Unranked Anti-Unification and Its Application in Software Code Clone Detection

Temur Kutsia

RISC, Johannes Kepler University Linz, Austria

Joint work with Jordi Levy and Mateu Villaret

What Are Code Clones

- Similar pieces of software code.
- Obtained by reusing code fragments.
- Quite 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 in the tasks of
    - program understanding
    - code quality analysis
    - aspect mining
    - 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, literals, 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, literals, 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, literals, 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, literals, 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 Techniques

From Roy, Cordy, and Koschke (2009):

1. Text-based: Little transformation, mostly raw source code in the detection process.
2. Token-based: Transforming the source code into a sequence of lexical “tokens”. The sequence is then scanned for duplicated subsequences of tokens.
3. Tree-based: Transforming the source code into trees (parse trees, ASTs, . . .) which can then be processed using either tree comparison or structural metrics to find clones.
4. Metrics-based: Gathering a number of metrics for code fragments and then compare metrics vectors rather than code or trees directly.
5. Graph-based: Find isomorphic subgraphs in PDGs.
6. Hybrid approaches.

Our Idea

1. Aiming at high precision for clones of type 3.
2. Tree-based approach (possibly combined with text- and metrics-based)
3. In the clone detection step, use anti-unification.

Our Idea

1. Aiming at high precision for clones of type 3.
2. Tree-based approach (possibly combined with text- and metrics-based)
3. In the clone detection step, use anti-unification.
4. Existing anti-unification based tools:
    - CloneDigger (Bulychev et al. 2009).
    - Wrangler (Li and Thompson, 2010).
    - HaRe (Brown and Thompson, 2010).

Our Idea

1. Aiming at high precision for clones of type 3.
2. Tree-based approach (possibly combined with text- and metrics-based)
3. In the clone detection step, use anti-unification.
4. Existing anti-unification based tools:
   - CloneDigger (Bulychev et al. 2009).
   - Wrangler (Li and Thompson, 2010).
   - HaRe (Brown and Thompson, 2010).
5. We propose using unranked anti-unification instead of the standard one.

Unranked Terms and Hedges

Unranked alphabet: The arity of function symbols is not fixed.

Variables: Term variables $x, y, z, \ldots$ and hedge variables $X, Y, Z, \ldots$.

Terms: A term variable or a compound term of the form $f(s_1, \ldots, s_n)$.

Hedges: A sequence $s_1, \ldots, s_n$ where each s_i is either a hedge variable or a term.

Substitutions

Substitution: a mapping

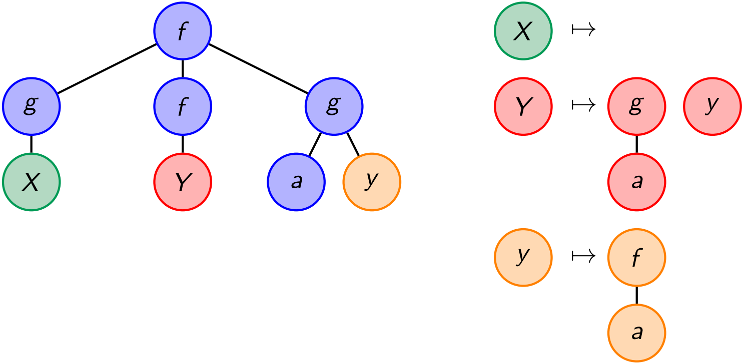
- from term variables to terms,
- from hedge variables to hedges,

which is identity almost everywhere.

Terms and Substitutions

Example

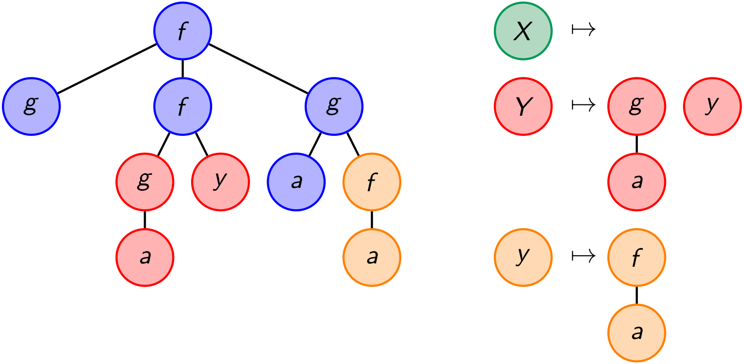
$f(g(X), f(Y), g(a, y))$ $\{X \mapsto (), Y \mapsto (g(a), y), y \mapsto f(a)\}$



Terms and Substitutions

Example

$$f(g, f(g(a), y), g(a, f(a))) \quad \{X \mapsto (), Y \mapsto (g(a), y), y \mapsto f(a)\}$$



Hedge Generalization

A hedge $\tilde{s}$ is a generalization (anti-instance) of the hedges $\tilde{s}_1$ and $\tilde{s}_2$ if

- $\tilde{s}$ is more general than $\tilde{s}_1$ ($\tilde{s} \preceq \tilde{s}_1$) and
- $\tilde{s}$ is more general than $\tilde{s}_2$ ($\tilde{s} \preceq \tilde{s}_2$),

i.e., if there exist σ_1 and σ_2 such that

- $\tilde{s}\sigma_1 = \tilde{s}_1$ and
- $\tilde{s}\sigma_2 = \tilde{s}_2$.

