International School on Rewriting (ISR 2012)

in the Alan Turing Year

MUG: Matching, Unification, Generalizations Part 1

Temur Kutsia
Research Institute for Symbolic Computation (RISC)
Johannes Kepler University Linz, Austria

Overview

Part 1

Syntactic unification and matching

Part 2

Equational unification and matching

Overview

Part 1

Syntactic unification and matching

Part 2

Equational unification and matching

Rewriting Requires Matching

- Rewrite $s(0) + s(s(0))$ by the rule $s(x) + y \to s(x + y)$.
- Match $s(x) + y$ to $s(0) + s(s(0))$.

Completion Requires Unification

- Compute a critical pair between $f(f(x,y),z) \to f(x,f(y,z))$ and $h(f(x_1,y_1)) \to f(h(x_1),h(y_1))$.
- Unify $f(f(x,y),z)$ and $f(x_1,y_1)$.

Rewriting Modulo Equalities Requires E-Matching

- Rewrite $f(a, f(b, e))$ by the rule $f(e, x) \to x$.
- f is commutative.
- Match $f(e, x)$ to $f(b, e)$ modulo commutativity of f.

Completion Modulo Equalities Requires E-Unification

- Compute a critical pair between $x \cdot (x^- \cdot z) \to 1$ and $(y_0 \cdot x_0)^- \to (x_0)^- \cdot (y_0)^-$
- $\cdot$ is associative and commutative.
- AC-unify $x \cdot (x^- \cdot z)$ and $y_0 \cdot x_0$.

Solving Term Equations

- Unification/matching problems: problems of solving equations between terms.
- Used in
  - rewriting
  - automated reasoning
  - logic and functional programming
  - type inference
  - program transformation
  - computational linguistics
  - ...

Subject of this Course

- Syntactic unification and matching.
- Generalizations to the equational case.
- MUG: Matching, Unification, Generalizations.

Notation

- First-order language.
- $\mathcal{F}$: Set of function symbols.
- $\mathcal{V}$: Set of variables.
- x, y, z: Variables.
- a, b, c: Constants.
- f, g, h: Arbitrary function symbols.
- s, t, r: Terms.
- $\mathcal{T}(\mathcal{F}, \mathcal{V})$: Set of terms over $\mathcal{F}$ and $\mathcal{V}$.
- Equation: a pair of terms, written $s \doteq t$.
- $vars(t)$: The set of variables in t. This notation will be used also for sets of terms, equations, and sets of equations.

Substitutions

Substitution

- A mapping from variables to terms, where all but finitely many variables are mapped to themselves.

Example

A substitution is represented as a set of *bindings*:

- $\{x \mapsto f(a,b), y \mapsto z\}$.
- $\{x \mapsto f(x,y), y \mapsto f(x,y)\}$.

All variables except x and y are mapped to themselves by these substitutions.

Substitutions

Notation

- σ, ϑ, η, ρ denote arbitrary substitutions.
- ε denotes the identity substitution.

Substitutions

Substitution Application

Applying a substitution σ to a term t:

$$t\sigma = \begin{cases} \sigma(x) & \text{if } t = x \\ f(t_1\sigma, \dots, t_n\sigma) & \text{if } t = f(t_1, \dots, t_n) \end{cases}$$

Example

- $\sigma = \{x \mapsto f(x, y), y \mapsto g(a)\}$.
- $t = f(x, g(f(x, f(y, z))))$.
- $t\sigma = f(f(x, y), g(f(f(x, y), f(g(a), z))))$.

Substitutions

Domain, Range, Variable Range

For a substitution σ:

- The *domain* is the set of variables:

$$dom(\sigma) = \{x \mid x\sigma \neq x\}.$$

- The *range* is the set of terms:

$$ran(\sigma) = \bigcup_{x \in dom(\sigma)} \{x\sigma\}.$$

- The *variable range* is the set of variables:

$$vran(\sigma) = vars(ran(\sigma)).$$

Substitutions

Example (Domain, Range, Variable Range)

$$
\begin{aligned}
dom(\{x \mapsto f(a,y), y \mapsto g(z)\}) &= \{x, y\}\\
ran(\{x \mapsto f(a,y), y \mapsto g(z)\}) &= \{f(a,y), g(z)\}\\
vran(\{x \mapsto f(a,y), y \mapsto g(z)\}) &= \{y, z\}
\end{aligned}
$$

$$dom(\varepsilon) = ran(\varepsilon) = vran(\varepsilon) = \varnothing$$

Substitutions

Restriction

Restriction of a substitution σ on a set of variables $\mathcal{X}$:
A substitution $\sigma|_{\mathcal{X}}$ such that for all x

$$x\sigma|_{\mathcal{X}} = \begin{cases} x\sigma & \text{if } x \in \mathcal{X} \\ x & \text{otherwise} \end{cases}$$

Example

- $\{x \mapsto f(a), y \mapsto x, z \mapsto b\}|_{\{x,y\}} = \{x \mapsto f(a), y \mapsto x\}$.
- $\{x \mapsto f(a), z \mapsto b\}|_{\{x,y\}} = \{x \mapsto f(a)\}$.
- $\{z \mapsto b\}|_{\{x,y\}} = \varepsilon$.

Substitutions

Composition of Substitutions

- Written: $\sigma\vartheta$.
- $t(\sigma\vartheta) = (t\sigma)\vartheta$.

