A Library of Anti-Unification Algorithms

Alexander Baumgartner and Temur Kutsia

RISC, Johannes Kepler University, Linz, Austria

Abstract. Generalization problems arise in many branches of artificial intelligence: machine learning, analogical and case-based reasoning, cognitive modeling, knowledge discovery, etc. Anti-unification is a technique used often to solve generalization problems. In this paper we describe an open-source library of some newly developed anti-unification algorithms in various theories: for first- and second-order unranked terms, higher-order patterns, and nominal terms.

1 Introduction

Given concrete examples, find an expression which adopts all their common features and has them as particular instances: This is an informal formulation of the generalization problem that arises in many branches of artificial intelligence. For instance, in inductive logic programming, which combines logic programming with machine learning, generalization is one of the steps used to fit the theory being learned to example clauses. In cognitive modeling, analogical reasoning relies on exploring and generalizing common features of different domains. Proof abstraction and lemma generation, software code clone detection and procedure invention are some other examples that involve generalization.

Anti-unification is a technique used often to solve generalization problems. Given two terms t_1 and t_2, this technique requires finding 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. Introduced in [21, 22] for the first-order syntactic case, anti-unification has been extended to more complex theories and is used in various applications. For some of those developments, one can see [2, 3, 4, 8, 9, 10, 11, 14, 15, 17, 18, 20, 23]. First-order order-sorted equational anti-unification (for combinations of associative and commutative theories with or without unit element) has been implemented in Maude and is freely available [1].

The open-source library described in this paper implements anti-unification for unranked terms, higher-order patterns, and nominal terms. Theories over these expressions have applications in knowledge representation, reasoning, programming, etc. Generalization problems in these theories may arise, for instance, in proof generalization or analogical reasoning in higher-order or nominal logic, in learning or refactoring λ-Prolog and α-Prolog programs, in detection of similarities in XML documents or in pieces of software code, just to name a few. Therefore, the algorithms provided by the library can be a valuable ingredient for tools that need to solve such generalization problems.

To be more specific, the library contains Java implementation of the following algorithms:

- first-order rigid unranked anti-unification from [16],

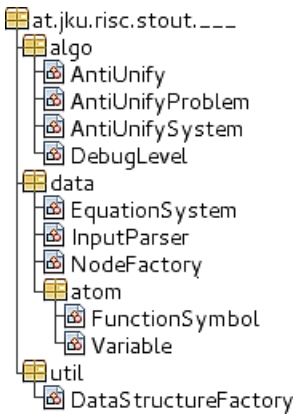
- second-order unranked anti-unification from [5],
- higher-order (pattern) anti-unification from [6] and
  - its subalgorithm for deciding α-equivalence,
- nominal anti-unification from [7] and
  - its subalgorithm for deciding equivariance.

The mentioned subalgorithms are needed to compute *least general* generalizations. All these algorithms can be accessed from the Web page of the SToUT project at RISC: `http://www.risc.jku.at/projects/stout/`. Each of them has a separate Web page with a convenient Web interface to try the algorithm online. There are also the link to the paper where the algorithm is described, a brief explanation of the syntax, and some examples. Besides using the Web interface, the user may try also a shell version of each algorithm, or download the sources, or embed the algorithm in her/his own project. A sample code of the latter option is also available from the Web.

In this paper, for each algorithm mentioned above we define the problem it solves, give some simple examples, indicate its Web address, and explain the Web interface. For some of them, we also explain how it can be embedded in users projects.

2 Structure of the Library

We describe the structure of the library in a bit more detail. It consists of four Java libraries for four anti-unification algorithms (urau.jar, urauc.jar, hoau.jar and nau.jar), which have the same structure. There is one main package which starts with the name `at.jku.risc.stout`, followed by a short abbreviation for the implemented algorithm (e.g. `urau`, `urauc`, `hoau` or `nau`). Under this main package there are three subpackages, namely `algo`, `data` and `util`. The `data` package has one subpackage of its own, which is called `data.atom`. The main package is irrelevant for using the library, as it only contains some test cases and the user interfaces. For instance, the applets which are used in the web frontend. Nevertheless, the source code might be interesting as those Java classes serve as reference implementations of the library.



As the name suggests, the package `algo` contains the algorithmic part of the library. There is a Java class named `AntiUnify` which serves as entry point of the respective anti-unification algorithm. The `data` package contains some Java classes which are needed to build the term structure. Furthermore, it includes the equation system which consists of some term pairs, and it offers a default implementation of an input parser, named `InputParser`. The Java class `EquationSystem` is implemented in a generic way, such that it can be used for different types of equation systems. In the `util` package there are some utility classes like `DataStructureFactory` which is used by the library to instantiate structures (e.g., lists, queues, maps, sets). The user of the library is free to choose an arbitrary implementation for all of those data structures, which might have some advantages on the performance of the provided algorithms. The package `data.atom` contains the atomic building blocks for constructing the terms.

3 Unranked First-Order Anti-Unification

The problem of unranked anti-unification is formulated for terms defined over unranked alphabet. Hedge variables are used to fill in gaps in generalizations, while term variables abstract single subterms with different top function symbols. Unranked anti-unification is finitary, but it turned out that a minimal and complete algorithm may compute up to 3^n generalizations, where n is the size of the input. To deal with this problem, the notion of $\mathcal{R}_\mathsf{T}$-*generalization* has been introduced in [16].

Definitions. Given pairwise disjoint countable sets of unranked function symbols $\mathcal{F}$ (symbols without fixed arity), term variables $\mathcal{V}_\mathsf{T}$, and hedge variables $\mathcal{V}_\mathsf{H}$, *terms* t and *hedges* $\tilde{s}$ are defined by the following grammar:

$$t ::= x \mid f(\tilde{s}) \quad s ::= t \mid X \quad \tilde{s} ::= s_1, \ldots, s_n \quad \text{where } x \in \mathcal{V}_\mathsf{T}, f \in \mathcal{F}, X \in \mathcal{V}_\mathsf{H}, n \geqslant 0.$$

Substitutions map term variables $(x, y, \ldots)$ to terms and hedge variables $(X, Y, \ldots)$ to hedges. For instance, $\{x \mapsto f(a), X \mapsto (g(y, b), c), Y \mapsto ()\}$ is a substitution, where $()$ is the empty hedge and a, b, c, f, g are unranked function symbols. Applying it to $f(x, X, Y)$ gives $f(f(a), g(y, b), c)$.

