \lambda X.c\}$ and, hence, $s = c \leq X(\sigma) = c$. Therefore, $\lambda X.s \leq \lambda X.X(\sigma)$.

If $s = x.s'$, then $t_1$ and $t_2$ must have the forms $t_1 = x.t'_1$ and $t_2 = y.t'_2$, and $s'$ must be an lgg of $t'_1$ and $t'_2$. Abs gives a new system $\{X'(X, x) : t'_1 \neq t'_2[x \rightarrow y]; \mathcal{O} : \sigma_1\}$, where $\sigma_1 = \{X \rightarrow \lambda X.x.X'(X, x)\}$. By the induction hypothesis, we can compute a substitution
σ₂ such that λΣ₂.x.x′ ≤ λΣ₂.x.X′(Σ₂,x)σ₂. Composing σ₁ and σ₂ into σ, we have X(Σ₂)σ = λΣ₂.X′(Σ₂,x)σ₂. Hence, we get λΣ₂.s = λΣ₁.λΣ₂.x.x′ ≤ λΣ₂.λΣ₂.x.X′(Σ₂,x)σ₂ = λΣ₂.X(Σ₂)σ.

Finally, assume that s is a compound term h(s₁,...,sₙ). If h \notin Σ₂ is a variable, then s₁,...,sₙ are distinct variables from Σ₂ (because a \Sigma₂.s is a higher-order pattern). That means that s₁,...,sₙ appear freely in t₁ or t₂. Moreover, either Head(t₁) = Head(t₂), or Head(t₁) = Head(t₂) = h. In both cases, we can apply the Sol rule to obtain σ = {X → λΣ₂.Y(s₁,...,sₙ)}. Obviously, λΣ₂.s ≤ λΣ₂.X(Σ₂)σ = λΣ₂.Y(s₁,...,sₙ).

If h ∈ Σ₂ or if it is a constant, then we should have Head(t₁) = Head(t₂). Assume they have the forms t₁ = h(t₁,1,...,tₙ₁) and t₂ = h(t₂,1,...,tₙ₂). We proceed by the Dec rule, obtaining {Y₁(Σ₂) : t₁ ≡ t₂ | 1 ≤ i ≤ n}; ∅; σ₀, where σ₀ = {X → λΣ₂.h(Y₁(Σ₂),...Yₙ(Σ₂))}. By the induction hypothesis, we can construct derivations Δ₁,...,Δₙ computing the substitutions σ₁,...,σₙ, respectively, such that λΣ₂.s₁ ≤ λΣ₂.Y₁(Σ₂)σ₁ for 1 ≤ i ≤ n. These derivations, together with the initial Dec step, can be combined into one derivation, of the form Δ = {X(Σ₂) : t₁ ≡ t₂}; ∅; σ₀ =⇒ {Y₁(Σ₂) : t₁ ≡ t₂ | 1 ≤ i ≤ n}; ∅; σ₀ =⇒ * ∅; Sₙ; σ₀σ₁,...,σₙ.

Let for any term t, t[p] denote the subterm of t at position p. If s does not contain duplicate variables free in Σ₂.s, then the construction of Δ and the fact that λΣ₂.s₁ ≤ λΣ₂.Y₁(Σ₂)σ₁ for 1 ≤ i ≤ n guarantee λΣ₂.s ≤ λΣ₂.X(Σ₂)σ₁,...,σₙ. If s contains duplicate variables free in Σ₂.s (e.g., of the form λΣ₂.z(Z(Σ₂)) and λΣ₂.z(Z(Σ₂)), where Σ₂ and Σ₂ have the same length) at positions p₁ and p₂, it indicates that

(a) t₁|p₁ and t₁|p₂ differ from each other by a permutation of variables bound in t₁,
(b) t₂|p₁ and t₂|p₂ differ from each other by the same (modulo variable renaming) permutation of variables bound in t₂,
(c) the path to p₁ is the same (modulo bound variable renaming) in t₁ and t₂. It equals (modulo bound variable renaming) the path to p₁ in s, and
(d) the path to p₂ is the same (modulo bound variable renaming) in t₁ and t₂. It equals (modulo bound variable renaming) the path to p₂ in s.

Then, because of (c) and (d), we should have two AUPs in Sₙ: One, between (renamed variants of) t₁|p₁ and t₂|p₁, and the other one between (renamed variants of) t₁|p₂ and t₂|p₂.

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(Σ₂₁) : r₁ ≡ r₂ and Z′(Σ₂₂) : r₁ ≡ r₂. The conditions (a) and (b) make sure that match([Z₁],[Z₂]), [r₁] = r₂} is a permuting matcher π, which means that we can apply the rule Mer with the substitution σ₂₁ = {Z₁ → λΣ₂₂.Z(Σ₂₂)}. We can repeat this process for all duplicated variables in s, extending Δ to the derivation Δ′ = {X(Σ₂) : t₁ ≡ t₂}; ∅; σ₀ =⇒ {Y₁(Σ₂) : t₁ ≡ t₂ | 1 ≤ i ≤ n}; ∅; σ₀ =⇒ * ∅; Sₙ; σ₀σ₁,...,σₙ, where σ₁,...,σₙ are substitutions introduced by the applications of the Mer rule. Let σ = σ₀σ₁,...,σₙσ₁′,...,σₙ′. By this construction, we have λΣ₂.s ≤ λΣ₂.X(Σ₂)σ, which finishes the proof.