Hedge Generalization

A hedge $\tilde{s}$ is a least general generalization (lgg) of the hedges $\tilde{s}_1$ and $\tilde{s}_2$ if

- $\tilde{s}$ is a generalization of $\tilde{s}_1$ and $\tilde{s}_2$ and
- no generalization of $\tilde{s}_1$ and $\tilde{s}_2$ is strictly less general than $\tilde{s}$.

Minimal Complete Set of Generalizations

A *minimal complete set of generalizations* of $\tilde{s}_1$ and $\tilde{s}_2$ is a set $\mathcal{G}$ of hedges that satisfies the properties:

Soundness: Each $\tilde{q} \in G$ is a generalization of both $\tilde{s}_1$ and $\tilde{s}_2$.

Completeness: For each generalization $\tilde{s}$ of $\tilde{s}_1$ and $\tilde{s}_2$, there exists $\tilde{q} \in G$ such that $\tilde{s} \preceq \tilde{q}$.

Minimality: For each $\tilde{q}_1, \tilde{q}_2 \in G$, if $\tilde{q}_1 \preceq \tilde{q}_2$ then $\tilde{q}_1 = \tilde{q}_2$.

The Anti-Unification Problem

Given: Two hedges $\tilde{s}_1$ and $\tilde{s}_2$.

Find: The minimal complete set of generalizations of $\tilde{s}_1$ and $\tilde{s}_2$.

Some Generalizations Might be Unexpected

Example

- What is the minimal complete set of generalizations of $g(f(a), f(a))$ and $g(f(a), f)$?

Some Generalizations Might be Unexpected

Example

- What is the minimal complete set of generalizations of $g(f(a), f(a))$ and $g(f(a), f)$?
- One might expect that it is $\{g(f(a), f(X))\}$.

Some Generalizations Might be Unexpected

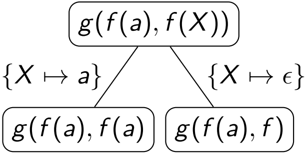
Example

- What is the minimal complete set of generalizations of $g(f(a), f(a))$ and $g(f(a), f)$?
- One might expect that it is $\{g(f(a), f(X))\}$.
- But it contains three elements:

Some Generalizations Might be Unexpected

Example

- What is the minimal complete set of generalizations of $g(f(a), f(a))$ and $g(f(a), f)$?
- One might expect that it is $\{g(f(a), f(X))\}$.
- But it contains three elements:



$g(f(a), f(X))$

$\{X \mapsto a\}$

$\{X \mapsto \epsilon\}$

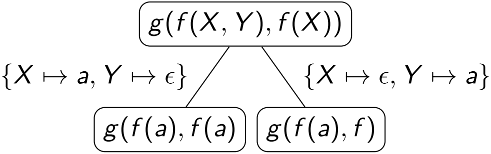
$g(f(a), f(a)$

$g(f(a), f)$

Some Generalizations Might be Unexpected

Example

- What is the minimal complete set of generalizations of $g(f(a), f(a))$ and $g(f(a), f)$?
- One might expect that it is $\{g(f(a), f(X))\}$.
- But it contains three elements:



$g(f(X, Y), f(X))$

$\{X \mapsto a, Y \mapsto \epsilon\}$

$\{X \mapsto \epsilon, Y \mapsto a\}$

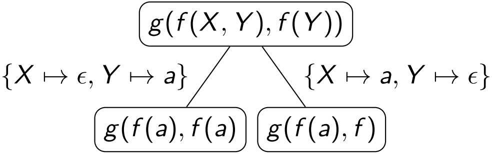
$g(f(a), f(a)$

$g(f(a), f)$

Some Generalizations Might be Unexpected

Example

- What is the minimal complete set of generalizations of $g(f(a), f(a))$ and $g(f(a), f)$?
- One might expect that it is $\{g(f(a), f(X))\}$.
- But it contains three elements:



$g(f(X, Y), f(Y))$

$\{X \mapsto \epsilon, Y \mapsto a\}$

$\{X \mapsto a, Y \mapsto \epsilon\}$

$g(f(a), f(a)$

$g(f(a), f)$

Rigid Unranked Generalizations: Idea

- Emphasis on keeping the common structure, rather than on uniform generalization of distinct parts.
- Avoiding consecutive hedge variables in the generalization.

Rigid Unranked Generalizations: Idea

More specifically:

- Given two hedges $f_1(\tilde{s}_1), \ldots, f_n(\tilde{s}_n)$ and $g_1(\tilde{r}_1), \ldots, g_m(\tilde{r}_m)$.

Rigid Unranked Generalizations: Idea

More specifically:

- Given two hedges $f_1(\tilde{s}_1), \ldots, f_n(\tilde{s}_n)$ and $g_1(\tilde{r}_1), \ldots, g_m(\tilde{r}_m)$.
- Take a common subsequence of $f_1, \ldots, f_n$ and $g_1, \ldots, g_m$.