The set of positions (typically I, J) of a hedge $\tilde{s}$, denoted $pos(\tilde{s})$, is a prefix-closed set of strings of positive integers. For example, $pos(a, f(b, g(c)), d) = \{1, 2, 2{\cdot}1, 2{\cdot}2, 2{\cdot}2{\cdot}1, 3\}$. The symbol g stands at the position $2{\cdot}2$ and c occurs at the position $2{\cdot}2{\cdot}1$.

Two symbols $\mathsf{s}_1, \mathsf{s}_2 \in \mathcal{F} \cup \mathcal{V}_\mathsf{H} \cup \mathcal{V}_\mathsf{C}$ of a hedge are *horizontal consecutive* if their positions $I_{\mathsf{s}_1}{\cdot}i_{\mathsf{s}_1}$ and $I_{\mathsf{s}_2}{\cdot}i_{\mathsf{s}_2}$ are in the relation $I_{\mathsf{s}_1} = I_{\mathsf{s}_1}$ and $i_{\mathsf{s}_1} + 1 = i_{\mathsf{s}_2}$. They are in a *vertical chain* if their positions I_{s_1} and I_{s_2} are in the relation $I_{\mathsf{s}_1}{\cdot}1 = I_{\mathsf{s}_2}$ and $I_{\mathsf{s}_1}{\cdot}2 \notin pos(\tilde{s})$. For example, in $(a, f(g(a, b)))$, the occurrence of a at position 1 and f at 2 are horizontal consecutive. The occurrence of f at 2 and g at $2{\cdot}1$ are in vertical chain.

Given two hedges $\tilde{s}$ and $\tilde{q}$, an *alignment* is a sequence of the form $f_1\langle I_1, J_1\rangle \ldots f_m\langle I_m, J_m\rangle$ such that $I_1 < \cdots < I_m$, $J_1 < \cdots < J_m$, and f_k is the symbol at position I_k in $\tilde{s}$ and at position J_k in $\tilde{q}$ for all $1 \leqslant k \leqslant m$. With $<$ we denote the (strict) lexicographic ordering on positions, e.g., $1{\cdot}2{\cdot}1 < 1{\cdot}2{\cdot}2$ and $1{\cdot}2{\cdot}1 < 1{\cdot}2{\cdot}1{\cdot}2$.

A *rigidity function* $\mathcal{R}$ is a function that returns a set of alignments for two hedges with all the positions in the alignments being singleton integers (allowing only top symbols). Typical examples of rigidity functions are those which return longest common subsequences or longest common substrings of the top symbols of the input hedges.

Given two variable-disjoint hedges $\tilde{s}$, $\tilde{q}$ and the rigidity function $\mathcal{R}$, we say that a hedge $\tilde{g}$ that generalizes both $\tilde{s}$ and $\tilde{q}$ is their $\mathcal{R}_\mathsf{T}$-*generalization*, if either $\mathcal{R}(\tilde{s}, \tilde{q}) = \varnothing$ and $\tilde{g}$ is a hedge variable or a sequence of term variables, or there exists an alignment $f_1\langle i_1, j_1\rangle \cdots f_n\langle i_n, j_n\rangle \in \mathcal{R}(\tilde{s}, \tilde{q})$, such that:

1. If the sequence $\tilde{g}$ contains a pair of horizontal consecutive variables, then both of them are term variables.
2. If we remove all variables that occur as elements of $\tilde{g}$, we get a sequence of the form $f_1(\tilde{g}_1), \ldots, f_n(\tilde{g}_n)$.
3. For every $1 \leqslant k \leqslant n$, there exists a pair of sequences $\tilde{s}_k$ and $\tilde{q}_k$ such that $\tilde{s}|_{i_k} = f_k(\tilde{s}_k)$, $\tilde{q}|_{j_k} = f_k(\tilde{q}_k)$ and $\tilde{g}_k$ is an $\mathcal{R}_\mathsf{T}$-generalization of $\tilde{s}_k$ and $\tilde{q}_k$.

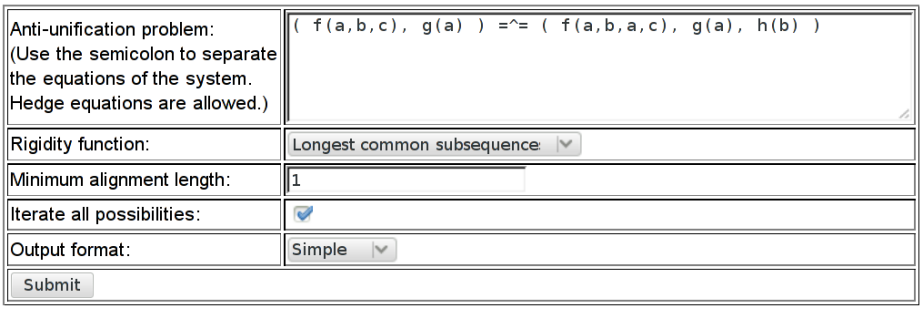
The implemented anti-unification algorithm solves the following problem:

Given: Two variable-disjoint hedges $\tilde{s}$ and $\tilde{q}$ and the rigidity function $\mathcal{R}$.
Find: A complete set of $\mathcal{R}_\mathsf{T}$-generalizations for $\tilde{s}, \tilde{q}$ and $\mathcal{R}$.

