Selected Topics from Unification Theory

Temur Kutsia

RISC, Johannes Kepler University of Linz, Austria
kutsia@risc.jku.at

April 7, 2015
Overview

The Anti-Unification Problem

Early Algorithms of Anti-Unification

Applications

Nominal Anti-Unification
The Anti-Unification Problem

Early Algorithms of Anti-Unification

Applications

Nominal Anti-Unification
The Anti-Unification Problem

**Given:** Two terms $t_1$ and $t_2$.

**Find:** Their generalization, a term $t$ such that both $t_1$ and $t_2$ are instances of $t$ under some substitutions.
The Anti-Unification Problem

Given: Two terms $t_1$ and $t_2$.
Find: Their generalization, a term $t$ such that both $t_1$ and $t_2$ are instances of $t$ under some substitutions.
The Anti-Unification Problem

**Given:** Two terms $t_1$ and $t_2$.

**Find:** Their generalization, a term $t$ such that both $t_1$ and $t_2$ are instances of $t$ under some substitutions.
The Anti-Unification Problem

**Given:** Two terms $t_1$ and $t_2$.

**Find:** Their generalization, a term $t$ such that both $t_1$ and $t_2$ are instances of $t$ under some substitutions.

$t_1 = t_{\sigma_1}$

$t_2 = t_{\sigma_2}$

$t = \sigma_1$ (generalization tree)

$t_{\sigma_1} = t_1$

$t_{\sigma_2} = t_2$
The Anti-Unification Problem

**Given:** Two terms $t_1$ and $t_2$.

**Find:** Their least general generalization $t$. 
The Anti-Unification Problem

**Given:** Two terms $t_1$ and $t_2$.

**Find:** Their *least general generalization* $t$. 

![Diagram showing the anti-unification process](image-url)
The Anti-Unification Problem

**Given:** Two terms $t_1$ and $t_2$.

**Find:** Their least general generalization $t$. 
Anti-Unification: Example

\[
\begin{align*}
  f(g(a, h(h(b))), h(b)) & & f(g(b, h(u)), u)
\end{align*}
\]
Anti-Unification: Example

\[
\begin{align*}
  f(g(a, h(h(b))), h(b)) & \quad f(g(b, h(u)), u)
\end{align*}
\]
Anti-Unification: Example

\[
\begin{aligned}
f(g(x, h(y)), y) &= f(g(a, h(h(b))), h(b)) \quad \text{and} \quad f(g(b, h(u)), u)
\end{aligned}
\]
Anti-Unification: Example

\[ f(g(x, h(y)), y) \]

\{ x \mapsto a, y \mapsto h(b) \} \quad \{ x \mapsto b, y \mapsto u \}

\[ f(g(a, h(h(b))), h(b)) \quad f(g(b, h(u)), u) \]
Anti-Unification and Unification

\[ t \sigma_1 = t_1 = t_2 = t \sigma_2 \]
Anti-Unification and Unification

\[ t_{\sigma_1} = t_1 = s = t_2 \vartheta = t_{\sigma_2} \]

\[ t_1 \vartheta = s = t_2 \vartheta \]
Anti-Unification and Weak Unification

\[ t_1 \sigma_1 = t_1 = s = t_2 \vartheta_2 \]

\[ t_2 = t_2 \sigma_2 \]
Outline

The Anti-Unification Problem

Early Algorithms of Anti-Unification

Applications

Nominal Anti-Unification
Anti-Unification: Origins

- Anti-unification was introduced in two papers:


Reynolds coined the term “anti-unification”.

Plotkin defined $C_1 \leq C_2$ for “a clause $C_1$ is more general than a clause $C_2$” iff there exists $\sigma$ such that $C_1\sigma \subseteq C_2$.

To justify this choice of notation, he writes:

*We have chosen to write $L_1 \leq L_2$ rather than $L_1 \geq L_2$ as Reynolds (1970) does, because in the case of clauses, ‘$\leq$’ is almost the same as ‘$\subseteq$’...*
Huet in 1976 formulated an algorithm in terms of recursive equations:

Let $\phi$ be a bijection from a pair of terms to variables.
Define a function $\lambda$, which maps pairs of terms to terms:

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

The Anti-Unification Problem

Early Algorithms of Anti-Unification

Applications

Nominal Anti-Unification
Anti-Unification: Applications

- The original motivation of introducing anti-unification was its application in automating induction.

- Since then, anti-unification has been used in reasoning by analogy, machine learning, inductive logic programming, software engineering, program synthesis, analysis, transformation, ...

- Algorithms suitable for those applications have been developed.
One of the interesting applications of anti-unification is in software code clone detection. Clones are similar pieces of software code. Obtained by reusing code fragments. Quite a typical practice.
Why Should Clones Be Detected?

In general, they are harmful:

- Additional maintenance effort.
- Additional work for enhancing and adapting.
- Inconsistencies presenting fault.
Why Should Clones Be Detected?

