Anti-Unification

Temur Kutsia

November 3, 2014

What is Anti-Unification

Early Algorithms

Applications

Anti-Unification Library

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

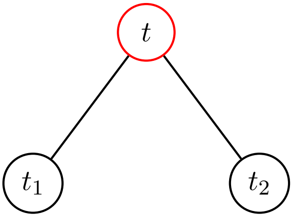
Given: Two terms t_1 and t_2.

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.

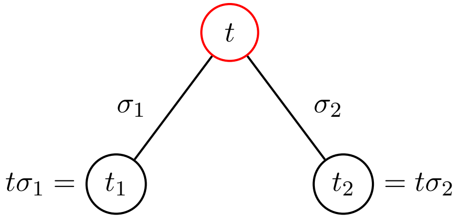
Find: Their generalization, a term t such that both t_1 and t_2 are instances of t under some substitutions.



The Anti-Unification Problem

Given: Two terms t_1 and t_2.

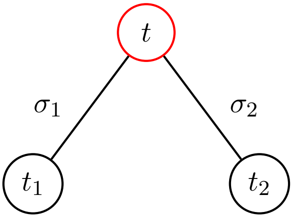
Find: Their generalization, a term t such that both t_1 and t_2 are instances of t under some substitutions.



The Anti-Unification Problem

Given: Two terms t_1 and t_2.

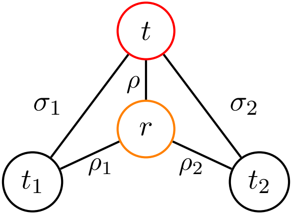
Find: Their least general generalization t.



The Anti-Unification Problem

Given: Two terms t_1 and t_2.

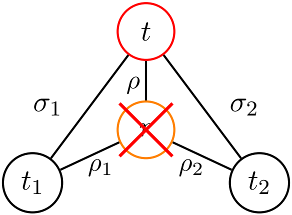
Find: Their least general generalization t.



The Anti-Unification Problem

Given: Two terms t_1 and t_2.

Find: Their least general generalization t.



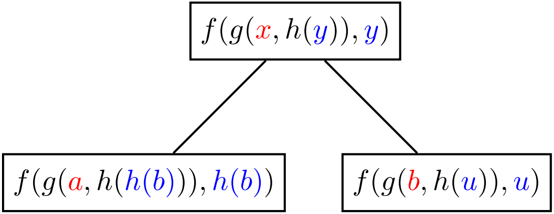
Anti-Unification: Example

$$\boxed{f(g(a, h(h(b))), h(b))} \qquad \boxed{f(g(b, h(u)), u)}$$

Anti-Unification: Example

$f(g(a, h(h(b))), h(b))$ $\quad$ $f(g(b, h(u)), u)$

Anti-Unification: Example



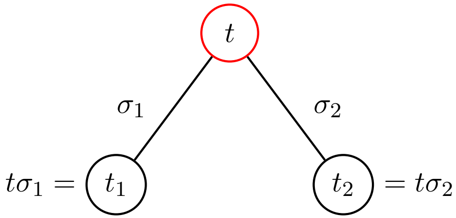
Anti-Unification: Example

$$f(g(x, h(y)), y)$$

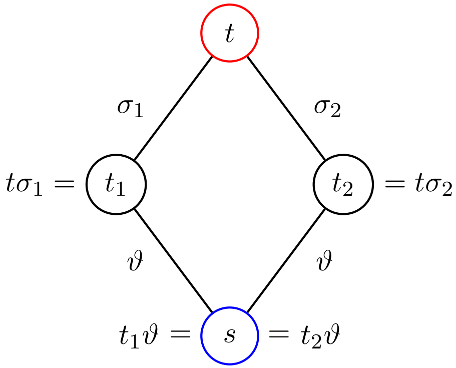
$\{x \mapsto a, y \mapsto h(b)\}$ $\qquad$ $\{x \mapsto b, y \mapsto u\}$

$$f(g(a, h(h(b))), h(b)) \qquad f(g(b, h(u)), u)$$

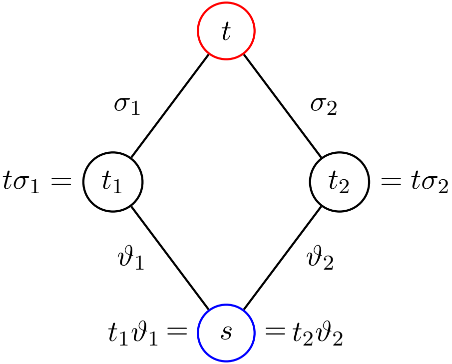
Anti-Unification and Unification



Anti-Unification and Unification



Anti-Unification and Weak Unification



What is Anti-Unification

Early Algorithms

Applications

Anti-Unification Library

Anti-Unification: Origins

- Anti-unification was introduced in two papers:

  Plotkin, G.D.: A note on inductive generalization. Mach. Intell. 5(1), 153–163 (1970)

  Reynolds, J.C.: Transformational systems and the algebraic structure of atomic formulas. Mach. Intell. 5(1), 135–151 (1970)

Anti-Unification: Origins

Plotkin's algorithm:

Let W_1, W_2 be any two compatible words. The following algorithm terminates at stage *3*, and the assertion made there is then correct.

1. Set V_i to $W_i (i=1,2)$. Set ε_i to $\varepsilon (i=1,2)$. ε is the empty substitution.

2. Try to find terms t_1, t_2 which have the same place in V_1, V_2 respectively and such that $t_1 \neq t_2$ and either t_1 and t_2 begin with different function letters or else at least one of them is a variable.

3. If there are no such t_1, t_2 then halt. V_1 is a least generalization of $\{W_1, W_2\}$ and $V_1 = V_2$, $V_i \varepsilon_i = W_i (i=1,2)$.

4. Choose a variable x distinct from any in V_1 or V_2 and wherever t_1 and t_2 occur in the same place in V_1 and V_2, replace each by x.

5. Change ε_i to $\{t_i | x\} \varepsilon_i (i=1,2)$.

6. Go to *2*.

Anti-Unification: Origins

Reynolds' algorithm:

(a) Set the variables $\bar{A}$ to A, $\bar{B}$ to B, ζ and η to the empty substitution, and i to zero.

(b) If $\bar{A}=\bar{B}$, exit with $A \sqcup B=\bar{A}=\bar{B}$.

(c) Let k be the index of the first symbol position at which $\bar{A}$ and $\bar{B}$ differ, and let S and T be the terms which occur, beginning in the kth position, in $\bar{A}$ and $\bar{B}$ respectively.

(d) If, for some j such that $1 \leqslant j \leqslant i$, $Z_j\zeta=S$ and $Z_j\eta=T$, then alter $\bar{A}$ by replacing the occurrence of S beginning in the kth position by Z_j, alter $\bar{B}$ by replacing the occurrence of T beginning in the kth position by Z_j, and go to step (b).

(e) Otherwise, increase i by one, alter $\bar{A}$ by replacing the occurrence of S beginning in the kth position by Z_i, alter $\bar{B}$ by replacing the occurrence of T beginning in the kth position by Z_i, replace ζ by $\zeta \cup \{S/Z_i\}$, replace η by $\eta \cup \{T/Z_i\}$, and go to step (b).

Anti-Unification: Origins

- 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 σ 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 ‘$\subseteq$’...

Anti-Unification: Origins

- Huet in 1976 formulated an algorithm in terms of recursive equations:

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

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

What is Anti-Unification

Early Algorithms

Applications

Anti-Unification Library

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.

Software Code Clone Detection with Anti-Unification

- 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

Classification

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
```

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

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
```

```
if (m >= n)
   { // Comment1'
    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)
   { // Comment1'
    y = x + n;
    z = 1; // Added statement
    x = x + 5; //Comment3
   }
else
   y = x - m; //Comment2'
```

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

From Roy, Cordy, and Koschke (2009):

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.

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).

Anti-unification of Feature Terms

Example

Input:

$$P_1 : person \begin{bmatrix} name \doteq N_1 : name \begin{bmatrix} first \doteq John \\ last \doteq Smith \end{bmatrix} \\ lives\text{-}at \doteq A_1 : address\,[city \doteq NYCity] \\ father \doteq X_1 : person\,[name \doteq Smith] \end{bmatrix}$$

$$P_2 : person \begin{bmatrix} name \doteq N_2 : name\,[last \doteq Taylor] \\ wife \doteq Y_2 : person\,[name \doteq M_2 : name\,[first \doteq Mary]] \\ father \doteq X_2 : person\,[name \doteq Taylor] \end{bmatrix}$$

