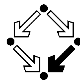
SOLVING AND COMPUTING WITH VARIADIC TERMS

Temur Kutsia
RISC, Johannes Kepler University Linz



Variadic languages

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

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

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

Sequences: $s_1, \ldots, s_n$, $n \geq 0$, where each s_i is either a sequence variable or a term.

Variadic languages

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

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

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

Sequences: $s_1, \ldots, s_n$, $n \geq 0$, where each s_i is either a sequence variable or a term.

Intuitive explanation rather than a formal definition.

For readability, sequences usually are put in the parentheses: $(s_1, \ldots, s_n)$.

Properties of sequences

Sequences are flat:

$$(s_1, s_2, (t_1, t_2), s_3, ()) = (s_1, s_2, t_1, t_2, s_3).$$

Sequences also flatten under function symbols:

$$f(a, (b, (c, d)), ()) = f(a, b, c, d).$$

Singleton sequence and its element are not distinguished:

$$(t) = t.$$

Terminology

Alternative names for variadic terms:

- flexary terms
- flexible arity terms
- polyadic terms
- multi-ary terms
- unranked terms

Alternative names for variadic term sequences:

- hedges
- (variadic, flexary, $\ldots$) forests

Terminology

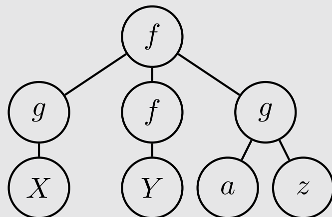
Sometimes term variables are called individual variables.

Sequence variables are also referred to as hedge variables.

Variadic terms

Example

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

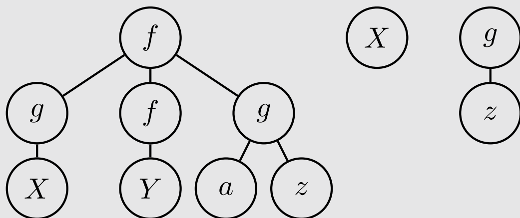


- The arity of function symbols is not fixed.
- Different occurrences of the same function symbol may have different number of arguments.

Variadic term sequences

Example

$$f(g(X), f(Y), g(a, z)), \quad X, \quad g(z)$$



- Finite, possibly empty, sequences of variadic terms.

Substitutions

Substitution: a mapping

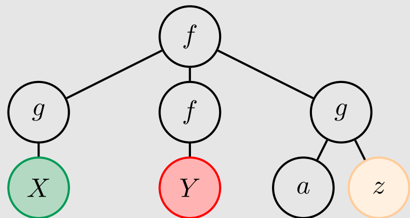
- from term variables to terms,
- from sequence variables to sequences,

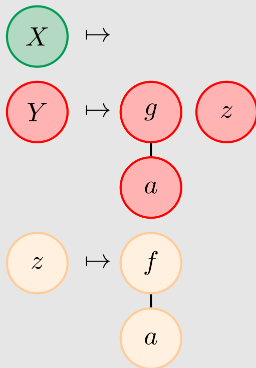
which is identity almost everywhere.

Terms and substitutions

Example

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

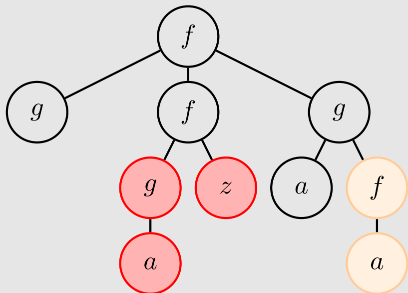


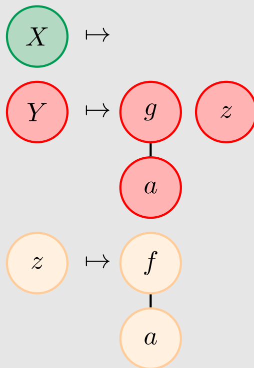


Terms and substitutions

Example

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



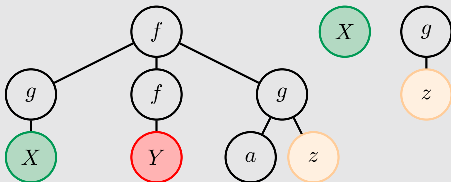


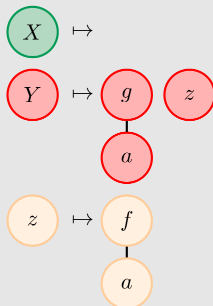
Terms and substitutions

Example

$f(g(X), f(Y), g(a, z)), \quad X, \quad g(z)$

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



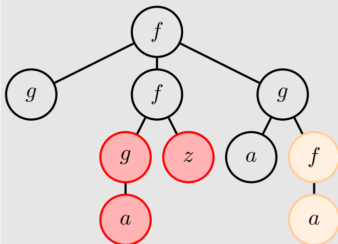


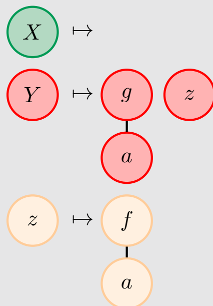
Terms and substitutions

Example

$f(g, f(g(a), z), g(a, f(a))), \quad g(f(a))$

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





Variadic terms and sequences

Convenient and useful feature for

- formalizing mathematical texts
  (LFS, OpenMath, MathML, Mizar, Theorema),
- interchange languages
  (KIF, Common Logic)
- representing symbolic computation data structures
  (Mathematica, Theorema),
- modeling XML documents
  (XCentric),
- expressing patterns in declarative programming
  (Wolfram, ρLog),
- etc.

Outline

Solving

Computing

Solving problems

Unification:

Given: Two (variadic) terms s and t.
Find: A substitution σ such that $s\sigma = t\sigma$.

σ: a unifier of s and t.

If such a σ exists, s and t are unifiable.

Example

$$s = f(X, f(a), Y)$$
$$t = f(a, f(z), b, f(u), c)$$
$$\sigma = \{X \mapsto a, z \mapsto a, Y \mapsto (b, f(u), c)\}$$
$$s\sigma = f(a, f(a), b, f(u), c) = t\sigma$$

Solving problems

Matching:

> **Given:** Two (variadic) terms s and t.
> **Find:** A substitution σ such that $s\sigma = t$.

σ: a matcher of s to t.

If such a σ exists, s matches t.