Extraction of similar code fragments may be required for

- program understanding
- code quality analysis
- plagiarism detection
- copyright infringement investigation
- software evolution analysis
- code compaction
- bug detection
Roy, Cordy and Koschke (2009) distinguish four types of clones:

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

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

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

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

1–3: Syntactic clones.
Examples of Syntactic Clone Types

if (a >= b) {
    c = d + b; // Comment1
    d = d + 1;
} else {
    c = d - a; // Comment2
}

Type 1: Identical code fragments except for variations in whitespace, layout, and comments.
Examples of Syntactic Clone Types

if (a >= b) {
    c = d + b; // Comment1
    d = d + 1;
} else
    c = d - a; // Comment2

else
    y = x + n; //Comment3

if (m >= n) {
    y = x + n;
    x = x + 5; //Comment3
} else
    y = x - m; //Comment2’

Type 2: Syntactically identical fragments except for variations in identifiers, types, whitespace, layout, and comments.
Examples of Syntactic Clone Types

if (a >= b) {
    c = d + b; // Comment1
    d = d + 1;
} else
    c = d - a; // Comment2

if (m >= n) {
    y = x + n;
} else
    y = x - m; //Comment2’

else z = 1; // Added statement
    x = x + 5; //Comment3

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


1. Preprocessing: Remove uninteresting code, determine source and comparison units/granularities.
2. Transformation: Obtain an intermediate representation of the preprocessed code.
3. Detection: Find similar source units in the transformed code.
4. Formatting: Clone locations of the transformed code are mapped back to the original code.
5. Filtering: Clone extraction, visualization, and manual analysis to filter out false positives.
Clone Detection and Anti-Unification

1. Tree-based approach.
2. Anti-unification is used in the detection step.
3. Anti-unification based tools:
   - Breakaway (Cottrel at al, 2007)
   - CloneDigger (Bulychev et al. 2009).
   - Wrangler (Li and Thompson, 2010).
   - HaRe (Brown and Thompson, 2010).
4. Achieve high precision.
5. Detect primarily clones of type 1 and 2.
Clones and Their Refactoring

```plaintext
if (a >= b) {
    c = d + b; // Comment1
    d = d + 1;
} else
    c = d - a; // Comment2

if (m >= n)
    { // Comment1'
        y = x + n;
        x = x + 5; //Comment3
    }
else
    y = x - m; //Comment2'

proc(x1,x2,x3,x4,x5) =
if (x1 >= x2) {
    x3 = x4 + x2;
    x4 = x4 + x5;
} else
    x3 = x4 - x1;

proc(a,b,c,d,1)                 proc(m,n,y,x,5)
```
Analogy Making and Anti-Unification

Example: Generalization of recursive program schemes from given structurally similar programs [Schmid, 2000].

Method: Restricted higher-order anti-unification.

Idea: Simple: abstract different heads of terms with a function variable if the arities coincide. Otherwise abstract with a term variable.

Example

Input:
- \( \text{fac}(x) = \text{if}(\text{eq0}(x), 1, * (x, \text{fac}(p(x)))) \)
- \( \text{sqr}(y) = \text{if}(\text{eq0}(y), 0, + (+ (y, p(y)), \text{sqr}(p(y)))) \)

Generalization
- \( \text{X}(z) = \text{if}(\text{eq0}(z), Y, Z(u, \text{X}(p(z)))) \)
Analogy Making and Anti-Unification

**Example:** Replay of program derivations [Hasker, 1995].

**Given:** Formal program specification together with a program fulfilling this specification, both connected by a derivation.

**Assume:** The specification has been slightly rewritten.

**Goal:** Instead of fully deriving a new program, alter the existing derivation and implementation along the changes of specification.

**Method:** Use higher-order anti-unification for combinator terms to detect changes and similarities between the old and the new specification, changes which can be propagated by adjusting the existing derivation.
Machine Learning and Anti-Unification

Example: An inductive learning method INDIE developed in [Armengol & Plaza, 2000].

Given: A training set of positive and negative examples, represented as feature terms.

Find: A description satisfied (subsumed) by all positive examples and no negative example.

Method: Feature term anti-unification (for positive examples).
Symbolic Computation and Anti-Unification

Example: Abstracting symbolic matrices [Almomen, Sexton, Sorge 2012]

Given: A concrete symbolic matrix.

Goal: Obtain a more compact representation employing ellipses in order to expose homogeneous regions present in the matrix.