Example

- $\sigma = \{x \mapsto f(y), y \mapsto z\}$
- $\vartheta = \{x \mapsto a, y \mapsto b, z \mapsto y\}$
- $\sigma\vartheta = \{x \mapsto f(b), z \mapsto y\}$

Composition is associative but not commutative:

$$\vartheta\sigma = \{x \mapsto a, y \mapsto b\} \neq \sigma\vartheta.$$

Substitutions

Triangular Form

Sequential list of bindings:

$$[x_1 \mapsto t_1; x_2 \mapsto t_2; \ldots; x_n \mapsto t_n],$$

represents composition of n substitutions each consisting of a single binding:

$$\{x_1 \mapsto t_1\}\{x_2 \mapsto t_2\} \ldots \{x_n \mapsto t_n\}.$$

Substitutions

Variable Renaming

A substitution $\sigma = \{x_1 \mapsto y_1, x_2 \mapsto y_2, \ldots, x_n \mapsto y_n\}$ is called *variable renaming* iff

- y's are distinct variables, and
- $\{x_1, \ldots, x_n\} = \{y_1, \ldots, y_n\}$.

(Permuting the domain variables.)

Example

- $\{x \mapsto y, y \mapsto z, z \mapsto x\}$ is a variable renaming.
- $\{x \mapsto a\}$, $\{x \mapsto y\}$, and $\{x \mapsto z, y \mapsto z\}$ are not.

Substitutions

Idempotent Substitution

A substitution σ is *idempotent* iff $\sigma\sigma = \sigma$.

Example

Let $\sigma = \{x \mapsto f(z), y \mapsto z\}$, $\vartheta = \{x \mapsto f(y), y \mapsto z\}$.

- σ is idempotent.
- ϑ is not: $\vartheta\vartheta = \sigma \neq \vartheta$.

Substitutions

Theorem

σ is idempotent iff $dom(\sigma) \cap vran(\sigma) = \emptyset$.

Proof.

Exercise. □

Substitutions

Instantiation Quasi-Ordering

- A substitution σ is *more general* than ϑ, written $\sigma \lessdot \vartheta$, if there exists η such that $\sigma\eta = \vartheta$.
- The relation $\lessdot$ is quasi-ordering (reflexive and transitive binary relation), called *instantiation quasi-ordering.*
- $\doteq$ is the equivalence relation corresponding to $\lessdot$.

Example

Let $\sigma = \{x \mapsto y\}$, $\rho = \{x \mapsto a, y \mapsto a\}$, $\vartheta = \{y \mapsto x\}$.

- $\sigma \lessdot \rho$, because $\sigma\{y \mapsto a\} = \rho$.
- $\sigma \lessdot \vartheta$, because $\sigma\{y \mapsto x\} = \vartheta$.
- $\vartheta \lessdot \sigma$, because $\vartheta\{x \mapsto y\} = \sigma$.
- $\sigma \doteq \vartheta$.

Substitutions

Theorem

For any σ and ϑ, $\sigma \doteq \vartheta$ iff there exists a variable renaming substitution η such that $\sigma\eta = \vartheta$.

Proof.

Exercise. □

Example

σ, ϑ from the previous example:

- $\sigma = \{x \mapsto y\}$.
- $\vartheta = \{y \mapsto x\}$.
- $\sigma \doteq \vartheta$.
- $\sigma\{x \mapsto y, y \mapsto x\} = \vartheta$.

Substitutions

Unifier, Most General Unifier, Unification Problem

- A substitution σ is a *unifier* of the terms s and t if $s\sigma = t\sigma$.
- A unifier σ of s and t is a *most general unifier (mgu)* if $\sigma \leq \vartheta$ for every unifier ϑ of s and t.
- A *unification problem* for s and t is represented as $s \doteq^{?} t$.

Substitutions

Example (Unifier, Most General Unifier)

Unification problem: $f(x, z) \doteq^? f(y, g(a))$.

- Some of the unifiers:

$$\{x \mapsto y, z \mapsto g(a)\}$$
$$\{y \mapsto x, z \mapsto g(a)\}$$
$$\{x \mapsto a, y \mapsto a, z \mapsto g(a)\}$$
$$\{x \mapsto f(x, y), y \mapsto f(x, y), z \mapsto g(a)\}$$
$$\dots$$

- mgu's: $\{x \mapsto y, z \mapsto g(a)\}$, $\{y \mapsto x, z \mapsto g(a)\}$.
- mgu is unique up to a variable renaming:

$$\{x \mapsto y, z \mapsto g(a)\} \simeq \{y \mapsto x, z \mapsto g(a)\}$$

Unification Algorithm

- Goal: Design an algorithm that for a given unification problem $s \doteq^? t$
  - returns an mgu of s and t if they are unifiable,
  - reports failure otherwise.

Naive Algorithm

Write down two terms and set markers at the beginning of the terms. Then:

1. Move the markers simultaneously, one symbol at a time, until both move off the end of the term (**success**), or until they point to two different symbols;
2. If the two symbols are both non-variables, then **fail**; otherwise, one is a variable (call it x) and the other one is the first symbol of a subterm (call it t):
   - If x occurs in t, then **fail**;
   - Otherwise, replace x everywhere from the marker positions by t (including in the solution), write down "$x \mapsto t$" as a part of the solution, and return to 1.

Naive Algorithm

- Finds disagreements in the two terms to be unified.
- Attempts to repair the disagreements by binding variables to terms.
- Fails when function symbols clash, or when an attempt is made to unify a variable with a term containing that variable.

Interesting Questions

Correctness:

- Does the algorithm always terminate?
- Does it always produce an mgu for two unifiable terms, and fail for non-unifiable terms?
- Do these answers depend on the order of operations?

Implementation:

- What data structures should be used for terms and substitutions?
- How should application of a substitution be implemented?
- What order should the operations be performed in?

Complexity:

- How much space does this take, and how much time?

Answers

On the coming slides.

Rule-Based Formulation of Unification

- Unification algorithm in a rule-base way.
- Repeated transformation of a set of equations.
- The left-to-right search for disagreements: modeled by term decomposition.

The Inference System $\mathcal{U}$

- A set of equations in *solved form*:

  $$\{x_1 \doteq t_1, \ldots, x_n \doteq t_n\}$$

  where each x_i occurs exactly once.
- For each idempotent substitution there exists exactly one set of equations in solved form.
- Notation:
  - $[\sigma]$ for the solved form set for an idempotent substitution σ.
  - σ_S for the idempotent substitution corresponding to a solved form set S.