Example

$$s = f(X_1, y, X_2, y, X_3)$$
$$t = f(a, f(a), f(a), a, b)$$
$$\sigma = \{y \mapsto a, X_1 \mapsto (), X_2 \mapsto (f(a), f(a)), X_3 \mapsto b\}$$
$$s\sigma = f(a, f(a), f(a), a, b) = t$$

Solving problems

Anti-unification:

Given: Two (variadic) terms s and t.
Find: A (variadic) term r that matches both s and t.

r: a generalization of s and t.

Example

$$s = g(f(a), b, c, f(a), b, c)$$

$$t = g(f(a), f)$$

$$r = g(f(a), X, f(Y), X)$$

$$r\{X \mapsto (b, c), Y \mapsto a\} = s$$

$$r\{X \mapsto (), Y \mapsto ()\} = t$$

Notation

Unification problem: $s =^? t$.

Matching problem: $s \preceq^? t$.

Anti-unification problem $s \triangleq^? t$.

Unification

Word unification is a special case of variadic unification.

It uses

- one variadic function symbol on the top position,
- any other function symbol without arguments,
- only sequence variables (no term variables).

Word unification problem

$$XaY =^? aXabc$$

encoded as a variadic unification problem

$$f(X, a, Y) =^? f(a, X, a, b, c).$$

Unification

Word unification is a well-studied classical problem.

It is decidable [Makanin 1977] and infinitary [Plotkin 1970, Siekmann 1978].

Variadic unification is also decidable and infinitary.

Example

Variadic unification problem: $f(X, a) =^? f(a, X)$

Unifiers:

$\{X \mapsto ()\}$

$\{X \mapsto a\}$

$\{X \mapsto (a, a)\}$

$\ldots$

Unification: unitary/finitary fragments

Fragment 1: the KIF fragment (KIF).

KIF: Knowledge Interchange Format.

Sequence variables occupy only the last argument position in each subterm where they occur.

Unification type is unitary: a single most general unifier exists.

Example

$$f(f(a, X), g(x, y, X), Y) =^? f(f(x, a, Y), g(a, Z), U)$$

mgu: $\{x \mapsto a, X \mapsto (a, U), Y \mapsto U, Z \mapsto (y, a, U)\}$

common instance: $f(f(a, a, U), g(a, y, a, U), U)$

Unification: unitary/finitary fragments

Fragment 2: the linear fragment (LIN).

In unification problems, no variable appears more than once.

Unification type is finitary: every solvable problem has a finite set of incomparable most general unifiers.

Example

$$f(X, f(x, c)) =^? f(a, b, f(Y, Z))$$

mcsu: $\{\sigma_1, \sigma_2, \sigma_3\}$, where

$$\sigma_1 = \{X \mapsto (a, b), Y \mapsto (), Z \mapsto (x, c)\},$$

$$\sigma_2 = \{X \mapsto (a, b), Y \mapsto x, Z \mapsto c\}$$

$$\sigma_3 = \{X \mapsto (a, b), Y \mapsto (x, c), Z \mapsto ()\}$$

common instance for σ_1: $f(a, b, f(x, c))$

Unification: unitary/finitary fragments

Fragment 3: unique postfix fragment (UPOST).

The sequence $t_1, \ldots, t_n$ is called a postfix of X in the term $f(s_1, \ldots, s_m, X, t_1, \ldots, t_n)$.

For instance, the postfixes of X in $f(a, X, b, X, g(X))$ are $(b, X, g(X))$ and $g(X)$.

In the unique postfix fragment of variadic unification, each sequence variable occurring in the unification problem has the same postfix in all subterms it occurs.

Unification: unitary/finitary fragments

Fragment 3: unique postfix fragment (UPOST).

Example

$$f(f(a, X, f(Y, b), y), g(Z, U)) =^? f(x, g(X, f(Y, b), y))$$

UPOST-unification problem.

Postfixes of variables:

For X: $(f(Y, b), y)$

For Y: b

For Z: U

For U: $()$

Unification: unitary/finitary fragments

Fragment 3: unique postfix fragment (UPOST).

KIF, LIN: special cases of UPOST.

UPOST is finitary.

Example

$$f(X, f(Y, b), Z, b) =^? f(f(a, b), f(b), Y, b)$$

mcsu: $\{\sigma_1, \sigma_2\}$, where

$$\sigma_1 = \{X \mapsto (), Y \mapsto a, Z \mapsto (f(b), a)\},$$

$$\sigma_2 = \{X \mapsto f(a, b), Y \mapsto (), Z \mapsto ()\}$$

common instance for σ_1: $f(f(a, b), f(b), a, b)$

common instance for σ_2: $f(f(a, b), f(b), b)$

Unification: unitary/finitary fragments

Fragment 4: inverse KIF fragment (I-KIF).

In unification problems, sequence variables occupy only the first argument position in each subterm where they occur.

I-KIF is unitary.

Fragment 5: unique prefix fragment (UPREF).

Dual to UPOST: every sequence variable has the same prefix in all subterms.

UPREF is finitary.

I-KIF and LIN are special cases of UPREF.

Unification: unitary/finitary fragments

Fragment 6: unique variables in one side (UV).

In unification problem $s =^? t$, each variable that occurs in t has the unique occurrence in the problem.

Example

$f(X, a, X) =^? f(a, Y, a)$ is in UV.

$f(f(a, X), f(X, a)) =^? f(y, z)$ is in UV.

$f(f(a, X), f(X, a)) =^? f(y, y)$ is not in UV.

$f(a, X) =^? f(X, a)$ is not in UV.

UV is finitary.

Unification: unitary/finitary fragments

Fragment 7: matching fragment (M).

Matching problem $s \preceq^? t$ can be also seen as a special case of unification, when t does not contain variables.

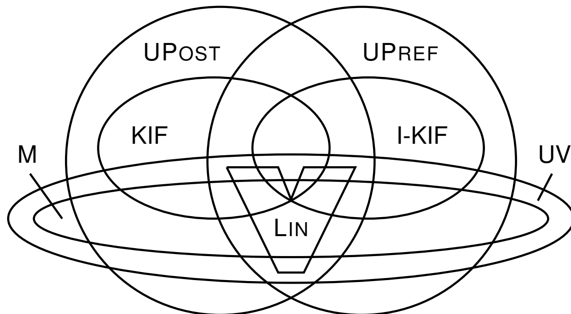
It is a special case of UV and is finitary.