Rigid Unranked Generalizations: Idea

More specifically:

- Given two hedges $f_1(\tilde{s}_1), \dots, f_n(\tilde{s}_n)$ and $g_1(\tilde{r}_1), \dots, g_m(\tilde{r}_m)$.
- Take a common subsequence of $f_1, \dots, f_n$ and $g_1, \dots, g_m$.
- Let it be $h_1, \dots, h_k$.

Rigid Unranked Generalizations: Idea

More specifically:

- Given two hedges $f_1(\tilde{s}_1), \ldots, f_n(\tilde{s}_n)$ 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 hedges has a form

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

Rigid Unranked Generalizations: Idea

More specifically:

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

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

  where
  - X's are (not necessarily distinct) new hedge 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 $\tilde{r}_l$.

Rigid Unranked Generalizations: Idea

More specifically:

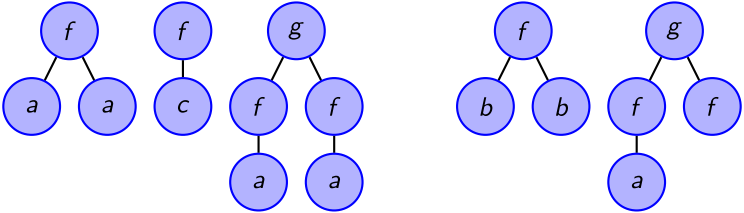
- Given two hedges $f_1(\tilde{s}_1), \dots, f_n(\tilde{s}_n)$ and $g_1(\tilde{r}_1), \dots, g_m(\tilde{r}_m)$.
- Take a common subsequence of $f_1, \dots, f_n$ and $g_1, \dots, g_m$.
- Let it be $h_1, \dots, h_k$.
- Then a rigid generalization of the given hedges has a form

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

  where
  - X's are (not necessarily distinct) new hedge 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.

Rigid Unranked Generalizations: How the Idea Works

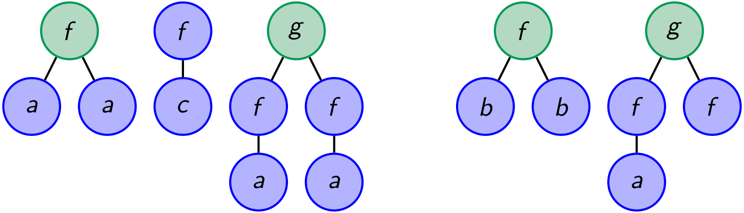
Example



Rigidity function computes longest common subsequences.

Rigid Unranked Generalizations: How the Idea Works

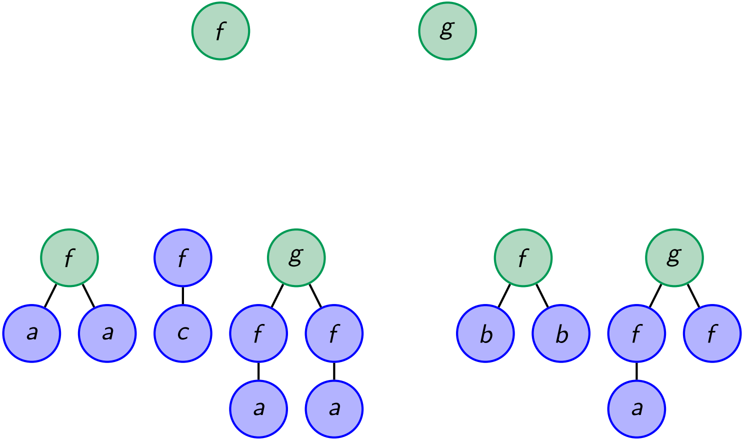
Example



Rigidity function computes longest common subsequences.

Rigid Unranked Generalizations: How the Idea Works

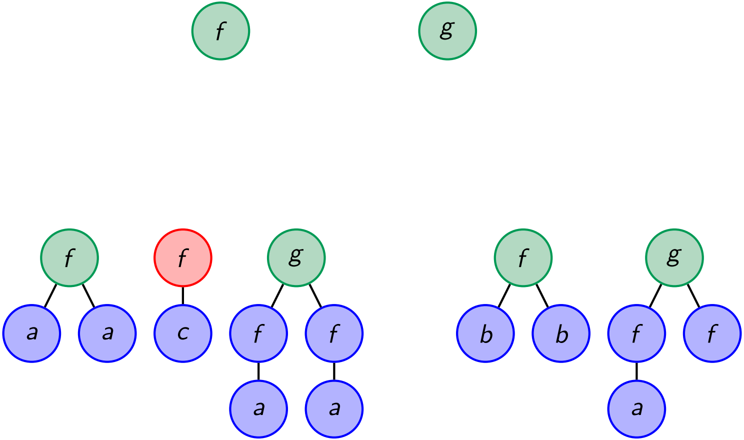
Example



Rigidity function computes longest common subsequences.