For instance, $\{(g(a,a), X, f(g(a), g(Y))),\ (X, g(x,x), f(g(a), g(Z)))\}$ is the minimal complete set of $\mathcal{R}_\mathsf{T}$-generalization of the hedges $(g(a,a), g(b,b), f(g(a), g(a)))$ and $(g(a,a), f(g(a), g))$, where $\mathcal{R}$ computes longest common subsequences.

Web page. The implementation of unranked rigid anti-unification is available from `http://www.risc.jku.at/projects/stout/software/urau.php`.

Web interface explanation. The input form of the web page of the first-order rigid unranked anti-unification algorithm consists of five rows:



In the first row, the anti-unification problem should be given. It consists of some anti-unification equations, separated by semicolons. Each anti-unification equation consists of two hedges, with =^= in between. The second row contains a drop-down menu to chose a rigidity function. Currently, the only two possibilities are longest common subsequence and longest common substring.

Furthermore, in the third row, one can specify the minimal alignment length l. We define $\mathcal{R}_l(\tilde{s}, \tilde{q}) := \{\mathfrak{a} : |\mathfrak{a}| \geqslant l, \mathfrak{a} \in \mathcal{R}(\tilde{s}, \tilde{q})\}$ as the rigidity function which corresponds to a given rigidity function $\mathcal{R}$ satisfying the length restriction. The implementation uses $\mathcal{R}_l$ and for any $\mathcal{R}$ holds $\mathcal{R}_0 = \mathcal{R}$. By unchecking the check-box from the fourth row, the user can specify to only compute the $\mathcal{R}_\mathsf{T}$-generalization for the first alignment which is returned by the rigidity function $\mathcal{R}_l$ (nondeterministically).

In the last row, the output format can be specified. One can choose form a drop-down box between simple, verbose and progress. The first choice only shows some basic facts and the computed $\mathcal{R}_\mathsf{T}$-generalizations. The verbose output format shows some additional information, like the differences at the input hedges. By choosing the progress output format, all the debug information will be shown to the user.

How to use. We assume that there are two data sources `in1` and `in2` available in form of `Reader` instances, each of them containing one of the hedges to be generalized. Moreover, the variable `eqSys` is of appropriate type and there is a Boolean variable `iterateAll` which corresponds to the option "Iterate all possibilities" of the web interface. We explain the usage of the library on a code fragment:

```
1  RigidityFnc rFnc=new RigidityFncSubsequence().setMinLen(3);
2  eqSys = new EquationSystem<AntiUnifyProblem>() {
```

```
3    public AntiUnifyProblem newEquation() {
4      return new AntiUnifyProblem();
5    } };
6  new InputParser<AntiUnifyProblem>(eqSys)
7      .parseHedgeEquation(in1, in2);
8  new AntiUnify(rFnc, eqSys, DebugLevel.SILENT) {
9    public void callback(AntiUnifySystem res, Variable var) {
10     System.out.println(res.getSigma().get(var));
11   }; }.antiUnify(iterateAll, null);
```

In the first line a certain rigidity function is instantiated and the minimum alignment size is set to the value 3. There are two rigidity functions available from the library. The one which is used in the code fragment computes longest common subsequence alignments. The other one is called `RigidityFncSubstring` and computes longest common substring alignments. It is easy to implement a different rigidity function. One simply has to extend the base class `RigidityFnc` which is provided by the library.

The lines 2 to 5 show the instantiation of an equation system which is of type `AntiUnifyProblem`. It is used in line 6 to instantiate a parser instance.

In line 7, the mentioned input sources are used to create one equation of two hedges, which is added to the equation system. One could add more equations to the system by just calling the method `parseHedgeEquation(in3, in4)` again.

After specifying the rigidity function and parsing the equation system, the main algorithm `AntiUnify` is invoked using this data (line 8). There is one additional argument, which specifies the debug level. For production use we want to silently compute all the generalizations and process them by a callback function, which is defined in the lines 9 to 11. For debugging, one must also specify a print stream at line 11 instead of `null`. The callback function is invoked for each generalization and it provides two arguments for the implementation. The first one is of type `AntiUnifySystem` and contains all the data which has been collected during the run: The substitution `getSigma`, the store `getStore` and some additional information. The second argument is the generalization variable. The computed generalization is the value which is associated with this variable in the substitution. Line 10 prints this generalization.

During the anti-unification process, fresh variables are introduced. They are named by a sequence number which is put between a prefix and a suffix. The counter for generating the number sequence is static and can be reset by calling the function `NodeFactory.resetCounter`. The prefix and the suffix for fresh term variables and also for fresh hedge variables can be specified by the user. Therefore the class `NodeFactory` offers four static variables, named `PREFIX_FreshTermVar`, `SUFFIX_FreshTermVar`, `PREFIX_FreshHedgeVar` and `SUFFIX_FreshHedgeVar`.

4 Unranked Second-Order Anti-Unification

The language used in section 3 does not permit higher-order variables. This imposes a natural restriction on solutions: The computed lggs do not reflect similarities between input hedges, which are located under distinct heads or at different depths. For instance, $f(a, b)$ and $g(h(a, b))$ are generalized by a single variable, although both terms contain a and b and a more natural generalization could be, e.g., $\mathring{X}(a, b)$, where $\mathring{X}$ is a

higher-order variable. In applications, it is often desirable to detect these similarities. Therefore, in [5], an anti-unification algorithm has been developed where second-order power is gained by using context variables to generalize vertical differences at the input hedges. Hedge variables are used to generalize horizontal differences.