Example

$$s = f(X_1, y, X_2, y, X_3)$$
$$t = f(a, f(a), f(a), a, b)$$
$$\sigma_1 = \{y \mapsto a, X_1 \mapsto (), X_2 \mapsto (f(a), f(a)), X_3 \mapsto b\}$$
$$\sigma_2 = \{y \mapsto f(a), X_1 \mapsto a, X_2 \mapsto (), X_3 \mapsto (a, b)\}$$

Unification: unitary/finitary fragments

Relations between various unitary or finitary fragments:



Unification: termination

Unification procedures from [Kutsia 2007] and [Kutsia & Marin, 2012] stop for these unitary or finitary fragments.

In general, termination is not granted even if the problem is unitary/finitary, because there might exist infinitely failing derivations (unless the decision algorithm is incorporated in the computation process).

It is possible to obtain a terminating algorithm for a special kind of infinitary problem, where no variable occurs more than twice: requires cycle checks and special representation of solutions.

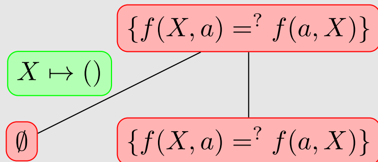
Unification Procedure

Example

$$\{f(X, a) =^? f(a, X)\}$$

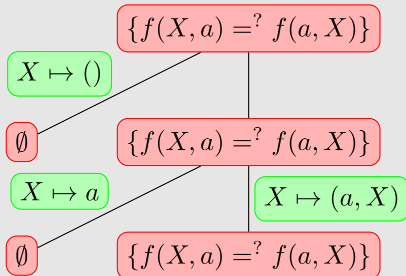
Unification Procedure

Example



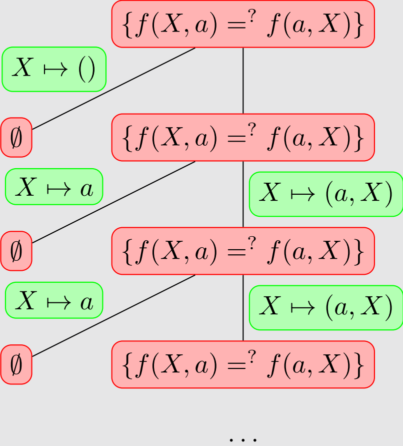
Unification Procedure

Example



Unification Procedure

Example



Matching

Variadic matching is NP-complete.

Hardness can be shown by reducing from positive 1-in-3-SAT.

In positive 1-in-3-SAT, each clause contains three propositional variables and exactly one from them should be assigned **true**.

Matching

Variadic matching is NP-complete.

Hardness can be shown by reducing from positive 1-in-3-SAT.

In positive 1-in-3-SAT, each clause contains three propositional variables and exactly one from them should be assigned **true**.

Reduction:

- Introduce a sequence variable X_p for each propositional variable p appearing in a positive 1-in-3-SAT problem.
- Translate each clause $p_1 \vee p_2 \vee p_3$ into a variadic matching problem $f(X_{p_1}, X_{p_2}, X_{p_3}) \preceq^? f(t)$.
- Three possible matchers:

  $\{X_{p_i} \mapsto t, X_{p_j} \mapsto (), X_{p_k} \mapsto ()\}$ for $\{i, j, k\} = \{1, 2, 3\}$

  $\rightsquigarrow p_i =$ **true**, $p_j =$ **false**, $p_k =$ **false**

Matching

Variadic matching is NP-complete.

Hardness can be shown by reducing from positive 1-in-3-SAT.

In positive 1-in-3-SAT, each clause contains three propositional variables and exactly one from them should be assigned **true**.

Reduction:

- Introduce a sequence variable X_p for each propositional variable p appearing in a positive 1-in-3-SAT problem.
- Translate each clause $p_1 \vee p_2 \vee p_3$ into a variadic matching problem $f(X_{p_1}, X_{p_2}, X_{p_3}) \preceq^? f(t)$.
- Three possible matchers:

  $\{X_{p_i} \mapsto t, X_{p_j} \mapsto (), X_{p_k} \mapsto ()\}$ for $\{i, j, k\} = \{1, 2, 3\}$

  $\rightsquigarrow p_i =$ **true**, $p_j =$ **false**, $p_k =$ **false**

Linear variadic matching can be decided in time $O(n^3)$.

Anti-unification

Variadic anti-unification is finitary.

Example

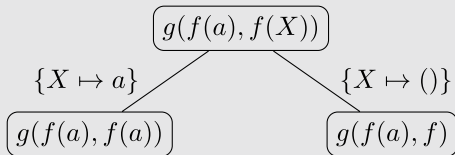
$g(f(a), f(a))$ and $g(f(a), f)$ have three incomparable least general generalizations:

Anti-unification

Variadic anti-unification is finitary.

Example

$g(f(a), f(a))$ and $g(f(a), f)$ have three incomparable least general generalizations:

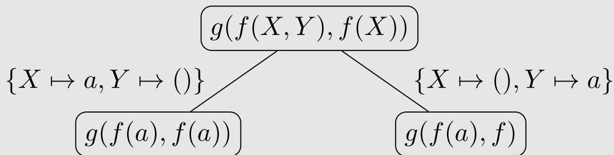


Anti-unification

Variadic anti-unification is finitary.

Example

$g(f(a), f(a))$ and $g(f(a), f)$ have three incomparable least general generalizations:

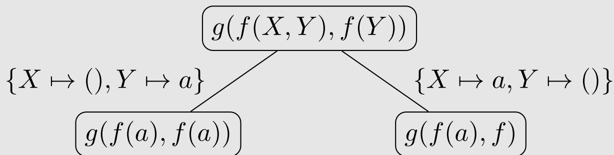


Anti-unification

Variadic anti-unification is finitary.

Example

$g(f(a), f(a))$ and $g(f(a), f)$ have three incomparable least general generalizations:



Rigid variadic generalizations: idea

Restricted variant of variadic anti-unification.

Emphasis on keeping the common structure, rather than on uniform generalization of distinct parts.

Avoiding consecutive sequence variables in the generalization.