Rigid Unranked Generalizations: How the Idea Works

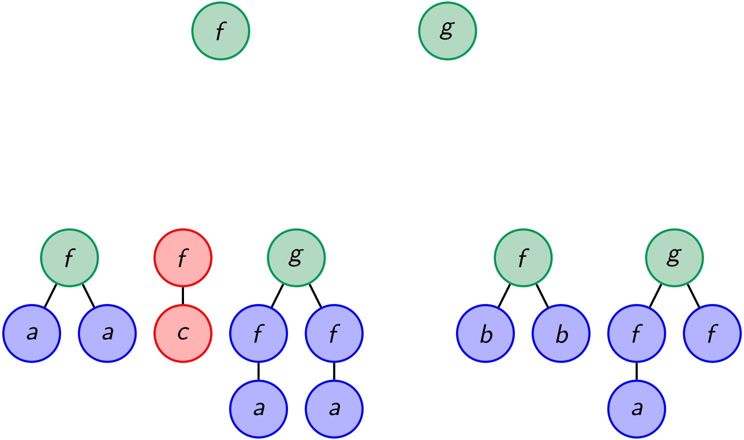
Example



Rigidity function computes longest common subsequences.

Rigid Unranked Generalizations: How the Idea Works

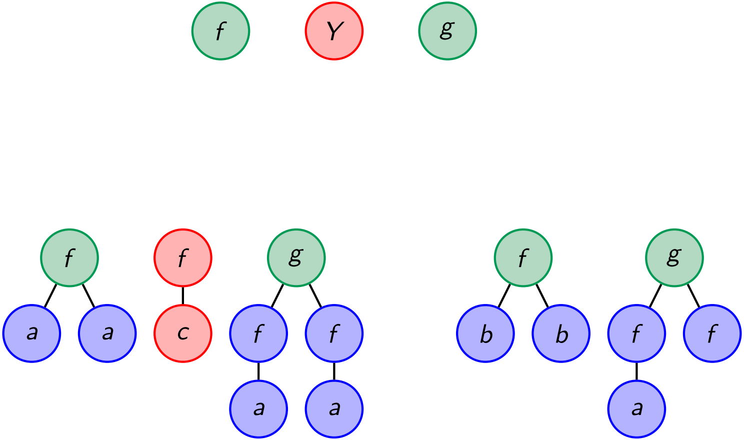
Example



Rigidity function computes longest common subsequences.

Rigid Unranked Generalizations: How the Idea Works

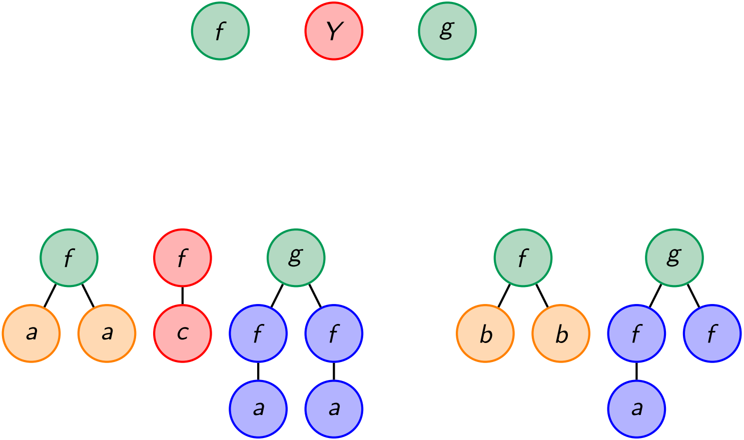
Example



Rigidity function computes longest common subsequences.

Rigid Unranked Generalizations: How the Idea Works

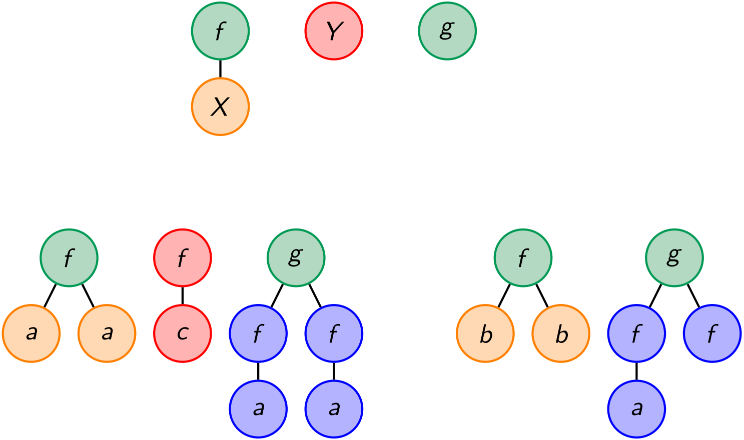
Example



Rigidity function computes longest common subsequences.

Rigid Unranked Generalizations: How the Idea Works

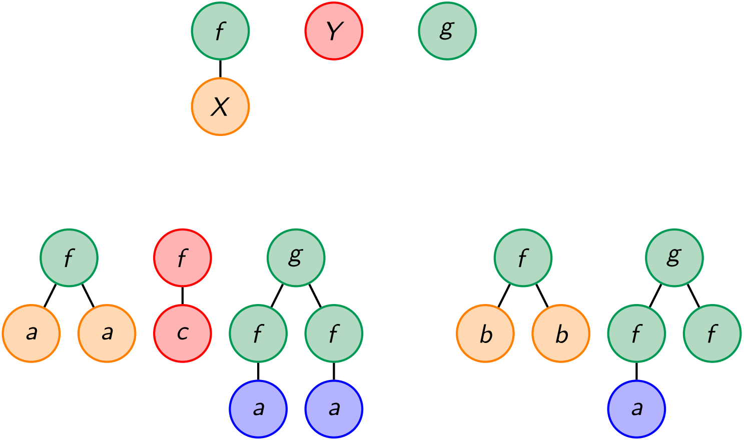
Example



Rigidity function computes longest common subsequences.