Definitions. Given pairwise disjoint countable sets of unranked function symbols $\mathcal{F}$ (typically $a, b, c, f, g, \ldots$), hedge variables $\mathcal{V}_{\mathsf{H}}$ (typically $X, Y, \ldots$), unranked context variables $\mathcal{V}_{\mathsf{C}}$ (typically $\mathring{X}, \mathring{Y}, \ldots$), and a special symbol $\circ$ (the hole), *terms* t, *hedges* $\tilde{s}$, and *contexts* $\tilde{c}$ are defined by the following grammar:

$$t ::= X \mid f(\tilde{s}) \mid \mathring{X}(\tilde{s}) \qquad \tilde{s} ::= t_1, \ldots, t_n \qquad \tilde{c} ::= \tilde{s}_1, \circ, \tilde{s}_2 \mid \tilde{s}_1, f(\tilde{c}), \tilde{s}_2 \mid \tilde{s}_1, \mathring{X}(\tilde{c}), \tilde{s}_2$$

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

A context $\tilde{c}$ can apply to a hedge $\tilde{s}$, denoted by $\tilde{c}[\tilde{s}]$, obtaining a hedge by replacing the hole in $\tilde{c}$ with $\tilde{s}$. For example, $(\mathring{X}(X), f(f(\circ), b))[a, \mathring{X}(a)] = (\mathring{X}(X), f(f(a, \mathring{X}(a)), b))$. Application of a context to a context is defined similarly.

A substitution is a mapping from hedge variables to hedges and from context variables to contexts. When substituting a context variable $\mathring{X}$ by a context, the context will be applied to the argument hedge of $\mathring{X}$. The definition of positions and all the relations defined on positions, as well as the definition of an alignment are taken from section 3.

We only give an informal definition of *admissible alignments*. A necessary and sufficient condition for alignments to be admissible, as well as the exact definitions can be found in [5]. An alignment $\mathfrak{a}$ of two hedges $\tilde{s}$ and $\tilde{q}$ is called *admissible* iff there exists a generalization $\tilde{g}$ of $\tilde{s}$ and $\tilde{q}$ which contains all the corresponding symbols from $\mathfrak{a}$.

We call such a $\tilde{g}$ a *supporting generalization* of $\tilde{s}$ and $\tilde{q}$ with respect to $\mathfrak{a}$.

Least general supporting generalizations might not be unique. For instance, for (a, b, a) and (b, c) with the admissible alignment $b\langle 2, 1\rangle$, we have two supporting least general generalizations (X, b, X, Y) and (X, b, Y, X). Therefore, we are interested in a special class of supporting generalizations, which we call $\mathcal{R}_{\mathsf{C}}$-generalizations.

Given two variable-disjoint hedges $\tilde{s}, \tilde{q}$ and their admissible alignment $\mathfrak{a}$, a hedge $\tilde{g}$ is called an *$\mathcal{R}_{\mathsf{C}}$-generalization* of $\tilde{s}$ and $\tilde{q}$ with respect to $\mathfrak{a}$, if $\tilde{g}$ is a supporting generalization of $\tilde{s}$ and $\tilde{q}$ with respect to $\mathfrak{a}$ such that the following conditions are fulfilled:

1. There exist substitutions σ, ϑ with $\tilde{g}\sigma = \tilde{s}$ and $\tilde{g}\vartheta = \tilde{q}$ such that all the contexts in σ and ϑ are singleton contexts.
2. No context variable in $\tilde{g}$ applies to the empty hedge.
3. $\tilde{g}$ doesn't contain horizontal consecutive hedge variables.
4. $\tilde{g}$ doesn't contain vertical chains of variables.
5. $\tilde{g}$ doesn't contain context variables with a hedge variable as the first or the last argument (i.e., no subterms of the form $\mathring{X}(X, \ldots)$ and $\mathring{X}(\ldots, X)$).

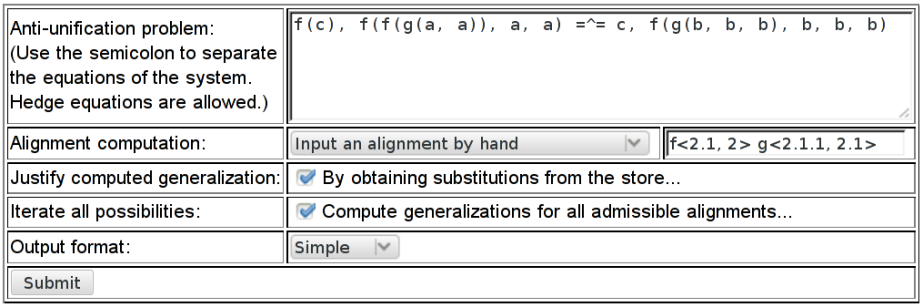
The implemented anti-unification algorithm has $O(n^2)$ time complexity and $O(n)$ space complexity, where n is the size of the input. It solves the following problem:

Given: Two variable-disjoint hedges $\tilde{s}$ and $\tilde{q}$ and their admissible alignment $\mathfrak{a}$.
Find: A least general $\mathcal{R}_{\mathsf{C}}$-generalization of $\tilde{s}$ and $\tilde{q}$ with respect to $\mathfrak{a}$.

For instance, $\mathring{X}(a, b)$ is an $\mathcal{R}_{\mathsf{C}}$-generalization of $f(g(a, b, c))$ and (a, b) with respect to $a\langle 1{\cdot}1{\cdot}1, 1\rangle b\langle 1{\cdot}1{\cdot}2, 2\rangle$, while $\mathring{X}(a, b, X)$ and $\mathring{X}(\mathring{Y}(a, b))$ are not.

Web page. The implementation of the algorithm is available from
`http://www.risc.jku.at/projects/stout/software/urauc.php`.

Web interface explanation. The input form of the web page of unranked second-order anti-unification consists of five rows, where the first, the fourth and the last row are equal to those of the unranked first-order anti-unification web interface.



In the second row, the alignment computation can be chosen. The only two possibilities are longest admissible alignments and the input of an alignment by hand. If the user selects the computation of longest admissible alignments, then the program automatically generates the set of all admissible alignments with maximum length, and the corresponding supporting generalizations are computed. Otherwise, the user has to specify an alignment in the input box next to the drop-down menu.