The Inference System $\mathcal{U}$

- *System*: The symbol $\perp$ or a pair $P; S$ where
  - P is a set of unification problems,
  - S is a set of equations in solved form.
- $\perp$ represents failure.
- A unifier (or a solution) of a system $P; S$: A substitution that unifies each of the equations in P and S.
- $\perp$ has no unifiers.

The Inference System $\mathcal{U}$

Example

- System: $\{g(a) \doteq^? g(y), g(z) \doteq^? g(g(x))\}; \{x \doteq g(y)\}$.
- Its unifier: $\{x \mapsto g(a), y \mapsto a, z \mapsto g(g(a))\}$.

The Inference System $\mathcal{U}$

Six transformation rules on systems:[1]

Trivial: $\{s \doteq^? s\} \uplus P'; S \Longrightarrow P'; S.$

Decomposition: $\{f(s_1, \ldots, s_n) \doteq^? f(t_1, \ldots, t_n)\} \uplus P'; S \Longrightarrow$
$\{s_1 \doteq^? t_1, \ldots, s_n \doteq^? t_n\} \cup P'; S,$
where $n \geq 0$.

Symbol Clash: $\{f(s_1, \ldots, s_n) \doteq^? g(t_1, \ldots, t_m)\} \uplus P'; S \Longrightarrow \bot$
if $f \neq g$.

[1] $\uplus$ stands for disjoint union.

The Inference System $\mathcal{U}$

Orient: $\{t \doteq^? x\} \cup P'; S \Longrightarrow \{x \doteq^? t\} \uplus P'; S$,
if t is not a variable.

Occurs Check: $\{x \doteq^? t\} \uplus P'; S \Longrightarrow \bot$
if $x \in vars(t)$ but $x \neq t$.

Variable Elimination: $\{x \doteq^? t\} \uplus P'; S \Longrightarrow$
$P'\{x \mapsto t\}; S\{x \mapsto t\} \cup \{x \doteq t\}$,
if $x \notin vars(t)$.

Unification with $\mathcal{U}$

In order to unify s and t:

1. Create an initial system $\{s \doteq^? t\}; \varnothing$.
2. Apply successively rules from $\mathcal{U}$.

The system $\mathcal{U}$ is essentially the Herbrand's Unification Algorithm.

Properties of $\mathcal{U}$: Termination

Lemma

For any finite set of equations P, every sequence of transformations in $\mathcal{U}$

$$P; \varnothing \Longrightarrow P_1; S_1 \Longrightarrow P_2; S_2 \Longrightarrow \cdots$$

terminates either with $\perp$ or with $\varnothing; S$, with S in solved form.

Properties of $\mathcal{U}$: Termination

Proof.

Complexity measure on the sets of equations: $\langle n_1, n_2, n_3 \rangle$, ordered lexicographically on triples of naturals, where

n_1 = The number of distinct variables in P.

n_2 = The number of symbols in P.

n_3 = The number of equations in P of the form $t \doteq^? x$ where t is not a variable.

Properties of $\mathcal{U}$: Termination

Proof [Cont.]

Each rule in $\mathcal{U}$ strictly reduces the complexity measure.

Rule	n_1	n_2	n_3
Trivial	$\geq$	$>$	
Decomposition	$=$	$>$	
Orient	$=$	$=$	$>$
Variable Elimination	$>$		

Properties of $\mathcal{U}$: Termination

Proof [Cont.]

- A rule can always be applied to a system with non-empty P.
- The only systems to which no rule can be applied are $\perp$ and $\varnothing; S$.
- Whenever an equation is added to S, the variable on the left-hand side is eliminated from the rest of the system, i.e. $S_1, S_2, \ldots$ are in solved form.

$\square$

Corollary

If $P; \varnothing \Longrightarrow^{+} \varnothing; S$ *then* σ_S *is idempotent.*

Properties of $\mathcal{U}$: Correctness

Notation: Γ for systems.

Lemma

For any transformation $P; S \Longrightarrow \Gamma$, a substitution ϑ unifies $P; S$ iff it unifies Γ.

Properties of $\mathcal{U}$: Correctness

Proof.

Occurs Check: If $x \in vars(t)$ and $x \neq t$, then

- x contains fewer symbols than t,
- $x\vartheta$ contains fewer symbols than $t\vartheta$ (for any ϑ).

Therefore, $x\vartheta$ and $t\vartheta$ can not be unified.

Variable Elimination: From $x\vartheta = t\vartheta$, by structural induction on u:

$$u\vartheta = u\{x \mapsto t\}\vartheta$$

for any term, equation, or set of equations u. Then

$$P'\vartheta = P'\{x \mapsto t\}\vartheta, \qquad S'\vartheta = S'\{x \mapsto t\}\vartheta.$$

□

Properties of $\mathcal{U}$: Correctness

Theorem (Soundness)

If $P; \varnothing \Longrightarrow^{+} \varnothing; S$, *then* σ_S *unifies any equation in* P.

Proof.

By induction on the length of derivation, using the previous lemma and the fact that σ_S unifies S. □

Properties of $\mathcal{U}$: Correctness

Theorem (Completeness)

If ϑ unifies every equation in P, then any maximal sequence of transformations $P; \varnothing \Longrightarrow \cdots$ ends in a system $\varnothing; S$ such that $\sigma_S \leq \vartheta$.

Proof.

Such a sequence must end in $\varnothing; S$ where ϑ unifies S (why?).
For every binding $x \mapsto t$ in σ_S, $x\sigma_S\vartheta = t\vartheta = x\vartheta$ and for every $x \notin dom(\sigma_S)$, $x\sigma_S\vartheta = x\vartheta$. Hence, $\vartheta = \sigma_S\vartheta$. □

Corollary

If P has no unifiers, then any maximal sequence of transformations from $P; \varnothing$ must have the form $P; \varnothing \Longrightarrow \cdots \Longrightarrow \bot$.