Method: Use a version of first-order anti-unification with a special treatment of integer constants.
Symbolic Computation and Anti-Unification

\[
\begin{bmatrix}
  a_1 & b & b & b & b & b & d_{11} & d_{12} & d_{13} & d_{14} \\
  0 & a_2 & b & b & b & b & d_{21} & d_{22} & d_{23} & d_{24} \\
  0 & 0 & a_3 & b & b & b & d_{31} & d_{32} & d_{33} & d_{34} \\
  0 & 0 & 0 & a_4 & b & b & d_{41} & d_{42} & d_{43} & d_{44} \\
  0 & 0 & 0 & 0 & a_5 & b & c_4 & c_3 & c_2 & c_1 \\
  0 & 0 & 0 & 0 & 0 & a_6 & c_4 & c_3 & c_2 & c_1 
\end{bmatrix}
\]

\[
\begin{bmatrix}
  a_1 & b & \cdots & \cdots & b & d_{11} & \cdots & d_{1m} \\
  0 & \vdots & \ddots & \ddots & \vdots & \vdots & \ddots & \vdots \\
  \vdots & \ddots & \ddots & \ddots & \vdots & d_{m1} & \cdots & d_{mm} \\
  \vdots & \ddots & \ddots & b & c_m & \cdots & c_1 \\
  0 & \cdots & \cdots & 0 & a_n & c_m & \cdots & c_1 
\end{bmatrix}
\]
Example: Invariant computation [Bulychev, Kostylev, Zakharov 2010]

Given: A program represented as a set assignment statements (with input and output points labeled by natural numbers), and a program point labeled by $l$.

Find: Most specific invariant at point $l$. An invariant at $l$ is a (existentially closed equational) formula which holds for any run at point $l$.

Method: Based on anti-unification of substitutions. Compute an lgg of substitutions induced by sequences of variable assignments in runs.
Example: Modeling metaphoric expressions [Gust, Kühnberger, Schmid 2006]

Given: A metaphor as e.g., in “Electrons are the planets of the atom”.

Find: Its formal representation.

Method: Using heuristic-driven theory projection, which is based on anti-unification.
More . . .

More . . .

- Anti-unification in $\lambda_2$ [Lu et al. 2000] for reusing proofs about programs.
- Anti-unification for simple unranked hedges [Yamamoto et al. 2001] for inductive reasoning about hedge logic programs.
More ...

- Order-sorted feature term generalization [Aït-Kaci, Sasaki 1983]
- AC anti-unification [Pottier 1989].
- Anti-unification in commutative theories [Baader 1991].
- Variants of second order anti-unification [Hirata, Ogawa, Harao 2004].
- Constrained anti-unification [Page 1993].
- E-generalizations using regular tree grammars [Burghardt 2005].
More . . .

- Anti-unification for unranked terms
  [Kutsia, Levy, Villaret 2011].
- Pattern anti-unification for simply-typed $\lambda$-calculus
  [Baumgartner et al. 2013].
- Restricted second-order unranked anti-unification
  [Baumgartner, Kutsia 2014].
- Nominal anti-unification
  [Baumgartner et al. 2014].
- Anti-Unification Library:
  http://www.risc.jku.at/projects/stout/
Outline

The Anti-Unification Problem

Early Algorithms of Anti-Unification

Applications

Nominal Anti-Unification
Nominal Anti-Unification: Syntax

- Nominal terms contain variables, atoms, and function symbols.
- Variables can be instantiated and atoms can be bound.
- A swapping $(a \, b)$ is a pair of atoms.
- A permutation $\pi$ is a sequence of swappings.
- Nominal terms:

  $$t ::= f(t_1, \ldots, t_n) \mid a \mid a.t \mid \pi \cdot X$$
Nominal Anti-Unification: Syntax

- Nominal terms contain variables, atoms, and function symbols.
- Variables can be instantiated and atoms can be bound.
- A swapping \((a b)\) is a pair of atoms.
- A permutation \(\pi\) is a sequence of swappings.
- Nominal terms:
  \[
t ::= f(t_1, \ldots, t_n) \mid a \mid a.t \mid \pi \cdot X
  \]
- Permutations apply to terms and cause swapping the names of atoms (permutation action).
- \((c b)(a b) \bullet f(c, b.g(a, b), X) = f(b, a.g(c, a), (c b)(a b) \cdot X)\).
Freshness Constraints

- Freshness constraint: $a \# X$.
- The instantiation of $X$ cannot contain free occurrences of $a$.
- “$a$ is fresh for $X$”, “$a$ is forbidden in $X$”.
- Freshness context: a finite set of freshness constraints.
Substitutions

- Substitution: a mapping from variables to terms.
- Substitution application allows atom capture:
  \[ a.X \{ X \mapsto a \} = a.a. \]
Substitutions

- Substitution: a mapping from variables to terms.
- Substitution application allows atom capture:
  \[ a.X \{ X \mapsto a \} = a.a. \]
- A substitution \( \sigma \) respects a freshness context \( \nabla \) if for all \( X \), free atoms in \( X\sigma \) (except those in suspensions) are not forbidden in \( X \) by \( \nabla \).
Substitutions

- Substitution: a mapping from variables to terms.
- Substitution application allows atom capture:
  \[ a.X \{ X \mapsto a \} = a.a. \]
- A substitution \( \sigma \) respects a freshness context \( \nabla \) if for all \( X \), free atoms in \( X\sigma \) (except those in suspensions) are not forbidden in \( X \) by \( \nabla \).
- For instance, if \( \nabla = \{ a\#X, b\#X, b\#Y \} \), then
  - \( \{ X \mapsto f(c), Y \mapsto b.f(b) \} \) respects \( \nabla \).
  - \( \{ X \mapsto (a\cdot c)(b\cdot a) \cdot Y, Y \mapsto f(a) \} \) respects \( \nabla \).
  - \( \{ X \mapsto f(a) \} \) does not respect \( \nabla \).
The intended meanings of \( \approx \) (the \( \alpha \)-equivalence predicate) and \# (the freshness predicate):

1. \( \nabla \vdash t \approx u \) holds, if for every substitution \( \sigma \) such that \( t\sigma \) and \( u\sigma \) are ground terms and \( \sigma \) respects the freshness context \( \nabla \), \( t\sigma \) and \( u\sigma \) are \( \alpha \)-equivalent.