Rigid variadic generalizations: idea

More specifically:

Rigid variadic generalizations: idea

More specifically:

- Given two sequences

$$\tilde{s} = (f_1(\tilde{s}_1), \ldots, f_n(\tilde{s}_n)), \qquad \tilde{t} = (g_1(\tilde{t}_1), \ldots, g_m(\tilde{t}_m))$$

Rigid variadic generalizations: idea

More specifically:

- Given two sequences

$$\tilde{s} = (f_1(\tilde{s}_1), \dots, f_n(\tilde{s}_n)), \qquad \tilde{t} = (g_1(\tilde{t}_1), \dots, g_m(\tilde{t}_m))$$

- 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 variadic generalizations: idea

More specifically:

- Given two sequences

$$\tilde{s} = (f_1(\tilde{s}_1), \dots, f_n(\tilde{s}_n)), \qquad \tilde{t} = (g_1(\tilde{t}_1), \dots, g_m(\tilde{t}_m))$$

- 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 $\tilde{s}$ and $\tilde{t}$ is a sequence

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

  where
  - X's are (not necessarily distinct) new sequence variables,
  - Some X's can be omitted,
  - if $h_i = f_j = g_l$, then $\tilde{r}_i$ is a rigid generalization of $\tilde{s}_j$ and $\tilde{r}_l$.

Rigid variadic generalizations: idea

More specifically:

- Given two sequences

$$\tilde{s} = (f_1(\tilde{s}_1), \dots, f_n(\tilde{s}_n)), \qquad \tilde{t} = (g_1(\tilde{t}_1), \dots, g_m(\tilde{t}_m))$$

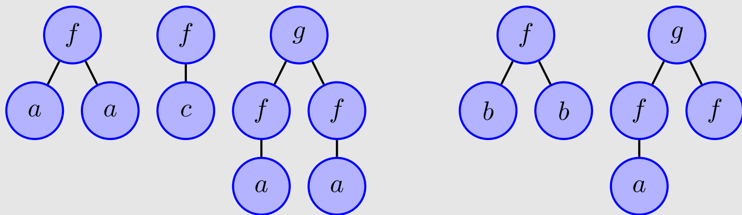
- Take a common subsequence of $f_1, \dots, f_n$ and $g_1, \dots, g_m$. Let it be $h_1, \dots, h_k$.
- Then a rigid generalization of $\tilde{s}$ and $\tilde{t}$ is a sequence

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

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

Computing rigid variadic generalizations

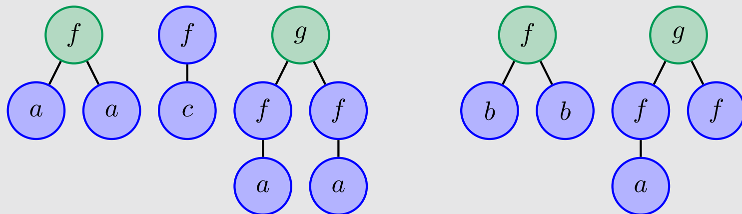
Example



Rigidity function computes longest common subsequences.

Computing rigid variadic generalizations

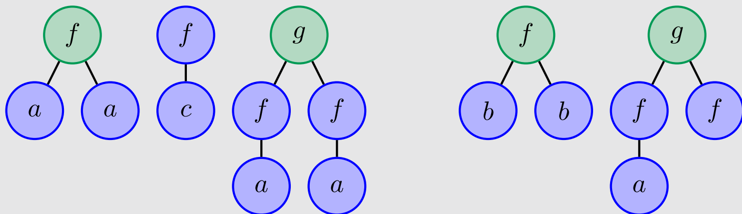
Example



Rigidity function computes longest common subsequences.

Computing rigid variadic generalizations

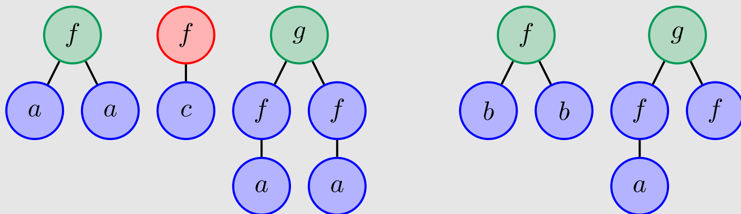
Example



Rigidity function computes longest common subsequences.

Computing rigid variadic generalizations

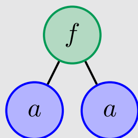
Example

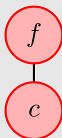


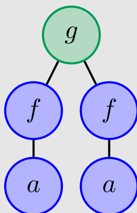
Rigidity function computes longest common subsequences.

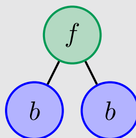
Computing rigid variadic generalizations

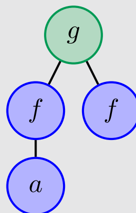
Example







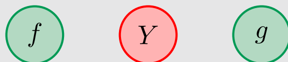


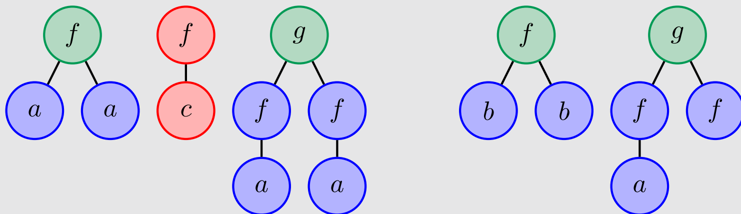


Rigidity function computes longest common subsequences.

Computing rigid variadic generalizations

Example

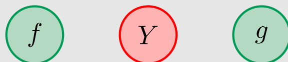


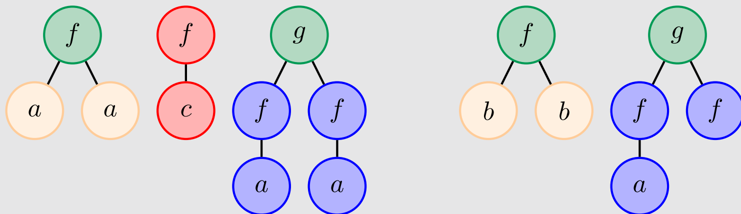


Rigidity function computes longest common subsequences.

Computing rigid variadic generalizations

Example

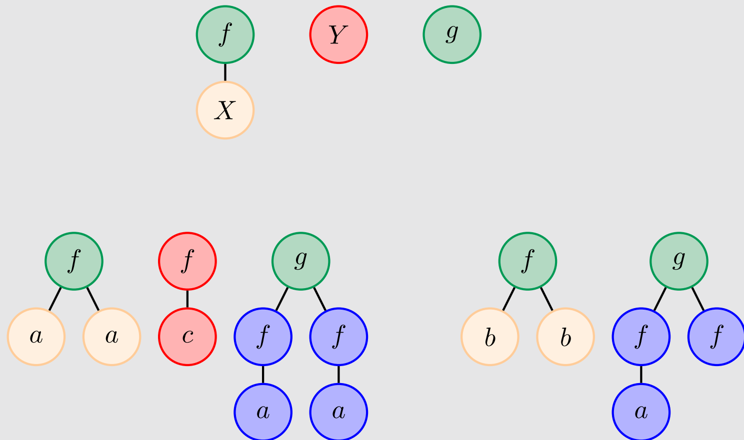




Rigidity function computes longest common subsequences.