Observations

- $\mathcal{U}$ computes an idempotent mgu.
- The choice of rules in computations via $\mathcal{U}$ is "don't care" nondeterminism (the word "any" in Completeness Theorem).
- Any control strategy will result to an mgu for unifiable terms, and failure for non-unifiable terms.
- Any practical algorithm that proceeds by performing transformations of $\mathcal{U}$ in any order is
  - sound and complete,
  - generates mgus for unifiable terms.
- Not all transformation sequences have the same length.
- Not all transformation sequences end in exactly the same mgu.

Matching

Matcher, Matching Problem

- A substitution σ is a *matcher* of s to t if $s\sigma = t$.
- A matching problem between s and t is represented as $s \ll^? t$.

Matching vs Unification

Example

$f(x,y) \ll^? f(g(z),c)$	$f(x,y) \doteq^? f(g(z),c)$
$\{x \mapsto g(z), y \mapsto c\}$	$\{x \mapsto g(z), y \mapsto c\}$
$f(x,y) \ll^? f(g(z),x)$	$f(x,y) \doteq^? f(g(z),x)$
$\{x \mapsto g(z), y \mapsto x\}$	$\{x \mapsto g(z), y \mapsto g(z)\}$
$f(x,a) \ll^? f(b,y)$	$f(x,a) \doteq^? f(b,y)$
No matcher	$\{x \mapsto b, y \mapsto a\}$
$f(x,x) \ll^? f(x,a)$	$f(x,x) \doteq^? f(x,a)$
No matcher	$\{x \mapsto a\}$
$x \ll^? f(x)$	$x \doteq^? f(x)$
$\{x \mapsto f(x)\}$	No unifier

How to Solve Matching Problems

- $s \doteq^? t$ and $s \ll^? t$ coincide, if t is ground.
- When t is not ground in $s \ll^? t$, simply regard all variables in t as constants and use the unification algorithm.
- Alternatively, modify the rules in $\mathcal{U}$ to work directly with the matching problem.

Matched Form

- A set of equations $\{x_1 \ll t_1, \ldots, x_n \ll t_n\}$ is in matched from, if all x's are pairwise distinct.
- The notation σ_S extends to matched forms.
- If S is in matched form, then

$$\sigma_S(x) = \begin{cases} t, & \text{if } x \ll t \in S \\ x, & \text{otherwise} \end{cases}$$

The Inference System $\mathcal{M}$

- *Matching system*: The symbol $\perp$ or a pair $P; S$, where
  - P is set of matching problems.
  - S is set of equations in matched form.
- A matcher (or a solution) of a system $P; S$: A substitution that solves each of the matching equations in P and S.
- $\perp$ has no matchers.

The Inference System $\mathcal{M}$

Five transformation rules on matching systems:[2]

Decomposition: $\{f(s_1,\ldots,s_n) \ll^? f(t_1,\ldots,t_n)\} \uplus P'; S \Longrightarrow$
$\{s_1 \ll^? t_1,\ldots,s_n \ll^? t_n\} \cup P'; S,$
where $n \geq 0$.

Symbol Clash: $\{f(s_1,\ldots,s_n) \ll^? g(t_1,\ldots,t_m)\} \uplus P'; S \Longrightarrow \bot,$
if $f \neq g$.

[2] $\uplus$ stands for disjoint union.

The Inference System $\mathcal{M}$

Symbol-Variable Clash: $\{f(s_1, \ldots, s_n) \ll^? x\} \uplus P'; S \Longrightarrow \perp$

Merging Clash: $\{x \ll^? t_1\} \uplus P'; \{x \ll t_2\} \uplus S' \Longrightarrow \perp,$
if $t_1 \neq t_2$.

Elimination: $\{x \ll^? t\} \uplus P'; S \Longrightarrow P'; \{x \ll t\} \cup S,$
if S does not contain $x \ll t'$ with $t \neq t'$.

Matching with $\mathcal{M}$

In order to match s to t

1. Create an initial system $\{s \ll^? t\}; \varnothing$.
2. Apply successively the rules from $\mathcal{M}$.

Matching with $\mathcal{M}$

Example

Match $f(x, f(a, x))$ to $f(g(a), f(a, g(a)))$:

$$\{f(x, f(a, x)) \ll^? f(g(a), f(a, g(a)))\}; \varnothing \Longrightarrow_{\text{Decomposition}}$$
$$\{x \ll^? g(a), f(a, x) \ll^? f(a, g(a))\}; \varnothing \Longrightarrow_{\text{Elimination}}$$
$$\{f(a, x) \ll^? f(a, g(a))\}; \{x \ll g(a)\} \Longrightarrow_{\text{Decomposition}}$$
$$\{a \ll^? a, x \ll^? g(a)\}; \{x \ll g(a)\} \Longrightarrow_{\text{Decomposition}}$$
$$\{x \ll^? g(a)\}; \{x \ll g(a)\} \Longrightarrow_{\text{Merge}}$$
$$\varnothing; \{x \ll g(a)\}$$

Matcher: $\{x \mapsto g(a)\}$.

Matching with $\mathcal{M}$

Example

Match $f(x,x)$ to $f(x,a)$:

$$\{f(x,x) \ll^? f(x,a)\}; \varnothing \Longrightarrow_{\text{Decomposition}}$$
$$\{x \ll^? x, x \ll^? a\}; \varnothing \Longrightarrow_{\text{Elimination}}$$
$$\{x \ll^? a\}; \{x \ll x\} \Longrightarrow_{\text{Merging Clash}}$$
$$\perp$$

No matcher.

Properties of $\mathcal{M}$: Termination

Theorem

For any finite set of matching problems P, every sequence of transformations in $\mathcal{M}$ of the form $P; \varnothing \Longrightarrow P_1; S_1 \Longrightarrow P_2; S_2 \Longrightarrow \cdots$ terminates either with $\bot$ or with $\varnothing; S$, with S in matched form.