2. \( \nabla \vdash a \# t \) holds, if for every substitution \( \sigma \) such that \( t\sigma \) is a ground term and \( \sigma \) respects the freshness context \( \nabla \), the atom \( a \) is not free in \( t\sigma \).
Rules for $\approx$

$$\begin{align*}
\forall a \in \nabla : a \approx a & \quad (\approx\text{-atom}) \\
\nabla \vdash t \approx t' & \quad (\approx\text{-abs-1})
\end{align*}$$

$$\begin{align*}
a \neq a' & \quad \nabla \vdash t \approx (a a') \cdot t' & \quad \forall a \in \nabla : \pi \cdot a \neq \pi' \cdot a \\
\nabla \vdash a \# t' & \quad (\approx\text{-abs-2})
\end{align*}$$

$$\begin{align*}
\forall a \in \nabla : a \# X & \quad \nabla \vdash \pi \cdot X \approx \pi' \cdot X \\
\nabla \vdash t_1 \approx t' & \quad \cdots \quad \nabla \vdash t_n \approx t' \\
\nabla \vdash f(t_1, \ldots, t_n) \approx f(t'_1, \ldots, t'_n) & \quad (\approx\text{-application})
\end{align*}$$
Rules for \( \# \)

\[
\frac{a \neq a'}{\nabla \vdash a \# a'} \quad (\#\text{-atom})
\]

\[
\frac{(\pi^{-1} \cdot a \# X) \in \nabla}{\nabla \vdash a \# \pi \cdot X} \quad (\#\text{-susp.})
\]

\[
\frac{\nabla \vdash a \# t_1 \quad \ldots \quad \nabla \vdash a \# t_n}{\nabla \vdash a \# f(t_1, \ldots t_n)} \quad (\#\text{-application})
\]

\[
\frac{a \neq a'}{\nabla \vdash a \# a \cdot t} \quad (\#\text{-abst-1})
\]

\[
\frac{a \neq a'}{\nabla \vdash a \# a' \cdot t} \quad (\#\text{-abst-2})
\]
Using the rules for $\approx$ bottom-up, one can solve the following problem:

- Given: A set $\{a_1 t_1, \ldots, a_n t_n\}$.
- Compute: A $\subseteq$-minimal freshness context $\nabla$ such that $\nabla \vdash a_1 t_1, \ldots, \nabla \vdash a_n t_n$.

Call this algorithm FC.
Instance of a Freshness Context

- Using the rules for ≈ bottom-up, one can solve the following problem:
  - Given: A set \( \{a_1#t_1, \ldots, a_n#t_n\} \).
  - Compute: A \( \subseteq \)-minimal freshness context \( \nabla \) such that \( \nabla \vdash a_1#t_1, \ldots, \nabla \vdash a_n#t_n \).
- Call this algorithm FC.
- An instance of a freshness context \( \nabla \) under a substitution \( \sigma \):
  \[
  \nabla \sigma := FC(\{a#X\sigma \mid a#X \in \nabla\}).
  \]
Instance of a Freshness Context

- Using the rules for $\approx$ bottom-up, one can solve the following problem:
  - Given: A set $\{a_1\#t_1, \ldots, a_n\#t_n\}$.
  - Compute: A $\subseteq$-minimal freshness context $\nabla$ such that $\nabla \vdash a_1\#t_1, \ldots, \nabla \vdash a_n\#t_n$.

- Call this algorithm FC.

- An instance of a freshness context $\nabla$ under a substitution $\sigma$:

  $\nabla \sigma := FC(\{a\#X \sigma \mid a\#X \in \nabla\})$.

- Example:

  - $\nabla = \{a\#X, b\#X, b\#Y\}$.
  - $\sigma = \{X \mapsto (a \, c)(a \, b) \cdot Y, \ Y \mapsto f(a)\}$.
  - $\nabla \sigma = FC(\{a\#(a \, c)(a \, b) \cdot Y, \ b\#(a \, c)(a \, b) \cdot Y, \ b\#f(a)\}) = \{(a \, b)(a \, c) \bullet a\#Y, \ (a \, b)(a \, c) \bullet b\#Y\} = \{c\#Y, a\#Y\}$. 
Term-In-Context, Subsumption Order