In the third row one can specify, whether or not to justify the computed $\mathcal{R}_C$-generalization. For justification of a generalization $\tilde{g}$, the recorded differences of the input hedges $\tilde{s}, \tilde{q}$ are used to obtain two substitutions σ, ϑ. Then the program tests whether $\tilde{g}\sigma = \tilde{s}$ and $\tilde{g}\vartheta = \tilde{q}$ holds. The justification fails if this is not the case.

How to use. The usage of this algorithm is very similar to the one we explained in section 3. Instead of a rigidity function there is an alignment computation function. The library offers two such functions: The first one, called `AlignFncLAA`, computes longest admissible alignments. The other one is `AlignFncInput` and can be used to specify a certain admissible alignment. The admissibility test for this alignment has to be done in advance. Therefore the `Alignment`-class offers a method `isAdmissible` which returns `true` iff an alignment is admissible. Alignment computation functions have the common base class `AlignFnc`. This base class can be used to implement other alignment computation functions.

5 Higher-Order Pattern Anti-Unification

The higher-order anti-unification algorithm described in [6] works on simply typed λ-terms: It takes as input two such terms of the same type, in η-long β-normal form, and returns their least general pattern generalization. Patterns here mean higher-order patterns à la Miller [19]. (Note that it is not required the input to be patterns.) Such a generalization always exists, is unique modulo α-equivalence and variable renaming, and can be computed in cubic time within linear space with respect to the size of the input, see [6].

Definitions. Simple types are constructed from *basic types* δ with the help of the type constructor $\rightarrow$ by the grammar $\tau := \delta \mid \tau \rightarrow \tau$. *Variables* and *constants* have an assigned type. Then λ*-terms* t are built using the grammar:

$$t ::= x \mid c \mid \lambda x.t \mid (t_1\ t_2) \quad \text{where } x \text{ is a typed variable and } c \text{ is a typed constant.}$$

Terms like $(\ldots(h\,t_1)\ldots t_m)$, where h is a constant or a variable, are written as $h(t_1, \ldots, t_m)$, and terms of the form $\lambda x_1.\cdots.\lambda x_n.t$ as $\lambda x_1, \ldots, x_n.t$. Substitutions map variables to terms of the same type, and can be extended to arbitrary terms as usual. A *higher-order pattern (HOP)* is a λ-term, in which, when written in η-long β-normal form, all free variables apply to pairwise distinct bound variables. For instance, if we use capital letters for free variables, $\lambda x.f(X(x), Y)$, $f(c, \lambda x.x)$ and $\lambda x, y.X(\lambda z.x(z), y)$ are patterns, while $\lambda x.f(X(X(x)), Y)$, $f(X(c), c)$ and $\lambda x, y.X(x, x)$ are not.

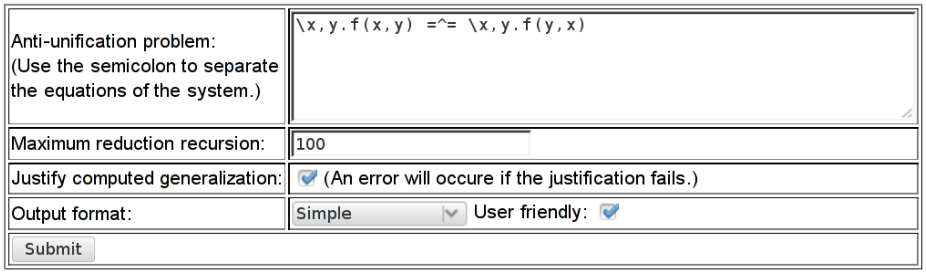
Given two variable-disjoint λ-terms t_1 and t_2, we say that a λ-term t that generalizes both t_1 and t_2 is their *higher-order pattern generalization*, if t is an HOP. The HOP anti-unification (HOPAU) algorithm solves the following problem:

Given: Higher-order terms t_1 and t_2 of the same type in η-long β-normal form.
Find: A least general higher-order pattern generalization of t_1 and t_2.

For instance, if $t_1 = \lambda x, y.f(h(x, x, y), h(x, y, y))$ and $t_2 = \lambda x, y.f(g(x, x, y), g(x, y, y))$, then $t = \lambda x, y.f(X(x, y), Y(x, y))$ is a higher-order pattern lgg of t_1 and t_2.

Web page. The implementation of the HOPAU algorithm is available from
`http://www.risc.jku.at/projects/stout/software/hoau.php`.

Web interface explanation. The implementation slightly differs from the theoretical algorithm: In addition to simply-typed terms, it can also take untyped input. It has an advantage that the user does not necessarily have to supply types, but has a disadvantage that the terms may not be typeable or normalizable. The input form of the Web interface to HOPAU algorithm consists of four rows shown below:



In the first row, the anti-unification problem should be given. The problems consist of one or more anti-unification equations, separated by semicolon. Each such equation consists of two λ-terms, with =^= in between. The backslash \ is used instead of λ.

In the second row, the maximum recursion depth of the β-reduction can be specified. This is to avoid infinite chain of reductions for terms like $(\lambda x.(x\ x))(\lambda x.(x\ x))$.

As in Sect. 4, one can choose to justify the computed lgg in the third row.

In the last row, the output format can be specified. One can choose form a drop-down box between `simple`, `verbose`, `progress`, and `progress-origin`. The

first three of them are like those described in Sect. 3. By choosing the output format `progress-origin`, all the debug information will be shown to the user, but the original names of bound variables are used. This is useful for debugging, as all the bound variables are renamed by the parser, giving them unique names.

5.1 Deciding α-equivalence

The HOPAU algorithm performs a constructive α-equivalence test to see whether different terms can be abstracted by the same variable. It is needed to ensure that the computed generalization is least general. Such a problem arises, e.g., in the course of generalization of the terms $t_1 = \lambda x, y, z.f(x(y, z), x(z, y))$ and $t_2 = \lambda x, y, z.f(X(y, \lambda u.u), X(z, \lambda v.v))$. To see if the same variable can be used in the generalization of the arguments of t_1 and t_2, we have to check whether there exists a bound variable renaming ρ such that $x(y, z)\rho = x(z, y)$ and $X(y, \lambda u.u)\rho \triangleq X(z, \lambda v.v)$.