Computing rigid variadic generalizations

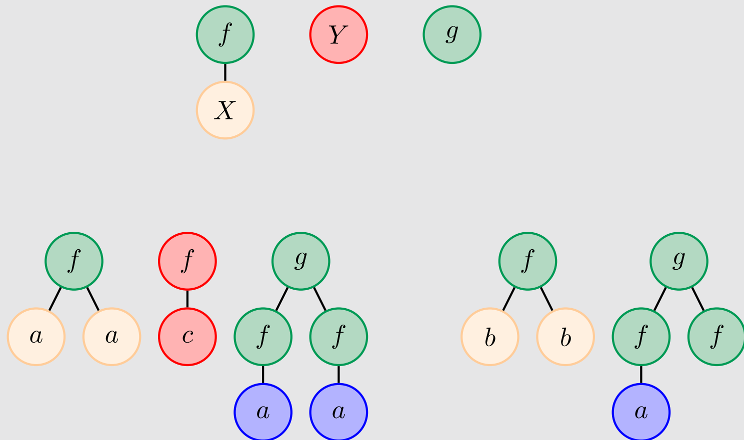
Example



Rigidity function computes longest common subsequences.

Computing rigid variadic generalizations

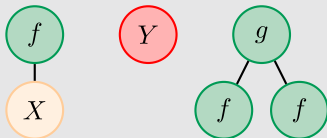
Example

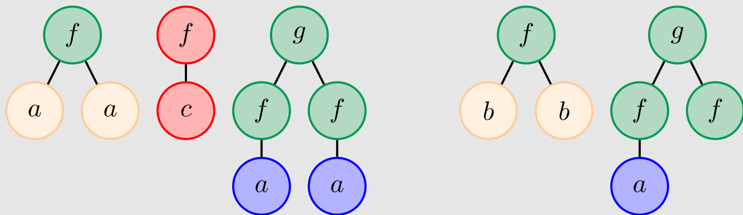


Rigidity function computes longest common subsequences.

Computing rigid variadic generalizations

Example

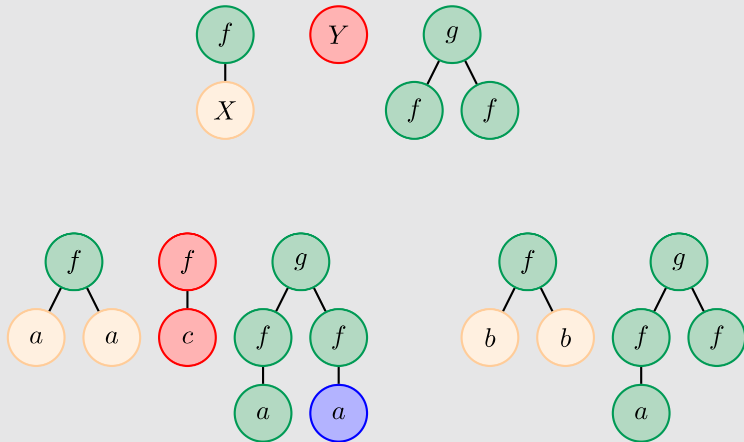




Rigidity function computes longest common subsequences.

Computing rigid variadic generalizations

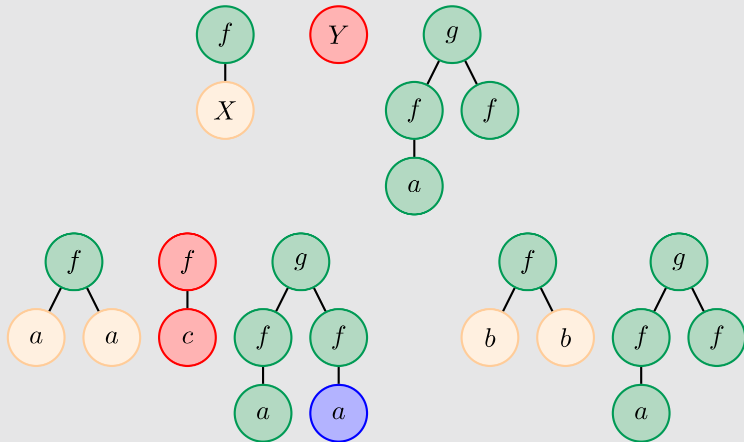
Example



Rigidity function computes longest common subsequences.

Computing rigid variadic generalizations

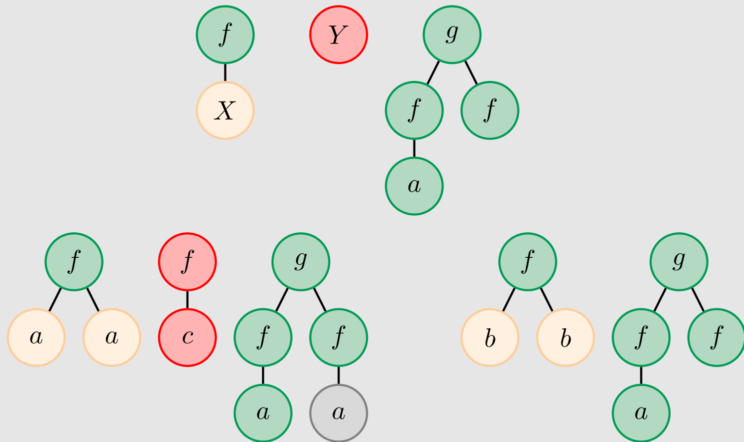
Example



Rigidity function computes longest common subsequences.

Computing rigid variadic generalizations

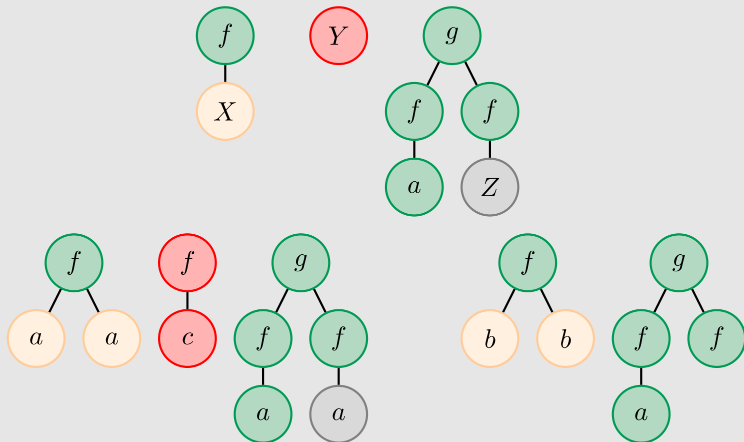
Example



Rigidity function computes longest common subsequences.