Rigid Unranked Generalizations: How the Idea Works

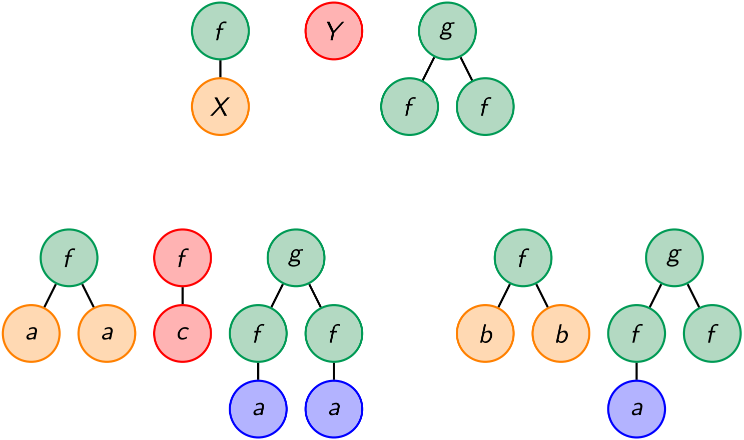
Example



Rigidity function computes longest common subsequences.

Rigid Unranked Generalizations: How the Idea Works

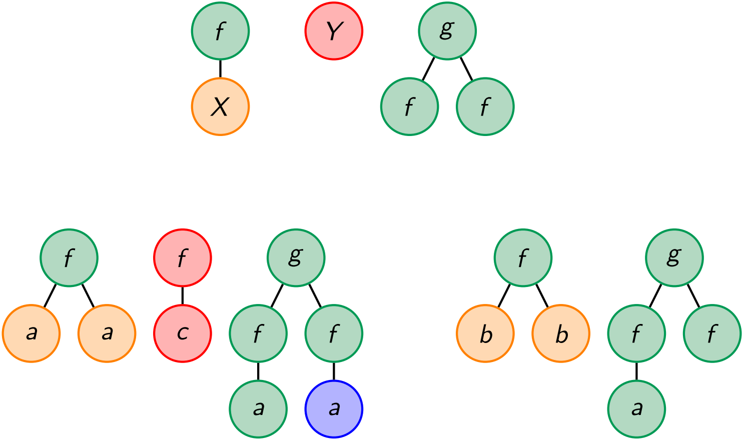
Example



Rigidity function computes longest common subsequences.

Rigid Unranked Generalizations: How the Idea Works

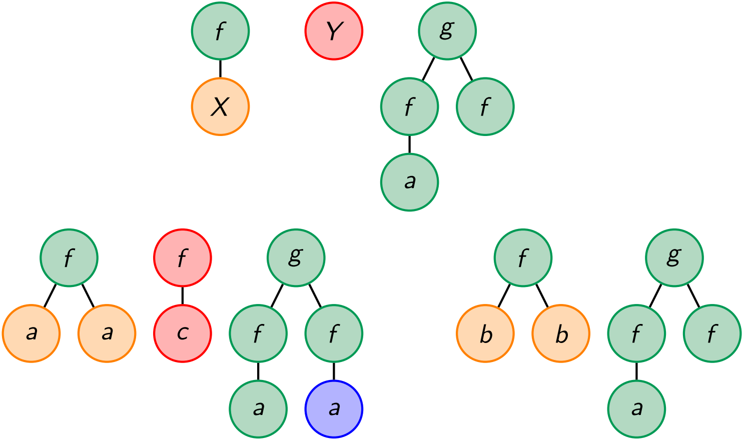
Example



Rigidity function computes longest common subsequences.

Rigid Unranked Generalizations: How the Idea Works

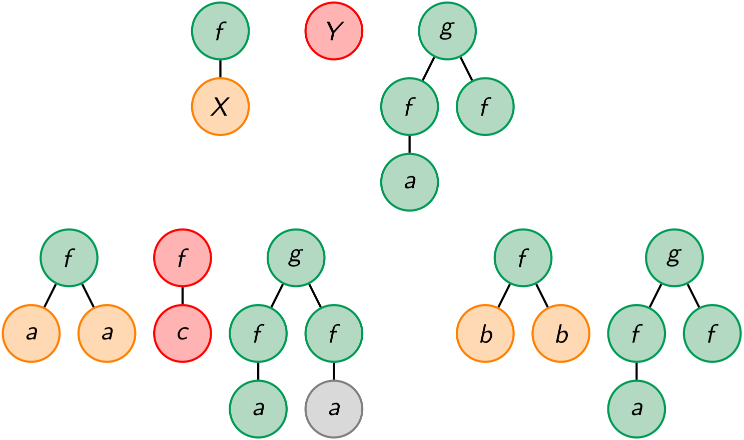
Example



Rigidity function computes longest common subsequences.