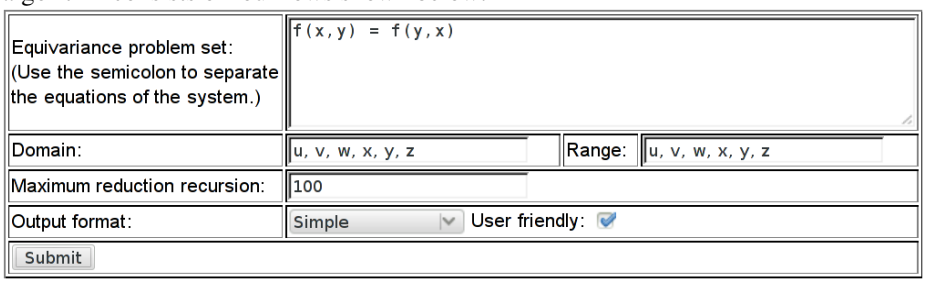
The algorithm that performs such a test is integrated in the HOPAU implementation, but we provide access to it separately as well, due to the fact that the problem is interesting, may appear in various contexts, and having a tool to solve it is useful. The algorithm solves the following problem (in linear time and space):

Given: A set of equations of the form $t \Rightarrow s$ where t and s are λ-terms, and two sets of variables, the domain D and the range R.
Find: A variable renaming substitution $\rho\colon D \to R$, such that $t\rho$ is α-equivalent to s for all equations $t \Rightarrow s$, if it exists. Otherwise report failure.

The generalization problem for t_1 and t_2 above creates the set of equations $\{x(y, z) \Rightarrow x(z, y), X(y, \lambda u.u) \Rightarrow X(z, \lambda v.v)\}$, the domain $D = \{x, y, z\}$ and the range $R = \{x, y, z\}$. Then the α-equivalence decision algorithm returns the renaming $\rho = \{x \mapsto x, y \mapsto z, z \mapsto y\}$. Afterwards, this renaming can be used to answer the original question of generalization of t_1 and t_2, obtaining the lgg $\lambda x, y, z.f(Y(x, y, z), Y(x, z, y))$ where, indeed, the variable Y appears twice.

Web page. The α-equivalence decision algorithm is available from `http://www.risc.jku.at/projects/stout/software/hoequiv.php`.

Web interface explanation. The input form of the Web interface to the α-equivalence algorithm consists of four rows shown below:



The first, the third and the fourth row are equivalent, respectively, to the first, the second and the fourth ones in the HOPAU interface, described above. (The terms of an

equivariance equation are separated by = instead of =^=.) In the second row, the two sets of variables which specify the domain and the range should be given.

How to use. We explain the usage on a code fragment and assume that there are two data sources `in1` and `in2` available in form of `Reader` instances, each of them contains one of the λ-terms. There is also an integer variable `maxReduce` which specifies the maximum recursion depth of β-reduction.

```
1 Set<Variable> ran = DataStructureFactory.$.newSet();
2 ran.add(new Variable("x", null));
3 ran.add(new Variable("y", null));
4 Map<Variable,Variable> permutation = new PermEquiv(eqSys,
      dom, ran).compute(DebugLevel.SILENT, null);
5 System.out.println(permutation);
```

The lines 1–3 show how range variables used in the mapping are specified. The second parameter of the `Variable`-constructor specifies the type of the variable. (`null` is used for untyped variables.) To obtain a new set, `DataStructureFactory.$` is used, which is a singleton instance of type `DataStructureFactory`. The user can change the behavior by simply assigning another implementation of this type to `$`. We assume that a set `dom` of domain variables is available and the set of equations `eqSys` exists (e.g., it can be created in a similar way as in unranked anti-unification above). In line 4, after specifying the domain and the range and parsing the equation system, the main algorithm `PermEquiv` is invoked using this data. It silently computes the renaming permutation, which is represented as a mapping from variables to variables.

6 Nominal Anti-Unification

Nominal techniques have been introduced in [12, 13] to formally represent and study systems with binding. The nominal anti-unification (NAU) algorithm developed in [7] takes as input two terms-in-contexts (pairs of a freshness constraint and a nominal term) and tries to compute a generalization term-in-context. Under the assumption that the set of atoms permitted in generalizations is finite, there is a unique lgg modulo variable renaming and α-equivalence. The algorithm has $O(n^4)$ time complexity and $O(n^2)$ space complexity, where n is the size of the input.

Definitions. Nominal terms contain *variables* and *atoms.* Variables can be instantiated and atoms can be bound. We have *sorts of atoms* ν and *sorts of data* δ as disjoint sets. *Atoms* $(a, b, \ldots)$ have one of the sorts of atoms. *Variables* $(X, Y, \ldots)$ have a sort of atom or data. Nominal function symbols $(f, g, \ldots)$ have an arity of the form $\tau_1 \times \cdots \times \tau_n \to \delta$, where δ is a sort of data and τ_i are sorts given by the grammar $\tau ::= \nu \mid \delta \mid \langle\nu\rangle\tau$. Abstractions have sorts of the form $\langle\nu\rangle\tau$. A *swapping* $(a\,b)$ is a pair of atoms of the same sort. A *permutation* π is a sequence of swappings. It can apply to terms and cause swapping the names of atoms. *Nominal terms* t are given by the grammar below, where $a.t$ is abstraction (it binds a) and $\pi{\cdot}X$ is called suspension:

$$t ::= f(t_1, \ldots, t_n) \mid a \mid a.t \mid \pi{\cdot}X$$

Suspensions suspend application of the permutation π to X until X is instantiated. Substitutions are defined in the standard way, and their application allows atom capture, for instance, $a.X\{X \mapsto a\} = a.a$.