Computing rigid variadic generalizations

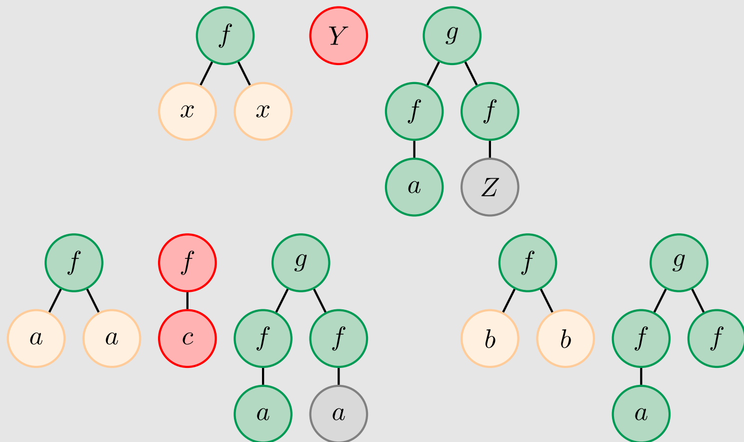
Example



Rigidity function computes longest common subsequences.

Computing rigid variadic generalizations

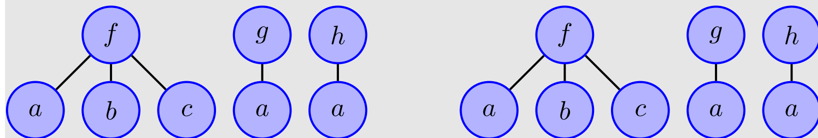
Example



Rigidity function computes longest common subsequences.

Computing rigid variadic generalizations

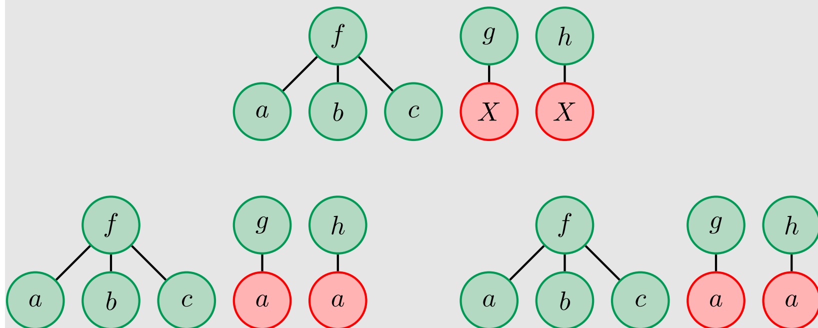
Example



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

Computing rigid variadic generalizations

Example



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

Rigid anti-unification: some interesting facts

By choosing appropriate rigidity functions, rigid variadic 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) variadic anti-unification algorithms can simulate AU anti-unification [Alpuente et al., 2014]

Rigid anti-unification and clone detection

Variadic representation of code pieces:

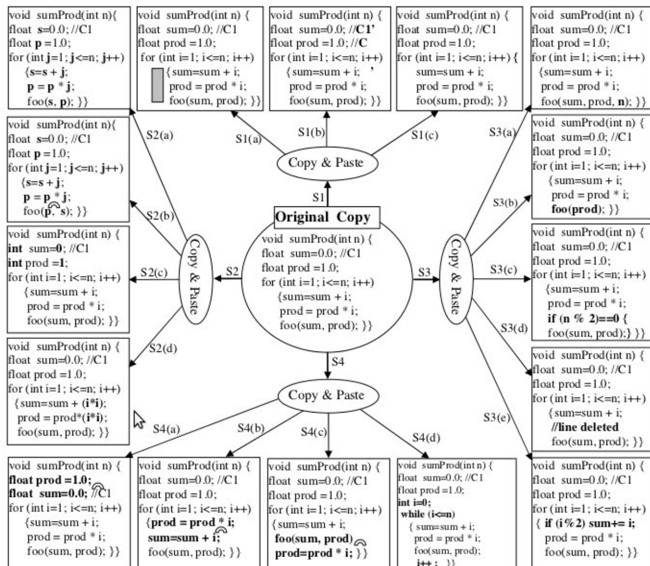
```
if(>=(a, b),
   then(=(c, +(d, b)),
        =(d, +(d, 1))),
   else(=(c, -(d, a))))
```

```
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))))
```

Rigid anti-unification and clone detection



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.

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

- 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.
- Generalizations reflect similarities between two inputs, while the store reflects differences between them.

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.
- Generalizations 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.

Outline

Solving

Computing

Towards logic programming

Bad news: Logic with variadic symbols and sequence variables is not compact.

Counterexample of compactness. An infinite set consisting of:

$$\begin{aligned}
\exists X.&\ \ p(X) \\
&\ \ \neg p \\
\forall x_1.&\ \ \neg p(x_1) \\
\forall x_1, x_2.&\ \ \neg p(x_1, x_2) \\
\forall x_1, x_2, x_3.&\ \ \neg p(x_1, x_2, x_3) \\
\dots&
\end{aligned}$$

Every finite subset of this set has a model, but the entire set does not.

Good news

The clausal fragment behaves well.

Clauses: universally closed disjunctions of literals.

$$\forall x, X, Y, Z.\, p(X, x, Y, f(x), Z) \vee \neg p(X, f(x), Y, x, Z).$$

Good news

The clausal fragment behaves well.

Clauses: universally closed disjunctions of literals.

$$\forall x, X, Y, Z.\, p(X, x, Y, f(x), Z) \vee \neg p(X, f(x), Y, x, Z).$$

Herbrand's theorem holds.

Refutationally complete proof method possible.

Clausal fragment covers many practical cases.

Horn clauses: Clauses with at most one positive literal.

Horn clauses

We focus on Horn clauses.

Resolution rule:

$$\frac{A \vee C \quad \neg B \vee D}{(C \vee D)\sigma}.$$

where A and B are atoms and σ belongs to the minimal complete set of unifiers of A and B: $\sigma \in \text{MCSU}(\{A =^? B\})$.