Rigid Unranked Generalizations: How the Idea Works

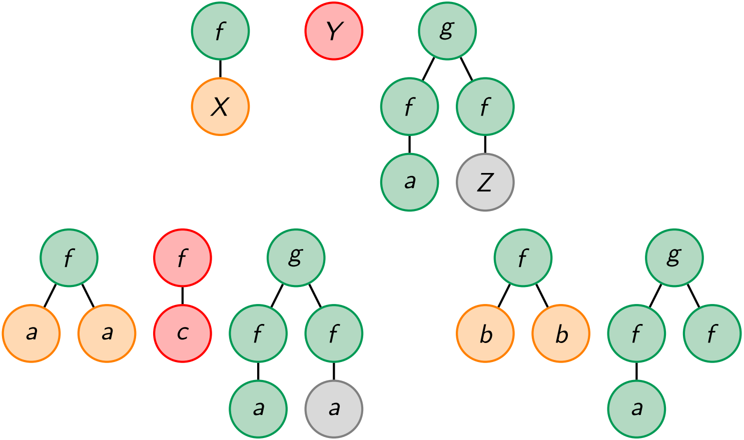
Example



Rigidity function computes longest common subsequences.

Rigid Unranked Generalizations: How the Idea Works

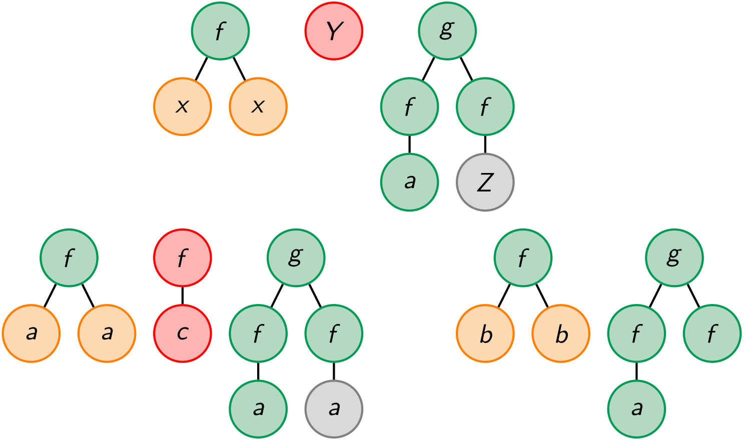
Example



Rigidity function computes longest common subsequences.

Rigid Unranked Generalizations: How the Idea Works

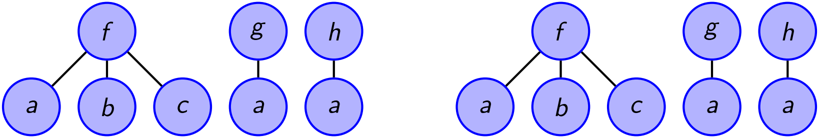
Example



Rigidity function computes longest common subsequences.

Rigid Unranked Generalizations: How the Idea Works

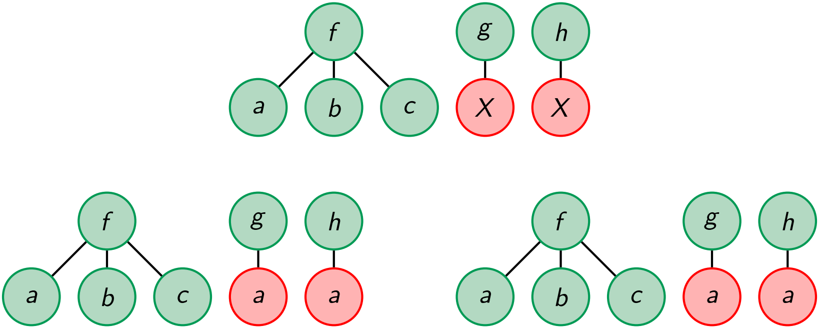
Example



Rigidity function computes longest common subsequences of length at least 3.

Rigid Unranked Generalizations: How the Idea Works

Example



Rigidity function computes longest common subsequences of length at least 3.

Rigid Unranked Anti-Unification Algorithm

- Anti-unification equations:

$$X : \tilde{s} \triangleq \tilde{r},$$

  meaning: X is a generalization of $\tilde{s}$ and $\tilde{r}$.

- The rule-based algorithm works on triples:

$$A; S; \sigma,$$

  where A is a set of anti-unification equations, S is a set of already solved anti-unification equations, σ is a substitution computed so far.

Rigid Unranked Anti-Unification Algorithm: Rules

$\mathcal{R}$: The rigidity function.