- Term-in-context: a pair $\langle \nabla, t \rangle$ of a freshness context $\nabla$ and a term $t$.
- Subsumption order defined on terms-in-context:

$$\langle \nabla_1, t_1 \rangle \preceq \langle \nabla_2, t_2 \rangle$$

if there exists a substitution $\sigma$ such that
- $\sigma$ respects $\nabla_1$,
- $\nabla_1 \sigma \subseteq \nabla_2$, and
- $\nabla_2 \vdash t_1 \sigma \approx t_2$. 
Subsumption Order: Examples

- $\langle \emptyset, f(a) \rangle \preceq \langle \{a#X\}, f(a) \rangle$ (with $\sigma = \varepsilon$).
- $\langle \{a#X\}, f(a) \rangle \preceq \langle \emptyset, f(a) \rangle$ (with $\sigma = \{X \mapsto b\}$).
- $\langle \{a#X\}, f(X) \rangle \not\preceq \langle \emptyset, f(X) \rangle$.
- $\langle \emptyset, f(X) \rangle \preceq \langle \{a#Y\}, f(Y) \rangle$ with $\sigma = \{X \mapsto Y\}$.
Subsumption Order: Examples

- $\langle \{a\#X\}, f(X) \rangle \not\preceq \langle \{a\#X\}, f(a) \rangle$.
- $\langle \{b\#X\}, (a\ b)\cdot X \rangle \preceq \langle \{c\#X\}, (a\ c)\cdot X \rangle$ (with $\sigma = \{X \mapsto (a\ b)(a\ c)\cdot X\}$).
- $\langle \{c\#X\}, (a\ c)\cdot X \rangle \preceq \langle \{b\#X\}, (a\ b)\cdot X \rangle$ (with $\sigma = \{X \mapsto (a\ c)(a\ b)\cdot X\}$).
Generalization

- \( \langle \Gamma, r \rangle \) is called a generalization of \( \langle \nabla_1, t \rangle \) and \( \langle \nabla_2, s \rangle \) if \( \langle \Gamma, r \rangle \preceq \langle \nabla_1, t \rangle \) and \( \langle \Gamma, r \rangle \preceq \langle \nabla_2, s \rangle \).
Generalization

- \( \langle \Gamma, r \rangle \) is called a generalization of \( \langle \nabla_1, t \rangle \) and \( \langle \nabla_2, s \rangle \) if \( \langle \Gamma, r \rangle \preceq \langle \nabla_1, t \rangle \) and \( \langle \Gamma, r \rangle \preceq \langle \nabla_2, s \rangle \).

- \( \langle \Gamma, r \rangle \) least general generalization (lgg) of \( \langle \nabla_1, t \rangle \) and \( \langle \nabla_2, s \rangle \) if there is no generalization \( \langle \Gamma', r' \rangle \) of \( \langle \nabla_1, t \rangle \) and \( \langle \nabla_2, s \rangle \) which satisfies \( \langle \Gamma, r \rangle \prec \langle \Gamma', r' \rangle \).
Generalization

- $\langle \Gamma, r \rangle$ is called a generalization of $\langle \nabla_1, t \rangle$ and $\langle \nabla_2, s \rangle$ if $\langle \Gamma, r \rangle \preceq \langle \nabla_1, t \rangle$ and $\langle \Gamma, r \rangle \preceq \langle \nabla_2, s \rangle$.

- $\langle \Gamma, r \rangle$ least general generalization (lgg) of $\langle \nabla_1, t \rangle$ and $\langle \nabla_2, s \rangle$ if there is no generalization $\langle \Gamma', r' \rangle$ of $\langle \nabla_1, t \rangle$ and $\langle \nabla_2, s \rangle$ which satisfies $\langle \Gamma, r \rangle \prec \langle \Gamma', r' \rangle$.

- Similar to unification, we can also talk about a minimal complete set of generalizations.
Nominal anti-unification is of type 0.

Example

- Given terms-in-contexts: \( \langle \emptyset, a_1 \rangle \) and \( \langle \emptyset, a_2 \rangle \).
- In any complete set of generalizations of them there is an infinite chain

\[
\langle \emptyset, X \rangle \prec \langle \{a_3 \# X\}, X \rangle \prec \langle \{a_3 \# X, a_4 \# X\}, X \rangle \prec \cdots,
\]

where \( \{a_1, a_2, a_3, \ldots\} \) is the set of all atoms of the language.
- Hence, \( \langle \emptyset, a_1 \rangle \) and \( \langle \emptyset, a_2 \rangle \) do not have a minimal complete set of generalizations.
Bad News

Nominal anti-unification is of type 0.

Example

▷ Given terms-in-contexts: \( \langle \emptyset, a_1 \rangle \) and \( \langle \emptyset, a_2 \rangle \).

▷ In any complete set of generalizations of them there is an infinite chain

\[
\langle \emptyset, X \rangle \prec \langle \{a_3 \# X\}, X \rangle \prec \langle \{a_3 \# X, a_4 \# X\}, X \rangle \prec \cdots ,
\]

where \( \{a_1, a_2, a_3, \ldots \} \) is the set of all atoms of the language.

▷ Hence, \( \langle \emptyset, a_1 \rangle \) and \( \langle \emptyset, a_2 \rangle \) do not have a minimal complete set of generalizations.