Resolution is refutationally complete for Horn clause sets.

We should make sure that MCSU is finite at every step.

Guaranteeing finite MCSU's

The MCSU's at each step are finite

- if each literal in each clause in the given set of Horn clauses belongs to the same finitary fragment of unification, or
- if each positive literal occurring in the clauses is linear. No restriction on negative literals (leads to UV-unification problems).

Any other case?

Guaranteeing finite MCSU's

The MCSU's at each step are finite

- if each literal in each clause in the given set of Horn clauses belongs to the same finitary fragment of unification, or
- if each positive literal occurring in the clauses is linear. No restriction on negative literals (leads to UV-unification problems).

Any other case?

- Yes. Well-moded Horn clauses, using matching instead of unification.

Guaranteeing finite MCSU's

The MCSU's at each step are finite

- if each literal in each clause in the given set of Horn clauses belongs to the same finitary fragment of unification, or
- if each positive literal occurring in the clauses is linear. No restriction on negative literals (leads to UV-unification problems).

Any other case?

- Yes. Well-moded Horn clauses, using matching instead of unification.
- It leads us to a special case: sequence transformation rules.

Transformations

Ternary predicate $::\rightarrow$.

Atoms: $::\rightarrow(s, \langle\tilde{l}\rangle, \langle\tilde{r}\rangle)$, where

- $\langle\ \rangle$ is an variadic function symbol.
- $\tilde{l}$ and $\tilde{r}$ are sequences.
- The term s is called a strategy.

Syntactic sugar: $s :: \tilde{l} \rightarrow \tilde{r}$.

Intuition: The strategy s transforms the sequence $\tilde{l}$ into the sequence $\tilde{r}$.

(Conditional) sequence transformation rules: nonnegative Horn clauses in this language.

Queries: negative clauses.

Transformations

We use special notation for rules and queries:

- Rules:

$$
\begin{aligned}
s_0 :: \tilde{l}_0 \to \tilde{r}_0 \Leftarrow \\
\quad & s_1 :: \tilde{l}_1 \to \tilde{r}_1, \\
& \dots \\
& s_n :: \tilde{l}_n \to \tilde{r}_n.
\end{aligned}
$$

- Queries

$$
\begin{aligned}
\Leftarrow\ & s_1 :: \tilde{l}_1 \to \tilde{r}_1, \\
& \dots \\
& s_n :: \tilde{l}_n \to \tilde{r}_n.
\end{aligned}
$$

Language extension

Term and sequence variables are first-order variables.

In matching, they help to explore the term structure "horizontally":

- term variables: to make one step
- sequence variables: to make arbitrary finite number of steps

.

Language extension

Term and sequence variables are first-order variables.

In matching, they help to explore the term structure "horizontally":

- term variables: to make one step
- sequence variables: to make arbitrary finite number of steps

But our terms (trees) have two dimensions.

Would be nice to be able to explore them not only "horizontally", but also "vertically".

Language extension

Second-order variables: for function symbols and for contexts.

In matching, they will help to explore the term structure "vertically":

- function variables: to make one step
- context variables: to make arbitrary finite number of steps

Language extension

Example

Matching problem: $C(F(a,X)) \preceq^? g(f(a,b),h(g(a),f))$.

C: context variable. F: function variable.

Solutions:

$$\sigma_1 = \{C \mapsto g(\circ, h(g(a),f)), F \mapsto f, X \mapsto b\}$$

$$\begin{aligned} C(F(a,X))\sigma_1 &= C\sigma_1[F(a,X)\sigma_1] \\ &= g(\circ, h(g(a),f))[f(a,b)] \\ &= g(f(a,b),h(g(a),f)) \end{aligned}$$

Language extension

Example

Matching problem: $C(F(a,X)) \preceq^{?} g(f(a,b),h(g(a),f))$.

C: context variable. F: function variable.

Solutions:

$$\sigma_2 = \{C \mapsto g(f(a,b),h(\circ,f)), F \mapsto g, X \mapsto ()\}$$

$$\begin{aligned} C(F(a,X))\sigma_2 &= C\sigma_2[F(a,X)\sigma_2] \\ &= g(f(a,b),h(\circ,f))[g(a)] \\ &= g(f(a,b),h(g(a),f)) \end{aligned}$$

Inference system: the ρLog calculus

- Resolution:

$$\frac{\Leftarrow s :: \tilde{l} \to \tilde{r}, Q \qquad s' :: \tilde{l}' \to \tilde{r}' \Leftarrow Body}{(\Leftarrow Body, \mathbf{id} :: \tilde{r}' \to \tilde{r}, Q)\sigma},$$

where $\sigma \in \mathsf{MCSM}(\{s' \preceq^? s, \tilde{l}' \preceq^? \tilde{l}\})$.

Inference system: the ρLog calculus

- Resolution:

$$\frac{\Leftarrow s :: \tilde{l} \to \tilde{r}, Q \qquad s' :: \tilde{l}' \to \tilde{r}' \Leftarrow Body}{(\Leftarrow Body, \mathbf{id} :: \tilde{r}' \to \tilde{r}, Q)\sigma},$$

where $\sigma \in \mathsf{MCSM}(\{s' \preceq^? s, \tilde{l}' \preceq^? \tilde{l}\})$.

- Identity factoring:

$$\frac{\Leftarrow \mathbf{id} :: \tilde{l} \to \tilde{r}, Q}{Q\sigma},$$

where $\sigma \in \mathsf{MCSM}(\{\tilde{r} \preceq^? \tilde{l}\})$.

Inference system: the ρLog calculus

- Resolution:

$$\frac{\Leftarrow s :: \tilde{l} \to \tilde{r}, Q \qquad s' :: \tilde{l}' \to \tilde{r}' \Leftarrow Body}{(\Leftarrow Body, \mathbf{id} :: \tilde{r}' \to \tilde{r}, Q)\sigma},$$

  where $\sigma \in \mathsf{MCSM}(\{s' \preceq^? s, \tilde{l}' \preceq^? \tilde{l}\})$.

- Identity factoring:

$$\frac{\Leftarrow \mathbf{id} :: \tilde{l} \to \tilde{r}, Q}{Q\sigma},$$

  where $\sigma \in \mathsf{MCSM}(\{\tilde{r} \preceq^? \tilde{l}\})$.