$\mathcal{R}$-Dec-H: $\mathcal{R}$-**Rigid Decomposition for Hedges**

$$
\begin{aligned}
&\{X : \tilde{s} \triangleq \tilde{q}\} \cup A;\ S;\ \sigma \Longrightarrow \\
&\qquad \{Z_k : \tilde{s}_k \triangleq \tilde{q}_k \mid 1 \leq k \leq n\} \cup A; \\
&\qquad \{Y_0 : \tilde{s}|_0^{i_1} \triangleq \tilde{q}|_0^{j_1}\} \cup \{Y_k : \tilde{s}|_{i_k}^{i_{k+1}} \triangleq \tilde{q}|_{j_k}^{j_{k+1}} \mid 1 \leq k \leq n-1\} \cup \\
&\qquad\quad \{Y_n : \tilde{s}|_{i_n}^{|\tilde{s}|+1} \triangleq \tilde{q}|_{j_n}^{|\tilde{q}|+1}\} \cup S; \\
&\qquad \sigma\{X \mapsto Y_0, f_1(Z_1), Y_1, \ldots, Y_{n-1}, f_n(Z_n), Y_n\},
\end{aligned}
$$

if $\mathcal{R}(top(\tilde{s}), top(\tilde{q}))$ contains a sequence $f_1[i_1, j_1] \cdots f_n[i_n, j_n]$ such that for all $1 \leq k \leq n$, $\tilde{s}|_{i_k} = f_k(\tilde{s}_k)$, $\tilde{q}|_{j_k} = f_k(\tilde{q}_k)$, and Y_0, Y_k's and Z_k's are fresh.

Rigid Unranked Anti-Unification Algorithm: Rules

$\mathcal{R}$-S-H: $\mathcal{R}$-**Rigid Solve for Hedges**

$$\{X : \tilde{s} \triangleq \tilde{q}\} \cup A;\ S;\ \sigma \Longrightarrow A;\ \{X : \tilde{s} \triangleq \tilde{q}\} \cup S;\ \sigma,$$

if $\mathcal{R}(top(\tilde{s}), top(\tilde{q})) = \emptyset$.

$\mathcal{R}$-CS1: $\mathcal{R}$-**Rigid Clean Store 1**

$$A;\ \{X_1 : \tilde{s} \triangleq \tilde{q}, X_2 : \tilde{s} \triangleq \tilde{q}\} \cup S;\ \sigma \Longrightarrow$$
$$A;\ \{X_1 : \tilde{s} \triangleq \tilde{q}\} \cup S;\ \sigma\{X_2 \mapsto X_1\}.$$

$\mathcal{R}$-CS2: $\mathcal{R}$-**Rigid Clean Store 2**

$$A;\ \{X : \epsilon \triangleq \epsilon\} \cup S;\ \sigma \Longrightarrow A;\ S;\ \sigma\{X \mapsto \epsilon\}.$$

Rigid Unranked Anti-Unification Algorithm: Rules

$\mathcal{R}$-CS3: $\mathcal{R}$-**Rigid Clean Store 3**

$$\begin{aligned} &A;\ \{x_1 : l \triangleq r,\ x_2 : l \triangleq r\} \cup S;\ \sigma \Longrightarrow \\ &\qquad A;\ \{x_1 : l \triangleq r\} \cup S;\ \sigma\{x_2 \mapsto x_1\}. \end{aligned}$$

$\mathcal{R}$-CS4: $\mathcal{R}$-**Rigid Clean Store 4**

$$\begin{aligned} &A;\ \{X : l_1, \ldots, l_n \triangleq r_1, \ldots, r_n\} \cup S;\ \sigma \Longrightarrow \\ &\qquad A;\ \{x_1 : l_1 \triangleq r_1, \ldots, x_n : l_n \triangleq r_n\} \cup S;\ \sigma\{X \mapsto x_1, \ldots, x_n\}, \end{aligned}$$

where $n \geq 1$ and x_i's are fresh.

Rigid Unranked Anti-Unification Algorithm: Control

- Given a rigidity function $\mathcal{R}$, to compute $\mathcal{R}$-generalizations of hedges $\tilde{s}$ and $\tilde{q}$, start with $\{X : \tilde{s} \triangleq \tilde{q}\}; \emptyset; \mathit{Id}$ and apply the rules exhaustively.

Rigid Unranked Anti-Unification Algorithm: Properties

Theorem (Termination)

The algorithm terminates on any input and produces a system $\emptyset; S; \sigma$.

Theorem (Soundness)

If the algorithm produces the derivation

$$\{X : \tilde{s}_1 \triangleq \tilde{s}_2\}; \emptyset; Id \Longrightarrow^{*} \emptyset; S; \sigma$$

then $X\sigma$ *is a rigid generalization of* $\tilde{s}_1$ *and* $\tilde{s}_2$.

Theorem (Completeness)

Let $\tilde{q}$ *be a rigid generalization of* $\tilde{s}_1$ *and* $\tilde{s}_2$. *Then the algorithm computes a rigid anti-unifier* σ *for* $X : \tilde{s}_1 \triangleq \tilde{s}_2$ *such that* $\tilde{q} \preceq X\sigma$.

Rigid Unranked Anti-Unification Algorithm: Example

Example

- $\mathcal{R}$ computes the set of longest common subsequences.
- $\mathcal{R}$-generalization of the hedges $f(a, a), f(c), g(f(a), f(a))$ and $f(b, b), g(f(a), f)$.