A *freshness context* ∇ is a finite set of pairs of the form $a\#X$ stating that the instantiation of X cannot contain free occurrences of a. A *term-in-context* is a pair $\langle \nabla, t \rangle$ of a freshness context ∇ and a term t. A term-in-context $\langle \nabla, t \rangle$ is *based* on a set of atoms A, if all the atoms which occur in t and ∇ are elements of A. The NAU algorithm solves the following problem:

Given: Two nominal terms t_1 and t_2 of the same sort, a freshness context ∇, and a *finite* set of atoms A such that $\langle \nabla, t_1 \rangle$ and $\langle \nabla, t_2 \rangle$ are based on A.
Find: A term-in-context $\langle \Gamma, t \rangle$ which is also based on A, such that $\langle \Gamma, t \rangle$ is a least general generalization of $\langle \nabla, t_1 \rangle$ and $\langle \nabla, t_2 \rangle$.

For instance, for $t_1 = f(b, a)$, $t_2 = f(X, (a\,b)\cdot X)$, $\nabla = \{b\#X\}$, and $A = \{a, b\}$, the NAU algorithm computes the lgg of $\langle \nabla, t_1 \rangle$ and $\langle \nabla, t_2 \rangle$, which is $\langle \varnothing, f(Y, (a\,b)\cdot Y) \rangle$.

Web page. The nominal anti-unification algorithm is available from
`http://www.risc.jku.at/projects/stout/software/nau.php`.

Web interface explanation. The input form of the Web interface to the NAU algorithm consists of five rows shown below, where the first, the fourth and the fifth row are similar to the first, third and fifth explained in section 4.

Anti-unification problem: (Use the semicolon to separate the equations of the system.)	`f(a,b) =^= f(b,c)`
Freshness context:	e.g. a#X, b#Y
Extra atoms:	(Atoms from the problem...
Justify computed generalization:	(An error will occure if the justification fails.)
Output format:	Simple
Submit	

All the anti-unification equations share the same freshness context ∇, which can be specified in the second row. The computed term-in-context is a generalization of $\langle \nabla, t \rangle$ and $\langle \nabla, s \rangle$ for every anti-unification equation `t =^= s`.

As all the terms-in-context $\langle \nabla, t \rangle$ and $\langle \nabla, s \rangle$ obtained by anti-unification equations `t =^= s` have to be based on the same set of atoms A, all the atoms which appear in the anti-unification problem as well as those from ∇ are assumed to be elements of A. In the third row, the user may specify some additional atoms which are in A.

How to use. To explain the library usage on a code example, we again assume the existence of two `Reader` instances `in1` and `in2` which contain the nominal terms to be generalized. Furthermore, we assume that there is a `Reader` instance `inA` for reading atoms and `inN` for the freshness context. Both of them are assumed to be comma separated sets, e.g., `inN = {a#X,b#Y,...}` and `inA = {c,d,...}`, where the braces are optional. The data source `inA` only specifies extra atoms, which do nor appear in `in1`, `in2` and `inN`.

```
1  final NodeFactory factory = new NodeFactory();
2  eqSys = new EquationSystem<AntiUnifyProblem>() {
3    public AntiUnifyProblem newEquation(NominalTerm t,
         NominalTerm s) {
4      return new AntiUnifyProblem(t, s, factory);
5    } };
6  FreshnessCtx nablaIn = new InputParser(factory)
7      .parseEquationAndCtx(in1, in2, inA, inN, eqSys);
8  new AntiUnify(eqSys, nablaIn, DebugLevel.SILENT, factory) {
9    public void callback(AntiUnifySystem res, Variable var) {
10     System.out.println(res.getNablaGen());
11     System.out.println(res.getSigma().get(var));
12   }; }.antiUnify(false, null);
```

In contrast to the other libraries, an instance of `NodeFactory` is needed, which we create in line 1. The lines 2 to 5 demonstrate the creation of an equation system.

All the input sources are parsed in line 7. The new equation is added to `eqSys` and the parsed freshness context is returned. Moreover, the factory instance remembers all the parsed atoms regardless of the input source they come from. More equations may be added `eqSys` by calling the method `parseEquation(in1, in2, eqSys)` from `InputParser`. Atoms and freshness contexts can also be parsed separately.

Line 10 shows that, additionally to the substitution and store, the generated freshness context is provided by the instance `res` of the class `AntiUnifySystem`.

Again, one can specify how fresh variables and fresh atoms are named. In contrast to the other three libraries, this functionality is implemented by private instance variables of `NodeFactory` and appropriate getter and setter methods.

6.1 Deciding Equivariance

The nominal equivariance algorithm checks whether two terms differ from each other only by a permutation and bound atom renaming, i.e., if they are equivariant. Equivariance problem arises, for instance, in the course of generalization of the terms-in-contexts $p_1 = \langle \varnothing, f(a, b)\rangle$ and $p_2 = \langle \varnothing, f(b, c)\rangle$, where the atoms permitted in the generalization are a, b, and c, then the term-in-context $\langle \{c\#X, a\#Y\}, f(X, Y)\rangle$ generalizes p_1 and p_2, but it is not least general. To compute the latter, we need to reflect the fact that generalizations of the atoms are related to each other: One can be obtained from the other by the permutation $(b\,c)(c\,a)$. This leads to a least general generalization $\langle \{c\#X\}, f(X, (b\,c)(c\,a)\cdot X)\rangle$.

The equivariance decision algorithm solves the following problem (in quadratic time and space):

Given: A set of equations of the form $t \Rightarrow s$, a freshness context ∇, and a finite set of atoms A such that all $\langle \nabla, t\rangle$ and $\langle \nabla, s\rangle$ are based on A.

Find: A permutation π of variables from A such that for all equations $t \Rightarrow s$, $\pi\cdot t$ is α-equivalent to s with respect to ∇, if such a π exists. Otherwise report failure.