- Resolution + identity factoring is refutationally complete for conditional sequence transformations.
- A special restriction on variable occurrences in clauses (well-modedness) guarantees that at each step there is a matching problem (and not unification).

Simple ρLog programs: duplicate merging

Program clause for finding duplicated elements in a sequence and removing one of them:

$$\mathrm{merge_duplicates} :: (X,\, x,\, Y,\, x,\, Z) \Longrightarrow (X,\, x,\, Y,\, Z).$$

Query: merge duplicates in $(a,\, b,\, c,\, b,\, a)$:

$$\mathrm{merge_duplicates} :: (a,\, b,\, c,\, b,\, a) \Longrightarrow X.$$

ρLog returns two answer substitutions:

$$\{X \mapsto (a,\, b,\, c,\, b)\}$$
$$\{X \mapsto (a,\, b,\, c,\, a)\}$$

Simple ρLog programs: duplicate merging

Program clause for merging all duplicated elements in a sequence:

$$\text{merge_all_duplicates} :: X \Longrightarrow Y \ \leftarrow$$
$$\quad \mathbf{nf}(\text{merge_duplicates}) :: X \Longrightarrow Y.$$

Computes in Y a normal form of X wrt merge_duplicates.

$\mathbf{nf}$: a builtin strategy for normal form computation.

Simple ρLog programs: duplicate merging

Program clause for merging all duplicated elements in a sequence:

$$\text{merge_all_duplicates} :: X \Longrightarrow Y \ \leftarrow$$
$$\quad \mathbf{nf}(\text{merge_duplicates}) :: X \Longrightarrow Y.$$

Computes in Y a normal form of X wrt merge_duplicates.

$\mathbf{nf}$: a builtin strategy for normal form computation.

Query: merge all duplicates in (a, b, c, b, a):

$$\text{merge_all_duplicates} :: (a, b, c, b, a) \Longrightarrow X.$$

Answer: $\{X \mapsto (a, b, c)\}$.

Simple ρLog programs: duplicate merging

It can happen that the same normal form is computed multiple times, or there exist multiple normal forms.

If we want only one answer, we can use the builtin strategy $\mathbf{first_one}$. It stops the computation after the first applicable strategy computes one answer.

$$\begin{aligned}&\mathrm{merge_all_duplicates} :: X \Longrightarrow Y \ \leftarrow\\&\qquad \mathbf{first_one}(\mathbf{nf}(\mathrm{merge_duplicates})) :: X \Longrightarrow Y.\end{aligned}$$

Simple ρLog programs: duplicate merging

It can happen that the same normal form is computed multiple times, or there exist multiple normal forms.

If we want only one answer, we can use the builtin strategy $\mathbf{first_one}$. It stops the computation after the first applicable strategy computes one answer.

$$\begin{aligned}&\mathrm{merge_all_duplicates} :: X \Longrightarrow Y \ \leftarrow \\ &\qquad \mathbf{first_one}(\mathbf{nf}(\mathrm{merge_duplicates})) :: X \Longrightarrow Y.\end{aligned}$$

Short notation for such clauses:

$$\mathrm{merge_all_duplicates} ::= \mathbf{first_one}(\mathbf{nf}(\mathrm{merge_duplicates})).$$

Simple ρLog programs: rewriting

Rewriting a term using some rule(s), one step:

$$\mathrm{rewrite}(x) :: C(y) \Longrightarrow C(z) \ \leftarrow x :: y \Longrightarrow z.$$

Specifying some rewrite rules:

$$\mathrm{r} :: f(x) \Longrightarrow g(x).$$

$$\mathrm{r} :: f(f(x)) \Longrightarrow x.$$

Compute a normal form of $h(f(f(a)), f(a))$ with respect to r. It should contain a subterm starting from g. Query:

$$\mathbf{nf}(\mathrm{rewrite}(\mathrm{r})) :: h(f(f(a)), f(a)) \Longrightarrow C(g(X)).$$

A single answer:

$$\{C \mapsto h(\circ, f(a)), X \mapsto a\}.$$

ρLog strategies

Rules are elementary strategies.

More complex strategies can be obtained by combining strategies.

Besides $\mathbf{first_one}$, $\mathbf{nf}$, and $\mathbf{id}$, ρLog comes with other builtin strategies and strategy combinators. Among them:

- $\mathbf{compose}$: for composing strategies
- $\mathbf{choose}$: for choosing a strategy among alternatives
- $\mathbf{map}$: for mapping a strategy on a sequence

Rule-based programming in Mathematica

Wolfram (the programming language of the software system Mathematica) is based on variadic terms with term, sequence, and function variables.

Slightly different syntax:

- Square brackets instead of round ones.
- Terms are allowed in the functional position: e.g., $f[a][g[b], 5, "c"]$ is a valid term.
- Function variables can be instantiated by terms in the functional position. Therefore, they syntactically do not differ from term variables.
- Two kinds of sequence variables: for arbitrary sequences and for nonempty sequences.

Variadic equational matching is permitted.

Simple Wolfram programs

Simple programs involving sequence patterns.

A function which picks out pairs of duplicated elements in `h`:

```
h[a___, x_, b___, x_, c___] :=
    Sequence[{x}, h[a, b, c]]
```

Picking out the two paired elements from `h[2,3,2,4,5,3]`

```
In[2]:=   h[2,3,2,4,5,3]
Out[2]:=  Sequence[{2}, {3}, h[4, 5]]
```

Simple Wolfram programs

Simple programs involving sequence patterns.

```
Clear[BubbleSort]
BubbleSort[Order_][{a___, x_, y_, b___}] /;
    !Order[x, y] :=
   BubbleSort[Order][{a, y, x, b}]
BubbleSort[_][sorted_List] := sorted
```

```
In[2]:=   BubbleSort[Greater][{3, 4, 2, 1, 5, 6}]
Out[2]:=  {6, 5, 4, 3, 2, 1}

In[3]:=   BubbleSort[Less][{3, 4, 2, 1, 5, 6}]
Out[3]:=  {1, 2, 3, 4, 5, 6}
```

Some useful links

Library of unification and anti-unification algorithms:

`www.risc.jku.at/projects/stout/library.html`

Among others, contains variadic unification, matching, and anti-unification algorithms (including extensions of variadic anti-unification a second order case and to term-graphs).

ρLog implementations:

- PρLog, an extension of Prolog:
  `www.risc.jku.at/people/tkutsia/software/prholog/`
- In wolfram (Mathematica):
  `staff.fmi.uvt.ro/~mircea.marin/rholog/`