Rigid Unranked Anti-Unification Algorithm: Example

Example

- $\mathcal{R}$ computes the set of longest common subsequences.
- $\mathcal{R}$-generalization of the hedges $f(a,a), f(c), g(f(a), f(a))$ and $f(b,b), g(f(a), f)$.
- Create the initial system:

$$\{X : f(a,a), f(c), g(f(a), f(a)) \triangleq f(b,b), g(f(a), f)\}, \emptyset, Id.$$

Rigid Unranked Anti-Unification Algorithm: Example

Example

- $\mathcal{R}$ computes the set of longest common subsequences.
- $\mathcal{R}$-generalization of the hedges $f(a,a), f(c), g(f(a), f(a))$ and $f(b,b), g(f(a), f)$.
- Create the initial system:

$$\{X : f(a,a), f(c), g(f(a), f(a)) \triangleq f(b,b), g(f(a), f)\}, \emptyset, Id.$$

- Obtain two terminal systems:
  1. $\emptyset, \{x : a \triangleq b, Y : f(c) \triangleq \epsilon, Z : a \triangleq \epsilon\};$
     $\{X \mapsto f(x,x), Y, g(f(a), f(Z))\}$
  2. $\emptyset, \{Y : f(a,a) \triangleq \epsilon, Z : c \triangleq b, b, U : a \triangleq \epsilon\};$
     $\{X \mapsto Y, f(Z), g(f(a), f(U))\}$

Rigid Unranked Anti-Unification Algorithm: Example

Example

- $\mathcal{R}$ computes the set of longest common subsequences.
- $\mathcal{R}$-generalization of the hedges $f(a,a), f(c), g(f(a), f(a))$ and $f(b,b), g(f(a), f)$.
- Create the initial system:

$$\{X : f(a,a), f(c), g(f(a), f(a)) \triangleq f(b,b), g(f(a), f)\}, \emptyset, Id.$$

- Obtain two terminal systems:
  1. $\emptyset, \{x : a \triangleq b, Y : f(c) \triangleq \epsilon, Z : a \triangleq \epsilon\}$;
     $\{X \mapsto f(x,x), Y, g(f(a), f(Z))\}$
  2. $\emptyset, \{Y : f(a,a) \triangleq \epsilon, Z : c \triangleq b, b, U : a \triangleq \epsilon\}$;
     $\{X \mapsto Y, f(Z), g(f(a), f(U))\}$
- The store tells how to obtain each original hedge from the generalization.

Rigid Anti-Unification: Some Interesting Facts

- By choosing appropriate rigidity functions, rigid unranked anti-unification can model various existing generalization algorithms:
    - Simple hedge anti-unification for inductive reasoning over semi-structured documents (Yamamoto et al., 2001).
    - Word anti-unification (Cicekli and Ciceckli, 2006).
    - ϵ-free word anti-unification (Biere, 2003).
    - First-order anti-unification (Plotkin, 1972, Reynolds, 1972).
- Combination of rigid and complete (non-rigid) anti-unification algorithms can simulate AU anti-unification (Alpuente et al., 2008)

Rigid Anti-Unification and Clone Detection

Unranked representation of code pieces:

```
if(>=(a, b),
   then(=(c, +(d, b)),
        =(d, +(d, 1))),
   else(=(c, -(d, a))))
```

```
if(>=(m, n),
   then(=(y, +(x, n)),
        =(z, 1),
        =(x, +(x, 5))),
   else(=(y, -(x, m))))
```

An interesting generalization:

```
if(>=(y1, y2),
   then(=(y3, +(y4, y2)),
        Y,
        =(y4, +(y4, y5))),
   else(=(y3, -(y4, y1))))
```

- $\{\mathtt{y1} \mapsto \mathtt{a}, \mathtt{y2} \mapsto \mathtt{b}, \mathtt{y3} \mapsto \mathtt{c}, \mathtt{y4} \mapsto \mathtt{d}, \mathtt{y5} \mapsto \mathtt{1}, \mathtt{Y} \mapsto \epsilon\}$
- $\{\mathtt{y1} \mapsto \mathtt{m}, \mathtt{y2} \mapsto \mathtt{n}, \mathtt{y3} \mapsto \mathtt{y}, \mathtt{y4} \mapsto \mathtt{x}, \mathtt{y5} \mapsto \mathtt{5}, \mathtt{Y} \mapsto \texttt{=(z, 1)}\}$

Rigid Anti-Unification and Clone Detection

> *Rigid generalisation comes as a way to express many (maybe all) interesting practical techniques [of clone detection].*

Anonymous referee of (Kutsia, Levy, Villaret, 2011)

Rigid Anti-Unification and Clone Detection

- Rigid anti-unification helps to detect inserted or deleted pieces of code, which is necessary for clones of type 3.
- If we are interested in clones whose length is greater than a predefined threshold, we can include this measure in the definition of the rigidity function.
- The approach is modular, where most of the computations are performed on strings. It may combine advantages of fast textual and precise structural techniques and consider rigidity functions modulo a given metrics.

Rigid Anti-Unification and Clone Detection

- Anti-unifiers reflect similarities between two inputs, while the store reflects differences between them.
- The output of anti-unification can be used for comparison utilities and for extracting a procedure. This process has a use in code refactoring.
- Rigid anti-unification works on unranked terms that can abstract XML documents. How to detect clones well in XML/HTML is mentioned as one of the open problems in clone detection research in (Roy and Cordy, 2007).