Proof.

- Termination is obvious, since every rule strictly decreases the size of the first component of the matching system.
- A rule can always be applied to a system with non-empty P.
- The only systems to which no rule can be applied are $\bot$ and $\varnothing; S$.
- Whenever $x \ll t$ is added to S, there is no other equation $x \ll t'$ in S. Hence, $S_1, S_2, \ldots$ are in matched form.

$\square$

Properties of $\mathcal{M}$: Correctness

The following lemma is straightforward:

Lemma

For any transformation of matching systems $P; S \Longrightarrow \Gamma$, *a substitution* ϑ *is a matcher for* $P; S$ *iff it is a matcher for* Γ.

Properties of $\mathcal{M}$: Correctness

Theorem (Soundness)

If $P; \varnothing \Longrightarrow^{+} \varnothing; S$*, then* σ_S *solves all matching equations in* P*.*

Proof.

By induction on the length of derivations, using the previous lemma and the fact that σ_S solves the matching problems in S. $\square$

Properties of $\mathcal{M}$: Correctness

Let $v(\{s_1 \ll t_1, \ldots, s_n \ll t_n\})$ be $vars(\{s_1, \ldots, s_n\})$.

Theorem (Completeness)

If ϑ is a matcher of P, then any maximal sequence of transformations $P; \varnothing \Longrightarrow \cdots$ ends in a system $\varnothing; S$ such that $\sigma_S = \vartheta|_{v(P)}$.

Proof.

Such a sequence must end in $\varnothing; S$ where ϑ is a matcher of S. $v(S) = v(P)$. For every equation $x \ll t \in S$, either $t = x$ or $x \mapsto t \in \sigma_S$. Therefore, for any such x, $x\sigma_S = t = x\vartheta$. Hence, $\sigma_S = \vartheta|_{v(P)}$. □

Corollary

If P has no matchers, then any maximal sequence of transformations from $P; \varnothing$ must have the form $P; \varnothing \Longrightarrow \cdots \Longrightarrow \bot$.

Improving the Unification Algorithm

Back to unification.

Unification via $\mathcal{U}$: Exponential in Time and Space

Example

Unifying s and t, where

$$s = h(x_1, x_2, \ldots, x_n, f(y_0, y_0), f(y_1, y_1), \ldots, f(y_{n-1}, y_{n-1}), y_n)$$
$$t = h(f(x_0, x_0), f(x_1, x_1), \ldots, f(x_{n-1}, x_{n-1}), y_1, y_2, \ldots, y_n, x_n)$$

will create an mgu where each x_i and each y_i is bound to a term with $2^{i+1} - 1$ symbols:

$$\{x_1 \mapsto f(x_0, x_0), x_2 \mapsto f(f(x_0, x_0), f(x_0, x_0)), \ldots,$$
$$y_0 \mapsto x_0, y_1 \mapsto f(x_0, x_0), y_2 \mapsto f(f(x_0, x_0), f(x_0, x_0)), \ldots\}$$

Can we do better?

Unification via $\mathcal{U}$: Exponential in Time and Space

First idea: Use triangular substitutions.

Example

Triangular unifier of s and t from the previous example:

$$[y_0 \mapsto x_0; y_n \mapsto f(y_{n-1}, y_{n-1}); y_{n-1} \mapsto f(y_{n-2}, y_{n-2}); \ldots]$$

- Triangular unifiers are not larger than the original problem.
- However, it is not enough to rescue the algorithm:
  - Substitutions have to be applied to terms in the problem, that leads to duplication of subterms.
  - In the example, unifying x_n and y_n, which by then are bound to terms with $2^{n+1} - 1$ symbols, will lead to exponential number of decompositions.

Unification via $\mathcal{U}$: Exponential in Time and Space

- Problem: Duplicate occurrences of the same variable cause the explosion in the size of terms.
- Fix: Represent terms as graphs which share subterms.

Term Dags

Term Dag

A term dag is a directed acyclic graph such that

- its nodes are labeled with function symbols or variables,
- its outgoing edges from any node are ordered,
- outdegree of any node labeled with a symbol f is equal to the arity of f,
- nodes labeled with variables have outdegree 0.

Term Dags

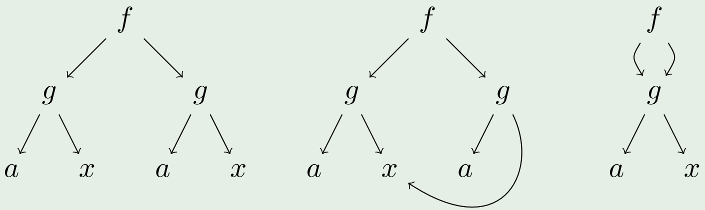
- Convention: Nodes and terms the term dags represent will not be distinguished.
- Example: "node" $f(a, x)$ is a node labeled with f and having two arcs to a and to x.

Term Dags

The only difference between various dags representing the same term is the amount of structure sharing between subterms.

Example

Three representations of the term $f(g(a,x),g(a,x))$:



Term Dags

- It is possible to build a dag with unique, shared variables for a given term in $O(n * \log(n))$ where n is the number of symbols in the term.
- There are subtle variations that can improve this result to $O(n)$.
- Assumption for the algorithm we plan to consider:
    - The input is a term dag representing the two terms to be unified, with unique, shared occurrences of all variables.

Term Dags

Representing substitutions involving only subterms of a term dag:

- Directly by a relation on the nodes of the dag, either
  - stored explicitly as a list of pairs, or
  - by storing a link ("substitution arcs") in the graph itself, and maintaining a list of variables (nodes) bound by the substitution.

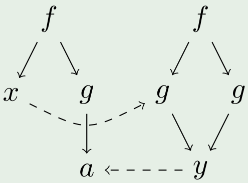
Term Dags

Substitution application.

- Implicit: Identifies two nodes connected with a substitution arc, without actually moving any of the subterm links.