Dependent which AUP is selected to perform a step, there can be different derivations in Ψ starting from the same AUP, leading to different generalizations. The next theorem states that all those generalizations are equivalent.

> **Theorem 4.4 (Uniqueness Modulo ≃).** Let \{X : t \neq s\}; ∅; ∅ =⇒ * ∅; S₁; σ₁ and \{X : t \neq s\}; ∅; ∅ =⇒ * ∅; S₂; σ₂ be two maximal derivations in Ψ from X : t \neq s. Then Xσ₁ ≃ Xσ₂.

**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 Δ₁ = A₁; S₁; σ₁ =⇒_{R₁} A₂; S₂; σ₁ν₁ =⇒_{R₂} A₃; S₃; σ₁ν₁ν₂ and Δ₂ = A₁; S₁; σ₁ =⇒_{R₂} A₃; S₃; σ₁ν₁ν₂ and Δ₂ = A₁; S₁; σ₁ =⇒_{R₂} A₃; S₃; σ₁ν₁ν₂.
\[ A_2' ; S_2'; \sigma_1 \vartheta_2 \rightarrow R_1' ; A_3' ; S_3' ; \sigma_1 \vartheta_2 \vartheta_1 \text{ are two two-step derivations, where } R_1 \text{ and } R_2 \text{ are (not necessarily different) rules and each of them transforms the same AUP(s) in both } A_1 \text{ and } \Delta_2, \text{ then } A_3 = A_1' ; S_1 = S_3', \text{ and } \sigma_1 \vartheta_2 \vartheta_1 = \sigma_1 \vartheta_2 \vartheta_1 \text{ (modulo the names of fresh variables).} \]

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 \( A; \{ Z(\underline{x}) : \text{true} \} \rightarrow t \) be the merging step with \( \pi \mapsto \text{match}(\{ \underline{x} \}, \{ y \} ) \}, \{ t_1 \mapsto t_2, s_1 \mapsto s_2 \}). \text{ If we do it in the other way around, we would get the step } A; \{ Z(\underline{x}) : \text{true} \} \rightarrow t_1 \text{ and } \{ Y(\underline{y}) : \text{true} \} \rightarrow s_1 \}

Our goal is to prove that \( X \vartheta_1 \approx X \vartheta_2 \). For this, we have to prove two inequalities: \( X \vartheta_1 \leq X \vartheta_2 \) and \( X \vartheta_2 \leq X \vartheta_1 \). To show \( X \vartheta_1 \leq X \vartheta_2 \), we first need to prove the equality:

\[
\lambda \underline{y}. Z(\underline{z}) \rho_2 = \lambda \underline{y}. Y(\underline{y}).
\]

Its left hand side is transformed as \( \lambda \underline{y}. Z(\underline{z}) \rho_2 = \lambda \underline{y}. Z(\underline{z}) \{ Z \rightarrow \lambda \underline{z}. Y(\underline{y}) \rho_1 \} = \lambda \underline{y}. \{ \lambda \underline{z}. Y(\underline{y}) \rho_1 \} \{ \underline{z} \rightarrow \underline{z} \} \). \text{ By (1)} \( X \rho \rightarrow \lambda \underline{y}. Z(\underline{z}) \rho_2, Z \rightarrow \lambda \underline{z}. Y(\underline{y}) \} = X \rho \rightarrow \lambda \underline{z}. Y(\underline{y}) \} \{ Z \rightarrow \lambda \underline{z}. Y(\underline{y}) \} \}

Hence, for given terms \( t \) and \( s \), the anti-unification algorithm \( \Psi \) 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:

\[ \text{Theorem 4.5 (Complexity of } \Psi \text{). The algorithm } \Psi \text{, when using } \mathfrak{M} \text{ to compute permuting matchers, has space complexity } O(n) \text{ and time complexity } O(n^2), \text{ where } n \text{ is the size (the number of symbols) of input.} \]

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

For proving the cubic time complexity, we can assume that the applications of the \text{Mer} rule are postponed till the end. The number of application of the other rules is bounded by the size of the input. \text{Abs} involves renaming which can be done in linear time. \text{Sol} requires selection of variables that occur freely in terms, which also needs linear time. Composi-

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 $\beta$-reduction (i.e., bound variable permutation). Traversal, replacement, and $\beta$-reduction can take at most quadratic time. Therefore, the complexity of this last step is also cubic. It implies that $\mathcal{P}$ 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 $\mathcal{P}$ and $\mathcal{M}$ rely), we can rename the variables before calling $\mathcal{P}$. 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 $\mathcal{P}$ 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 $\eta$-expansion: The AUP of the form $X(\overline{x}) : \lambda y.t \triangleq h(s_1, \ldots, s_m)$ should be transformed into $X(\overline{x}) : \lambda y.t \triangleq \lambda z.h(s_1, \ldots, s_m, z)$ for a fresh $z$. (Dually for abstractions in the right hand side.) The expansion should be performed both in $\mathcal{P}$ and $\mathcal{M}$. In addition, $\text{Sol}$ needs an extra condition for the case when $\text{Head}(t) = \text{Head}(s)$ 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 $F_{\tilde{\gamma}}$ [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