For instance, in the example above, the permutation $(b\,c)(c\,a)$ was computed by the equivariance algorithm for $\{a \Rightarrow b, b \Rightarrow c\}$, $A = \{a, b, c\}$, and $\nabla = \varnothing$.

Web page. The equivariance decision algorithm is available from `http://www.risc.jku.at/projects/stout/software/nequiv.php`.

Web interface explanation. The input form is nearly the same as the one for NAU:

Equivariance problem set: (Use the semicolon to separate the equations of the system.)	f(a,b) = f(b,c)
Freshness context:	e.g. a#X, b#Y
Justify computed permutation:	(An error will occure if the justification fails.)
Output format:	Simple
Submit	

There are two differences: The row to specify extra atoms is missing, because the computed permutation must only permute atoms which appear in the problem set and further on, terms of an equivariance equation are separated by = instead of =^=.

How to use. We assume to have data sources for two nominal terms `in1` and `in2`, and another one for a freshness context, called `inN`, similarly to the NAU algorithm. Moreover, we assume that an equation system `eqSys` has already been instantiated and that a `NodeFactory` instance, called `factory`, exists. We explain the usage of the library on the following code fragment:

```
1  InputParser parser = new InputParser(factory);
2  parser.parseEquation(in1, in2, eqSys);
3  FreshnessCtx nablaIn = parser.parseNabla(inN);
4  Collection<? extends Atom> atomSet = factory
5      .getAllByType(factory.classAtom);
6  Permutation pi = new Equivariance(eqSys, atomSet, nablaIn)
7      .compute(factory, false, DebugLevel.SILENT, null);
8  System.out.println(pi);
```

In line 1 the parser instance is created, which afterwards is used to parse the equation and the freshness context from the input sources. The lines 4 and 5 demonstrate how one can obtain the collected set of atoms from the `NodeFactory` instance.

Later in line 6 this set is needed to instantiate a class named `Equivariance`, which encapsulates the computation of a permutation `pi`. The computation returns `null`, if no permutation exists for the input. The class `Permutation` contains two mappings from atoms to atoms (`Map<Atom, Atom>`): The permutation itself can be obtained by calling `getPerm` and the inverse permutation, which can be obtained by `getInverse`. Furthermore the class `Permutation` provides some methods to work with permutations and swappings.

Acknowledgments. This research has been supported by the Austrian Science Fund (FWF) under the project SToUT (P 24087-N18).

Bibliography

[1] M. Alpuente, S. Escobar, J. Espert, and J. Meseguer. ACUOS: order-sorted modular ACU generalization. `http://safe-tools.dsic.upv.es/acuos/`, 2013.

[2] M. Alpuente, S. Escobar, J. Meseguer, and J. Espert. A modular order-sorted equational generalization algorithm. *Information and Computation*, 235:98–136, 2014.

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

[4] F. Baader. Unification, weak unification, upper bound, lower bound, and generalization problems. In R. V. Book, editor, *RTA*, volume 488 of *Lecture Notes in Computer Science*, pages 86–97. Springer, 1991.

[5] A. Baumgartner and T. Kutsia. Unranked second-order anti-unification. In U. Kohlenbach, editor, *Proc. 21st Workshop on Logic, Language, Information and Computation, WoLLIC'14*. Springer, 2014. To appear.

[6] A. Baumgartner, T. Kutsia, J. Levy, and M. Villaret. A variant of higher-order anti-unification. In F. van Raamsdonk, editor, *RTA*, volume 21 of *LIPIcs*, pages 113–127. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2013.

[7] A. Baumgartner, T. Kutsia, J. Levy, and M. Villaret. Nominal anti-unification. In T. Kutsia and C. Ringeissen, editors, *Proc. 28th International Workshop on Unification, UNIF'14*, number 14-06 in RISC Technical Report Series, 2014. To appear.

[8] P. E. Bulychev, E. V. Kostylev, and V. A. Zakharov. Anti-unification algorithms and their applications in program analysis. In A. Pnueli, I. Virbitskaite, and A. Voronkov, editors, *Ershov Memorial Conference*, volume 5947 of *Lecture Notes in Computer Science*, pages 413–423. Springer, 2009.

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

[10] T. de Souza Alcantara, J. Ferreira, and F. Maurer. Interactive prototyping of tabletop and surface applications. In P. Forbrig, P. Dewan, M. Harrison, and K. Luyten, editors, *EICS*, pages 229–238. ACM, 2013.

[11] A. L. Delcher and S. Kasif. Efficient parallel term matching and anti-unification. *J. Autom. Reasoning*, 9(3):391–406, 1992.

[12] M. Gabbay and A. M. Pitts. A new approach to abstract syntax with variable binding. *Formal Asp. Comput.*, 13(3-5):341–363, 2002.

[13] M. J. Gabbay. *A Theory of Inductive Definitions with alpha-Equivalence*. PhD thesis, University of Cambridge, UK, 2000.

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

[15] U. Krumnack, A. Schwering, H. Gust, and K.-U. Kühnberger. Restricted higher-order anti-unification for analogy making. In M. A. Orgun and J. Thornton, editors, *Australian Conference on Artificial Intelligence*, volume 4830 of *Lecture Notes in Computer Science*, pages 273–282. Springer, 2007.

[16] T. Kutsia, J. Levy, and M. Villaret. Anti-unification for unranked terms and hedges. *J. Autom. Reasoning*, 52(2):155–190, 2014.

[17] H. Li and S. J. Thompson. Similar code detection and elimination for Erlang programs. In M. Carro and R. Peña, editors, *PADL*, volume 5937 of *Lecture Notes in Computer Science*, pages 104–118. Springer, 2010.

[18] 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.

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

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

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

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

[23] U. Schmid. *Inductive Synthesis of Functional Programs, Universal Planning, Folding of Finite Programs, and Schema Abstraction by Analogical Reasoning*, volume 2654 of *Lecture Notes in Computer Science*. Springer, 2003.