A way out: restrict the set of atoms permitted in generalizations.
A term-in-context $\langle \nabla, t \rangle$ is based on a set of atoms $A$, if all the atoms in $t$ and $\nabla$ are elements of $A$.

For instance, $\langle \{b\#X\}, f(c.g(c), (a b) \cdot X) \rangle$ is based on $\{a, b, c\}$ and on $\{a, b, c, d\}$, but not on $\{a, b, d\}$.
Nominal Anti-Unification Problem

Given: Two nominal terms $t_1$ and $t_2$, a freshness context $\nabla$, 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 $\langle \nabla, t_2 \rangle$. 
Solving The Nominal Anti-Unification Problem

Notation:
- Anti-unification triple (AUT): $X : t \triangleleft s$.
- $X$: generalization variable.
Solving The Nominal Anti-Unification Problem

The rule-based NAU algorithm works on tuples $P; S; \Gamma; \sigma$ and two global parameters $A$ and $\nabla$, where

- $P$ and $S$ are sets of AUTs.
- If $X : t \triangleq s \in P \cup S$, then $X$ is unique in $P \cup S$.
- $P$: AUTs to be solved (the problem).
- $S$: already solved AUTs (the store).
- $A$ is a finite set of atoms.
- $\nabla$ does not constrain generalization variables.
- $\Gamma$: freshness context (computed so far) which constrains generalization variables.
- $\sigma$: substitution (computed so far) mapping generalization variables to nominal terms.
- $P, S, \nabla, \Gamma$: $A$-based.
Example

To anti-unify two terms-in-context ⟨{b#Y}, f(b, a)⟩ and ⟨{b#Y}, f(Y, (a b)·Y)⟩, based on the set of atoms \{a, b\}, we need to create the initial system \( P; S; \Gamma; \sigma \) where

\[
\begin{align*}
P &= \{X : f(b, a) \triangleq f(Y, (a b)\cdot Y)\}, \\
S &= \emptyset, \\
\Gamma &= \emptyset, \\
\sigma &= \varepsilon,
\end{align*}
\]

and initialize the global parameters \( A \) and \( \nabla \) as

\[
\begin{align*}
A &= \{a, b\}, \\
\nabla &= \{b#Y\}.
\end{align*}
\]
Rules of The Nominal Anti-Unification Algorithm

**Decomposition:**

\[
\{ X : h(t_1, \ldots, t_m) \triangleq h(s_1, \ldots, s_m) \} \cup P; S; \Gamma; \sigma \Rightarrow \\
\{ Y_1 : t_1 \triangleq s_1, \ldots, Y_m : t_m \triangleq s_m \} \cup P; S; \Gamma; \\
\sigma\{ X \mapsto h(Y_1, \ldots, Y_m) \}
\]

where \( h \) is a function symbol or an atom, \( Y_1, \ldots, Y_m \) are fresh, and \( m \geq 0 \).
Abstraction:

\[
\{X : a.t \triangleq b.s\} \cup P; \ S; \ \Gamma; \ \sigma \Rightarrow
\{Y : (c \ a) \bullet t \triangleq (c \ b) \bullet s\} \cup P; \ S; \ \Gamma; \ \sigma\{X \mapsto c.Y\},
\]

where \(Y\) is fresh, \(c \in A\), \(\nabla \vdash c\#a.t\) and \(\nabla \vdash c\#b.s\).
Rules of The Nominal Anti-Unification Algorithm

Solving:

\[
\{X : t \triangleq s}\cup P; S; \Gamma; \sigma \implies
\]

\[
P; S\cup\{X : t \triangleq s\}; \Gamma\cup\Gamma'; \sigma,
\]

if none of the previous rules is applicable. The set $\Gamma'$ is defined as

$\Gamma' := \{a\#X \mid a \in A \land \nabla \vdash a\#t \land \nabla \vdash a\#s\}$. 
Merging:

\[ P; \{ X : t_1 \triangleq s_1, Y : t_2 \triangleq s_2 \} \cup S; \Gamma; \sigma \Rightarrow \]

\[ P; \{ X : t_1 \triangleq s_1 \} \cup S; \Gamma\{ Y \mapsto \pi \cdot X \}; \sigma\{ Y \mapsto \pi \cdot X \}, \]

where \( \pi \) is an \( Atoms(t_1, s_1, t_2, s_2) \)-based permutation such that \( \nabla \vdash \pi \cdot t_1 \approx t_2 \), and \( \nabla \vdash \pi \cdot s_1 \approx s_2 \).
Let $t = f(a, b)$, $s = f(b, c)$, $\nabla = \emptyset$, and $A = \{a, b, c, d\}$.