Example

A term dag for the terms $f(x, g(a))$ and $f(g(y), g(y))$, with the implicit application of their mgu $\{x \mapsto g(a), y \mapsto a\}$.



Term Dags

- With implicit application, the binding for a variable can be determined by traversing the graph depth first, left to right.

Improvement 1: Linear Space, Exponential Time

Assumptions:

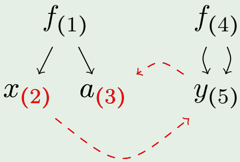
- Dags consist of nodes.
- Any node in a given dag defines a unique subdag (consisting of the nodes which can be reached from this node), and thus a unique subterm.
- Two different types of nodes: variable nodes and function nodes.
- Information at function nodes:
  - The name of the function symbol.
  - The arity n of this symbol.
  - The list (of length n) of successor nodes (corresponds to the argument list of the function)
- Both function and variable nodes may be equipped with one extra pointer (dashed arrow in diagrams) to another node.

Auxiliary procedures for Unification on Term Dags

- `Find`:
  Takes a node of a dag as input, and follows the additional pointers until it reaches a node without such a pointer. This node is the output of `Find`.

Example

- `Find`(3)=(3)
- `Find`(2)= (3)



Auxiliary procedures for Unification on Term Dags

- `Union`:
  Takes as input a pair of nodes u, v that do not have additional pointers and creates such a pointer from u to v.

Auxiliary procedures for Unification on Term Dags

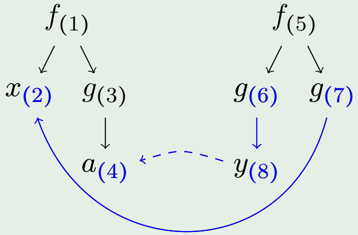
- `Occur`:
  Takes as input a variable node u and another node v (both without additional pointers) and performs the occur check, i.e. it tests whether the variable is contained in the term corresponding to v. The test is performed on the virtual term expressed by the additional pointer structure, i.e. one applies `Find` to all nodes that are reached during the test.

Auxiliary procedures for Unification on Term Dags

- Occur

Example

- Occur(2,6)=False
- Occur(2,7)=True



Unification of Term Dags

Input: A pair of nodes k_1 and k_2 in a dag

Output: *True* if the terms corresponding to k_1 and k_2 are unifiable. *False* Otherwise.

Side Effect: A pointer structure which allows to read off an mgu and the unified term.

Procedure `Unify1`. Unification of term dags.

(Continues on the next slide)

Unification of Term Dags

`Unify1` (k_1, k_2)

if $k_1 = k_2$ **then return** *True*; /* Trivial */
else
  if *function-node*(k_2) **then**
    $u := k_1; v := k_2$
  else
    $u := k_2; v := k_1$; /* Orient */
  end

Procedure `Unify1`. Unification of term dags.

(Continues on the next slide)

Unification of Term Dags

```
if variable-node(u) then
    if Occurs (u, v) ;                              /* Occur-check */
    then
        return False
    else
        Union(u, v) ;                               /* Variable elimination */
        return True
    end
else if function-symbol(u) ≠ function-symbol(v)
then
    return False;                                   /* Symbol clash */
```

Procedure `Unify1`. Unification of term dags. Continued.

(Continues on the next slide)

Unification of Term Dags

else

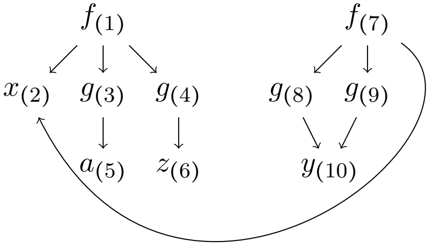
n :=arity(function-symbol(u));
$(u_1, \ldots, u_n)$:=succ-list(u);
$(v_1, \ldots, v_n)$:=succ-list(v);
$i := 0$; bool:=True;

while $i \leq n$ **and** *bool* **do**
$i := i + 1$; $bool := \texttt{Unify1}(\texttt{Find}(u_i), \texttt{Find}(v_i))$; /* Decomp. */
end
return *bool*

Procedure `Unify1`. Unification of term dags. Finished.

Unification of Term Dags. Example 1

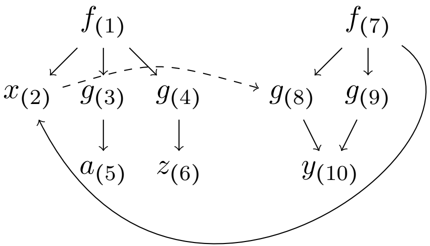
- Unify $f(x, g(a), g(z))$ and $f(g(y), g(y), x)$.
- First, create dags.
- Numbers indicate nodes.



Unification of Term Dags. Example 1

Algorithm run starts with Unify1$(1, 7)$ and continues:

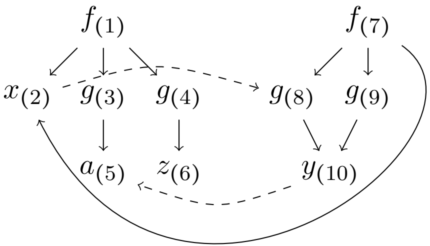
Unify1(Find(2), Find(8))
  Find(2) = (2)
  Find(8) = (8)
  Occur$(2, 8)$ =*False*
  Union$(2, 8)$



Unification of Term Dags. Example 1

Algorithm run starts with $\texttt{Unify1}(1, 7)$ and continues:

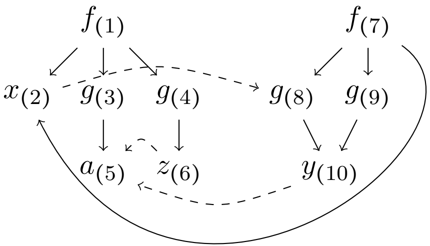
$\texttt{Unify1}(\texttt{Find}(3), \texttt{Find}(9))$
  $\texttt{Find}(3) = (3)$
  $\texttt{Find}(9) = (9)$
  $\texttt{Unify1}(\texttt{Find}(5), \texttt{Find}(10))$
    $\texttt{Find}(5) = 5$
    $\texttt{Find}(10) = 10$
    $orient(10, 5)$
    $\texttt{Occur}(10, 5) =$ *False*
    $\texttt{Union}(10, 5)$