Anti-unification of Feature Terms

Example

Input:

$$P_1 : person \left[\begin{array}{l} name \doteq N_1 : name \left[\begin{array}{l} first \doteq John \\ last \doteq Smith \end{array}\right] \\ lives\text{-}at \doteq A_1 : address\,[city \doteq NYCity] \\ father \doteq X_1 : person\,[name \doteq Smith] \end{array}\right]$$

$$P_2 : person \left[\begin{array}{l} name \doteq N_2 : name\,[last \doteq Taylor] \\ wife \doteq Y_2 : person\,[name \doteq M_2 : name\,[first \doteq Mary]] \\ father \doteq X_2 : person\,[name \doteq Taylor] \end{array}\right]$$

Output:

$$P_3 : person \left[\begin{array}{l} name \doteq N_3 : name\,[last \doteq family\text{-}name] \\ father \doteq X_3 : person\,[name \doteq family\text{-}name] \end{array}\right]$$

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:

```
fac(x) = if(eq0(x), 1, *(x,            fac(p(x)))
sqr(y) = if(eq0(y), 0, +(+(y,p(y)), sqr(p(y)))
```

Generalization

```
  X(z) = if(eq0(z), Y, Z(u,              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.

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.

Program Analysis and Anti-Unification

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.

Linguistics and Anti-Unification

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 . . .

- Relative lgg [Plotkin 1971] taking into account background knowledge.
- Anti-unification in the Calculus of Constructions [Pfenning 1991] aiming at proof generalizations.
- Anti-unification for relaxed patterns [Feng and Muggleton 1992] for inductive logic programming.
- Generalization under implication (special forms) [Idestam-Almquist 1995, Nienhuys-Cheng & de Wolf 1996] for inductive logic programming.

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.
- Second-order generalization [Chiba, Aoto, Toyama 2008] for automatic construction of program transformatione templates.
- Variations of restricted higher-order anti-unification [Bobere & Besold 2012] in analogy-making.
- Anti-unification for relational rules [de Souza Alcantara et al. 2012] for learning custom gestures.

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].
- Word anti-unification [Biere 1993, Ciceckli & Ciceckli 2006].
- Constrained anti-unification [Page 1993].
- E-generalizations using regular tree grammars [Burghardt 2005].
- Equational and order-sorted anti-unification [Alpuente et al, 2008, 2009, 2013].

More ...

- Anti-unification for unranked terms [Kutsia, Levy, Villaret 2011].
- Pattern anti-unification for simply-typed λ-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/`

Contains Java implementation of the following algorithms:

- first-order rigid unranked anti-unification,
- second-order unranked anti-unification,
- higher-order (pattern) anti-unification and
  - its subalgorithm for deciding α-equivalence,
- nominal anti-unification and
  - its subalgorithm for deciding equivariance.

First-order Rigid Unranked Anti-Unification

- Given two sequences $f_1(\tilde{s}_1), \ldots, f_n(\tilde{s}_n)$ and $g_1(\tilde{r}_1), \ldots, g_m(\tilde{r}_m)$.
- Take a common subsequence of $f_1, \ldots, f_n$ and $g_1, \ldots, g_m$.
- Let it be $h_1, \ldots, h_k$.
- Then a rigid generalization of the given sequences has a form

$$X_1, h_1(\tilde{q}_1), X_2, h_2(\tilde{q}_2), \ldots, X_{k-1}, h_k(\tilde{q}_k), X_k,$$

  where
  - X's are (not necessarily distinct) new sequence variables,
  - Some X's can be omitted,
  - if $h_i = f_j = g_l$, then $\tilde{q}_i$ is a rigid generalization of $\tilde{s}_j$ and $\tilde{r}_l$.
- The algorithm is parametrized by a rigidity function. It decides which common subsequences are taken.

Second-Order Unranked Anti-Unification

- In first-order rigid anti-unification, the computed lggs do not reflect similarities that are located under distinct heads or at different depths.