The NAU algorithm run:

\[
\{X : f(a, b) \triangleq f(b, c)\}; \emptyset; \emptyset; \varepsilon \implies\text{Dec}
\]
\[
\{Y : a \triangleq b, Z : b \triangleq c\}; \emptyset; \emptyset; \{X \mapsto f(Y, Z)\} \implies\text{Sol}
\]
\[
\{Z : b \triangleq c\}; \{Y : a \triangleq b\}; \{c\#Y, d\#Y\}; \{X \mapsto f(Y, Z)\} \implies\text{Sol}
\]
\[
\emptyset; \{Y : a \triangleq b, Z : b \triangleq c\}; \{c\#Y, d\#Y, a\#Z, d\#Z\}; \{X \mapsto f(Y, Z)\} \implies\text{Mer}
\]
\[
\emptyset; \{Y : a \triangleq b\}; \{c\#Y, d\#Y\}; \{X \mapsto f(Y, (a \cdot b)(b \cdot c) \cdot Y)\}
\]
Let $t = f(b, a)$, $s = f(Y, (a b) \cdot Y)$, $\nabla = \{b \# Y\}$, and $A = \{a, b\}$.

The NAU algorithm computes $p = \langle \emptyset, f(Z, (a b) \cdot Z) \rangle$.

It generalizes the input pairs:

$p\{Z \mapsto b\} = \langle \emptyset, f(b, a) \rangle \preceq \langle \nabla, t \rangle$,

$p\{Z \mapsto Y\} = \langle \emptyset, f(Y, (a b) \cdot Y) \rangle \preceq \langle \nabla, s \rangle$. 
Let \( t = f(a.b, X) \), \( s = f(b.a, Y) \), \( \nabla = \{ c\#X \} \), \( A = \{ a, b, c, d \} \).

The NAU algorithm computes \( p = \langle \{ c\#Z_1, d\#Z_1 \}, f(c.Z_1, Z_2) \rangle \).

It generalizes the input pairs:

\[
\begin{align*}
p\{Z_1 \mapsto b, Z_2 \mapsto X\} &= \langle \emptyset, f(c.b, X) \rangle \preceq \langle \nabla, t \rangle, \\
p\{Z_1 \mapsto a, Z_2 \mapsto Y\} &= \langle \emptyset, f(c.a, Y) \rangle \preceq \langle \nabla, s \rangle.
\end{align*}
\]
Informally:

- **Soundness**: The result computed by the algorithm is indeed an \( A \)-based generalization of the input terms-in-context.

- **Completeness**: For any \( A \)-based generalization of the input terms-in-context, the algorithm can compute one which is at most as general that the given generalization.

- **Uniqueness**: All generalizations computed by the algorithm via different AUT choices are the same modulo variable renaming and \( \alpha \)-equivalence.
The Result Depends on the Choice of the Set of Atoms

Example
Let $A_1 = \{a, b\}$ and $A_2 = \{a, b, c\}$.

- $A_1$-based lgg of $\langle \emptyset, a.b \rangle$ and $\langle \emptyset, b.a \rangle$: $\langle \emptyset, X \rangle$.
- $A_2$-based lgg of $\langle \emptyset, a.b \rangle$ and $\langle \emptyset, b.a \rangle$: $\langle \{c\#X\}, c.X \rangle$. 
Given $t$, $s$ and $\nabla$, how to choose a set of atoms $A$ so that

(a) $t$, $s$, $\nabla$ are $A$-based and

(b) in the $A$-based lgg $\langle \Gamma, r \rangle$ of $\langle \nabla, t \rangle$ and $\langle \nabla, s \rangle$, the term $r$ generalizes $s$ and $t$ in the “best way”, maximally preserving similarities and uniformly abstracting differences between $s$ and $t$. 
Given $t$, $s$ and $\nabla$, how to choose a set of atoms $A$ so that

(a) $t$, $s$, $\nabla$ are $A$-based and
(b) in the $A$-based lgg $\langle \Gamma, r \rangle$ of $\langle \nabla, t \rangle$ and $\langle \nabla, s \rangle$, the term $r$ generalizes $s$ and $t$ in the “best way”, maximally preserving similarities and uniformly abstracting differences between $s$ and $t$.

Answer: Besides all the atoms occurring in $t$, $s$, or $\nabla$, $A$ should contain at least $m$ more atoms, where $m = \min\{\|t\|_{\text{Abs}}, \|s\|_{\text{Abs}}\}$. $\|t\|_{\text{Abs}}$ stands for the number of abstraction occurrences in $t$. 
Deciding Equivariance

Recall **Merging**:

\[
P; \{ X : t_1 \triangleq s_1, Y : t_2 \triangleq s_2 \} \cup S; \Gamma; \sigma \implies \]

\[
P; \{ X : t_1 \triangleq s_1 \} \cup S; \Gamma \{ Y \mapsto \pi \cdot X \}; \sigma \{ Y \mapsto \pi \cdot X \},
\]

where \( \pi \) is an \( Atoms(t_1, s_1, t_2, s_2) \)-based permutation such that \( \nabla \vdash \pi \bullet t_1 \approx t_2 \), and \( \nabla \vdash \pi \bullet s_1 \approx s_2 \).