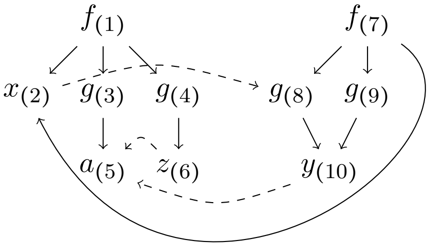
Unification of Term Dags. Example 1

Algorithm run starts with $\texttt{Unify1}(1, 7)$ and continues:

$\texttt{Unify1}(\texttt{Find}(4), \texttt{Find}(2))$
  $\texttt{Find}(4) = 4$
  $\texttt{Find}(2) = 8$
$\texttt{Unify1}(4, 8)$
  $\texttt{Unify1}(\texttt{Find}(6), \texttt{Find}(10))$
    $\texttt{Find}(6) = 6$
    $\texttt{Find}(10) = 5$
    $\texttt{Occur}(6, 5) =$ *False*
    $\texttt{Union}(6, 5)$
True



Unification of Term Dags. Example 1 (Cont.)



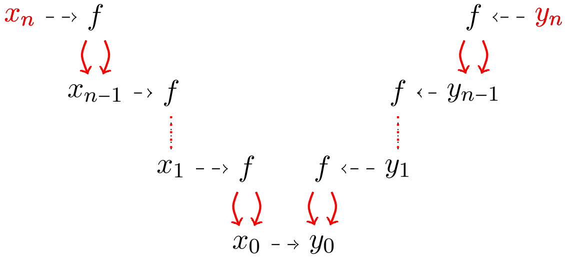
- From the final dag one can read off:
  - The unified term $f(g(a), g(a), g(a))$.
  - The mgu in triangular form $[x \mapsto g(y); y \mapsto a; z \mapsto a]$.
- No new nodes. Only one extra pointer for each variable node.
- Needs linear space.
- Time is still exponential. See the next example.

Unification of Term Dags. Example 2

Consider again the problem $s \doteq^? t$, where

$$s = h(x_1, x_2, \ldots, x_n, f(y_0, y_0), f(y_1, y_1), \ldots, f(y_{n-1}, y_{n-1}), y_n)$$
$$t = h(f(x_0, x_0), f(x_1, x_1), \ldots, f(x_{n-1}, x_{n-1}), y_1, y_2, \ldots, y_n, x_n)$$

A dag representation of the term bound to x_n and y_n:



Exponential number of recursive calls.

Unification of Term Dags: Correctness

`Unify1` can be simulated by $\mathcal{U}$ such that

- If the call to `Unify1` ends in failure, then the corresponding transformation sequence in $\mathcal{U}$ ends in $\perp$.
- If the call to `Unify1` terminates with success, with a substitution σ read from the pointer structure, then the corresponding transformation sequence $\mathcal{U}$ ends in $\emptyset; S$ where $\sigma_S = \sigma$.

Unification of Term Dags: Complexity

- Linear space: terms are not duplicated anymore.
- Exponential time: Calls `Unify1` recursively exponentially often.
- Fortunately, with an easy trick one can make the running time quadratic.
- Idea: Keep from revisiting already-solved problems in the graph.
- The algorithm of Corbin and Bidoit:

J. Corbin and M. Bidoit.
A rehabilitation of Robinson's unification algorithm.
In R. Mason, editor, *Information Processing 83*, pages 909–914. Elsevier Science, 1983.

Improvement 2. Linear Space, Quadratic Time

Input: A pair of nodes k_1 and k_2 in a dag.

Output: *True* if the terms corresponding to k_1 and k_2 are unifiable. *False* Otherwise.

Side Effect: A pointer structure which allows to read off an mgu and the unified term.

Procedure `Unify2`. Quadratic Algorithm.

(No difference from `Unify1` so far. Continues on the next slide)

Quadratic Algorithm

`Unify2` (k_1, k_2)

if $k_1 = k_2$ **then return** *True*; /* Trivial */
else
  if *function-node*(k_2) **then**
    $u := k_1; v := k_2$
  else
    $u := k_2; v := k_1$; /* Orient */
  end

Procedure `Unify2`. Quadratic Algorithm.

(No difference from `Unify1` so far. Continues on the next slide)

Quadratic Algorithm

```
if variable-node(u) then
    if Occurs (u, v) ;                              /* Occur-check */
    then
        return False
    else
        Union(u, v) ;                               /* Variable elimination */
        return True
    end
else if function-symbol(u) ≠ function-symbol(v)
then
    return False;                                   /* Symbol clash */
```

Procedure `Unify2`. Quadratic Algorithm. Continued.

(No difference from `Unify1` so far. Continues on the next slide)

Quadratic Algorithm

else

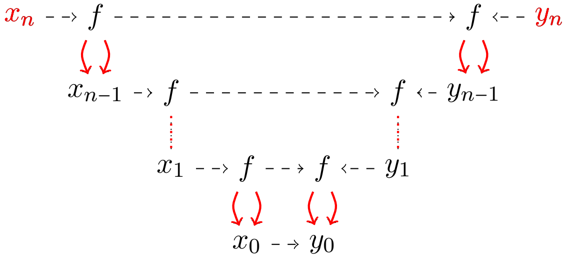
n :=arity(function-symbol(u));
($u_1, \ldots, u_n$) :=succ-list(u);
($v_1, \ldots, v_n$) :=succ-list(v);
$i := 0$; bool:=True;

`Union(u,v);`
while $i \leq n$ **and** *bool* **do**
$i := i + 1$; *bool* := `Unify2`(`Find`(u_i), `Find`(v_i)); /* Decomp. */
end
return *bool*

Procedure `Unify2`. Quadratic Algorithm. Finished.
(The only difference from `Unify1` is `Union(u,v)`.)