- First order lgg of $f(a, b)$ and $g(h(a, b))$ is just a variable, despite the fact that the terms share a and b.

Second-Order Unranked Anti-Unification

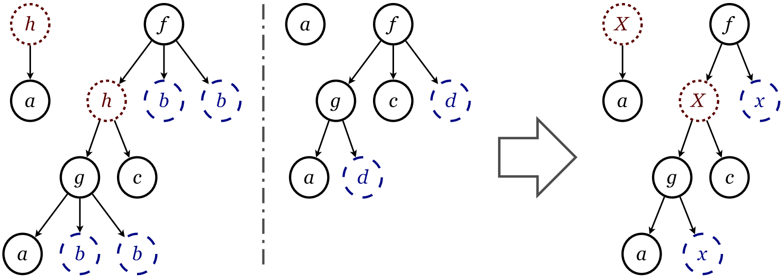
- In first-order rigid anti-unification, the computed lggs do not reflect similarities that are located under distinct heads or at different depths.
- First order lgg of $f(a,b)$ and $g(h(a,b))$ is just a variable, despite the fact that the terms share a and b.
- Second order Unranked Anti-Unification addresses this problem.
- For $f(a,b)$ and $g(h(a,b))$, it will return $X(a,b)$, where X is a higher-order (context) variable.

Second-Order Unranked Anti-Unification

The idea:

- Take the input term sequences and first construct a "skeleton" of a their generalization.
- The "skeleton" corresponds to a sequence embedded into each of the input sequence.
- Next, insert context and/or hedge variables into the skeleton, to uniformly generalize (vertical and horizontal) differences between the input sequences.
- The skeleton computation function is the parameter of the algorithm.

Second-Order Unranked Anti-Unification



Anti-Unification for Simply-Typed Lambda Terms

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.

Anti-Unification for Simply-Typed Lambda Terms

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.

Higher-order pattern (HOP):

- a λ-term, in which, when written in η-long β-normal form, all free variables apply to pairwise distinct bound variables.
- Patterns: $\lambda x.f(X(x), Y)$, $f(c, \lambda x.x)$, $\lambda x, y.X(\lambda z.x(z), y)$.
- Non-patterns: $\lambda x.f(X(X(x)), Y)$, $f(X(c), c)$, $\lambda x, y.X(x, x)$.

Deciding α-Equivalence

- Higher-order pattern anti-unification requires to decide α-equivalence constructively.
- The corresponding algorithm: Given two terms, if they are α-equivalent, the algorithm returns the justifying renaming of bound variables. Otherwise, it fails.

Nominal Anti-Unification

- 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 π 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

- Permutation can apply to terms and cause swapping the names of atoms.
- $(c\,b)(a\,b) \cdot f(c,\, b.g(a,b),\, X) = f(b,\, a.g(c,a),\, (c\,b)(a\,b) \cdot X)$.

Nominal Anti-Unification

- Permutation can apply to terms and cause swapping the names of atoms.
- $(c\,b)(a\,b) \cdot f(c,\, b.g(a,b),\, X) = f(b,\, a.g(c,a),\, (c\,b)(a\,b) \cdot X)$.
- Freshness constraint: $a\#X$
- The instantiation of X cannot contain free occurrences of a.

Nominal Anti-Unification

- Permutation can apply to terms and cause swapping the names of atoms.
- $(c\,b)(a\,b) \cdot f(c,\, b.g(a,b),\, X) = f(b,\, a.g(c,a),\, (c\,b)(a\,b) \cdot X)$.
- Freshness constraint: $a \# X$
- The instantiation of X cannot contain free occurrences of a.
- Freshness context: a finite set of freshness constraints.

Nominal Anti-Unification

- Term-in-context: a pair $\langle \nabla, t \rangle$ of a freshness context ∇ and a term t.

Nominal Anti-Unification

- Term-in-context: 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 in t and ∇ are elements of A.

Nominal Anti-Unification

- Term-in-context: 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 in t and ∇ are elements of A.
- For instance, $\langle \{b \# X\}, f(X, (a\, b) \cdot X) \rangle$ is based on $\{a, b\}$ and on $\{a, b, c\}$, but not on $\{a, c\}$.

Nominal Anti-Unification

- Term-in-context: 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 in t and ∇ are elements of A.
- For instance, $\langle \{b \# X\}, f(X, (a\, b) \cdot X) \rangle$ is based on $\{a, b\}$ and on $\{a, b, c\}$, but not on $\{a, c\}$.
- There is a subsumption order defined on terms-in-context.

Nominal Anti-Unification Problem

Given: Two nominal terms t_1 and t_2, 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 $\langle \nabla, t_2 \rangle$.