The condition requires an algorithm that constructively decides whether \( t \) and \( s \) are equivariant with respect to \( \nabla \):

**Given**: \( t, s, \) and \( \nabla \).

**Find**: An \( Atoms(t, s) \)-based permutation \( \pi \) such that \( \nabla \vdash \pi \bullet t \approx s \).
Equivariance Decision Algorithm

A rule-based algorithm $\mathcal{E}$, working on tuples $E; \nabla; A; \pi$.

- $E$: A set of equivariance equations of the form $t \approx s$.
- $\nabla$: A freshness context.
- $A$: A finite set of atoms which are available for computing $\pi$.
- $\pi$ holds the permutation to be returned in case of success.
Equvariance Decision Algorithm

- First phase: simplification. Function applications, abstractions and suspensions are decomposed as long as possible.
- Second phase: permutation computation.
Equivalence Decision Algorithm: First Phase Rules

**Decomposition in Equivariance**
\[
\{ f(t_1, \ldots, t_m) \approx f(s_1, \ldots, s_m) \} \cup E; \nabla; A; \text{Id} \implies \\
\{ t_1 \approx s_1, \ldots, t_m \approx s_m \} \cup E; \nabla; A; \text{Id}.
\]

**Alpha Equivalence**
\[
\{ a.t \approx b.s \} \cup E; \nabla; A; \text{Id} \implies \\
\{ (a \cdot c) \cdot t \approx (b \cdot c) \cdot s \} \cup E; \nabla; A; \text{Id},
\]
where \(c\) is a fresh atom of the same sort as \(a\) and \(b\).

**Suspension**
\[
\{ \pi_1 \cdot X \approx \pi_2 \cdot X \} \cup E; \nabla; A; \text{Id} \implies \\
\{ \pi_1 \cdot a \approx \pi_2 \cdot a \mid a \in A \land a \neq X \not\in \nabla \} \cup E; \nabla; A; \text{Id}.
\]
Remove
\[
\{a \approx b\} \cup E; \nabla; A; \pi \mapsto E; \nabla; A \setminus \{b\}; \pi,
\]
if \(\pi \cdot a = b\).

Solve in Equivariance
\[
\{a \approx b\} \cup E; \nabla; A; \pi \mapsto E; \nabla; A \setminus \{b\}; (\pi \cdot a \ b)\pi
\]
if \(\pi \cdot a \in A, b \in A, \text{ and } \pi \cdot a \neq b\).
Equivariance Decision Algorithm: Example

Equivariance problem $E = \{ a \approx a, \ (a\ b)(c\ d) \cdot X \approx b\cdot X \}$ and $\nabla = \{ a\#X \}$:

\[
\begin{align*}
\{ a \approx a, \ (a\ b)(c\ d) \cdot X & \approx b\cdot X \}; \{ a\#X \}; \\
\{ a, b, c, d \}; Id & \implies \text{Alpha Equivalence}
\end{align*}
\]
\[
\begin{align*}
\{ a \approx a, \ (a\ é)(a\ b)(c\ d) \cdot X & \approx (b\ é)\cdot X \}; \{ a\#X \}; \\
\{ a, b, c, d \}; Id & \implies \text{Suspension}
\end{align*}
\]
\[
\begin{align*}
\{ a \approx a, \ é \approx é, \ d \approx c, \ c \approx d \}; \{ a\#X \}; \\
\{ a, b, c, d \}; Id & \implies \text{Remove}
\end{align*}
\]
\[
\begin{align*}
\{ é \approx é, \ d \approx c, \ c \approx d \}; \{ a\#X \}; \{ b, c, d \}; Id & \implies \text{Remove}
\end{align*}
\]
\[
\begin{align*}
\{ d \approx c, \ c \approx d \}; \{ a\#X \}; \{ b, c, d \}; Id & \implies \text{Solve in Equivariance}
\end{align*}
\]
\[
\begin{align*}
\{ c \approx d \}; \{ a\#X \}; \{ b, d \}; (d\ c) & \implies \text{Remove}
\end{align*}
\]
\[
\emptyset; \{ a\#X \}; \{ b \}; (d\ c).
\]
Properties of the Equivariance Algorithm

Informally:

- **Soundness**: The computed permutation shows that the input terms are equivariant.
- **Completeness**: For any permutation that shows that the input terms are equivariant, the algorithm computes one that is equal to the given permutation (on the set of free atoms of the corresponding term).
Given a set of equivariance equations $E$ and a freshness context $\nabla$, the equivariance algorithm has $O(n^2 + m)$ time complexity, where $m$ be the size of $\nabla$, and $n$ is the size of $E$. The nominal anti-unification algorithm has $O(n^5)$ time complexity and $O(n^4)$ space complexity, where $n$ is the input size.
Given a set of equivariance equations $E$ and a freshness context $\nabla$, the equivariance algorithm has $O(n^2 + m)$ time complexity, where $m$ be the size of $\nabla$, and $n$ is the size of $E$.

The nominal anti-unification algorithm has $O(n^5)$ time complexity and $O(n^4)$ space complexity, where $n$ is the input size.
Implementation

- Implemented in Java, open source.
- Both algorithms are accessible online and can be downloaded.
- The equivariance algorithm is also available separately.