Quadratic Algorithm. Example

The same example that revealed exponential behavior of RDA:



Why is it Quadratic?

- The algorithm is quadratic in the number of symbols in original terms:
  - Each call of `Unify2` either returns immediately, or makes one more node unreachable for the `Find` operation.
  - Therefore, there can be only linearly many calls of `Unify2`.
  - Quadratic complexity comes from the fact that `Occur` and `Find` operations are linear.

Improvement 3. Almost Linear Algorithm

How to eliminate two sources of nonlinearity of `Unify2`?

- `Occur`: Just omit the occur check during the execution of the algorithm.
  - Consequence: The data structure may contain cycles.
  - Since the occur-check failures are not detected immediately, at the end an extra check has to be performed to find out whether the generated structure is cyclic or not.
  - Detecting cycles in a directed graph can be done by linear search.
- `Find`: Use more efficient union-find algorithm from

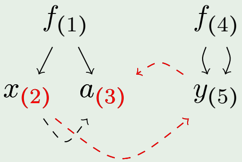
R. Tarjan.
Efficiency of a good but not linear set union algorithm.
J. ACM, 22(2):215–225, 1975.

Auxiliary Procedures for the Almost Linear Algorithm

- `Collapsing-find`:
  - Like `Find` it takes a node k of a dag as input, and follows the additional pointers until the node $\texttt{Find}(k)$ is reached.
  - In addition, `Collapsing-find` relocates the pointer of all the nodes reached during this process to $\texttt{Find}(k)$.

Example

- $\mathrm{CF}(3)=(3)$
- $\mathrm{CF}(2)=(3)$



Auxiliary Procedures for the Almost Linear Algorithm

- `Union-with-weight`:
  - Takes as input a pair of nodes u, v that do not have additional pointers.
  - If the set $\{k \mid \texttt{Find}(k) = u\}$ larger than the set $\{k \mid \texttt{Find}(k) = v\}$ then it creates an additional pointer from v to u.
  - Otherwise, it creates an additional pointer from u to v.
  - Hence, the link is created from the smaller tree to the larger one, increasing the path to the root (the result of `Find`) for fewer nodes.

Weighted union does not apply when we have a variable node and a function node.

Almost Linear Algorithm

One more auxiliary procedure:

- `Not-cyclic`:
  - Takes a node k as input, and tests the graph which can be reached from k for cycles.
  - The test is performed on the virtual graph expressed by the additional pointer structure, i.e. one first applies `Collapsing-find` to all nodes that are reached during the test.

Almost Linear Algorithm

Input: A pair of nodes k_1 and k_2 in a directed graph.
Output: *True* if k_1 and k_2 correspond unifiable terms. *False* Otherwise.
Side Effect: A pointer structure which allows to read off an mgu and the unified term.

`Unify3` (k_1, k_2)
if `Cyclic-unify` (k_1, k_2) `and` `Not-Cyclic` (k_1) **then**
  return *True*
else
  return *False*
end

Procedure `Unify3`. Almost Linear Algorithm.

(Continues on the next slide)

Almost Linear Algorithm

`Cyclic-unify` (k_1, k_2)

if $k_1 = k_2$ **then return** *True*; /* Trivial */
else
  if *function-node*(k_2) **then**
    $u := k_1; v := k_2$
  else
    $u := k_2; v := k_1$; /* Orient */
  end

Procedure `Cyclic-unify`.

(Continues on the next slide)

Almost Linear Algorithm

if *variable-node*(u) **then**
  if *variable-node*(v) **then**
    `Union-with-weight`(u, v)
  else
    `Union`(u, v); /* No occur-check. Variable elimination */
    return *True*
  end
else if *function-symbol*(u) $\neq$ *function-symbol*(v)
then
  return *False*; /* Symbol clash */

Procedure `Cyclic-unify`.

(Continues on the next slide)

Almost Linear Algorithm

else

n :=arity(function-symbol(u));
$(u_1, \ldots, u_n)$:=succ-list(u);
$(v_1, \ldots, v_n)$:=succ-list(v);
$i := 0$; bool:=True;

`Union-with-weight` (u,v);
while $i \leq n$ **and** *bool* **do**
$i := i + 1$;
bool:=`Cyclic-unify(Collapsing-find(`u_i`)`
`Collapsing-find(`v_i`))`; /* Decomposition */

end
return *bool*

Procedure `Cyclic-unify`. Finished.

Almost Linear Algorithm

The algorithm is very similar to the one described in Gerard Huet's thesis:

G. Huet.
Résolution d'Équations dans des Langages d'ordre $1, 2, \ldots, \omega$.
Thèse d'État, Université de Paris VII, 1976.

Complexity

- The algorithm is almost linear in the number of symbols in original terms:
    - Each call of `Cyclic-unify` either returns immediately, or makes one more node unreachable for the `Collapsing-find` operation.
    - Therefore, there can be only linearly many calls of `Cyclic-unify`.
    - A sequence of n `Collapsing-find` and `Union-with-weight` operations can be done in $O(n * \alpha(n))$ time, where α is an extremely slowly growing function (functional inverse of Ackerman's function) never exceeding 5 for practical input.
    - The use of nonoptimal `Union` can increase the time complexity at most by a summand $O(m)$ where m is the number of different variable nodes.
    - Therefore, complexity of `Cyclic-unify` is $O(n * \alpha(n))$.
    - Complexity of `Not-cyclic` is linear.
    - Hence, complexity of `Unify3` is $O(n * \alpha(n))$.

Implementation: Matching vs. Unification

- Unlike matching, efficient unification algorithms require sophisticated data structures.
- When efficiency is an issue, matching should be implemented separately from unification.

ISR 2012 sponsors