Introduction to Unification Theory

Syntactic Unification

Temur Kutsia

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

What is Unification

- Goal: Identify two symbolic expressions.
- Method: Replace certain subexpressions (variables) by other expressions.

What is Unification

- Goal: Identify two symbolic expressions.
- Method: Replace certain subexpressions (variables) by other expressions.

Example

- Goal: Identify $f(x, a)$ and $f(b, y)$.
- Method: Replace the variable x by b, and the variable y by a. Both initial expressions become $f(b, a)$.

What is Unification

- Goal: Identify two symbolic expressions.
- Method: Replace certain subexpressions (variables) by other expressions.

Example

- Goal: Identify $f(x, a)$ and $f(b, y)$.
- Method: Replace the variable x by b, and the variable y by a. Both initial expressions become $f(b, a)$.
- Of course, one should know what expressions are variables, and what are not.
  (Syntax: variables, function symbols, terms, etc.)

What is Unification

- Goal: Identify two symbolic expressions.
- Method: Replace certain subexpressions (variables) by other expressions.

Example

- Goal: Identify $f(x, a)$ and $f(b, y)$.
- Method: Replace the variable x by b, and the variable y by a. Both initial expressions become $f(b, a)$.
- Of course, one should know what expressions are variables, and what are not.
  (Syntax: variables, function symbols, terms, etc.)
- The *substitution* $\{x \mapsto b, y \mapsto a\}$ *unifies* the *terms* $f(x, a)$ and $f(b, y)$.

What is Unification

- Goal: Identify two symbolic expressions.
- Method: Replace certain subexpressions (variables) by other expressions.

Example

- Goal: Identify $f(x,a)$ and $f(b,y)$.
- Method: Replace the variable x by b, and the variable y by a. Both initial expressions become $f(b,a)$.
- Of course, one should know what expressions are variables, and what are not.
  (Syntax: variables, function symbols, terms, etc.)
- The *substitution* $\{x \mapsto b, y \mapsto a\}$ *unifies* the *terms* $f(x,a)$ and $f(b,y)$.
- Solving the equation $f(x,a) = f(b,y)$ for x and y.

What is Unification

- Goal of unification: Identify two symbolic expressions.
- Method: Replace certain subexpressions (variables) by other expressions.

Depending what is meant under "identify" (syntactic identity or equality modulo some equations) one speaks about *syntactic unification* or *equational unification*.

Example

- The terms $f(x, a)$ and $g(a, x)$ are not syntactically unifiable.
- However, they are unifiable modulo the equation $f(a, a) = g(a, a)$ with the substitution $\{x \mapsto a\}$.

What is Unification

- Goal of unification: Identify two symbolic expressions.
- Method: Replace certain subexpressions (variables) by other expressions.

Depending at which positions the variables are allowed to occur, and which kind of expressions they are allowed to be replaced by, one speaks about *first-order unification* or *higher-order unification*.

Example

- If G and x are variables, the terms $f(x,a)$ and $G(a,x)$ can not be subjected to first-order unification.
- $G(a,x)$ is not a first-order term: G occurs in the top position.
- However, $f(x,a)$ and $G(a,x)$ can be unified by higher-order unification with the substitution $\{x \mapsto a, G \mapsto f\}$.

What is Unification Good For?

- To make an inference step in theorem proving.
- To perform an inference in logic programming.
- To make a rewriting step in term rewriting.
- To generate a critical pair in completion.
- To extract a part from structured or semistructured data.
- For type inference in programming languages.
- For matching in pattern-based languages.
- For program schema manipulation.
- For various formalisms in computational linguistics.
- etc.

What this Course Is (Not) About

The course gives an introduction to

What this Course Is (Not) About

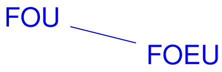
The course gives an introduction to

- First-order syntactic unification.

What this Course Is (Not) About

The course gives an introduction to

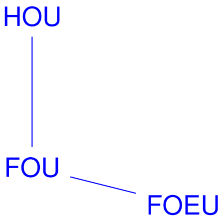
- First-order syntactic unification.
- First-order equational unification.



What this Course Is (Not) About

The course gives an introduction to

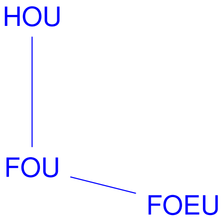
- First-order syntactic unification.
- First-order equational unification.
- Higher-order unification.



What this Course Is (Not) About

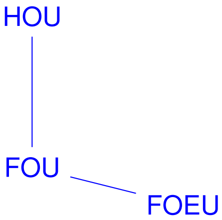
The course gives an introduction to

- First-order syntactic unification.
- First-order equational unification.
- Higher-order unification.
- Applications of unification.



What this Course Is (Not) About

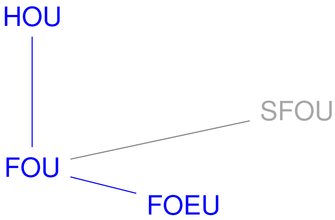
There are many interesting topics not considered here, e.g.,



What this Course Is (Not) About

There are many interesting topics not considered here, e.g.,

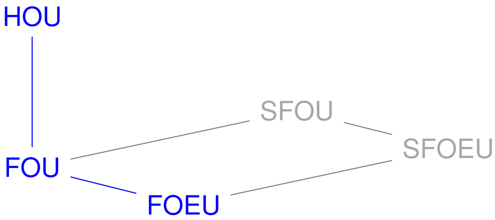
- First-order (order-)sorted syntactic unification.



What this Course Is (Not) About

There are many interesting topics not considered here, e.g.,

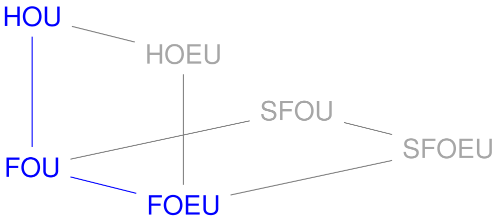
- First-order (order-)sorted syntactic unification.
- First-order (order-)sorted equational unification.



What this Course Is (Not) About

There are many interesting topics not considered here, e.g.,

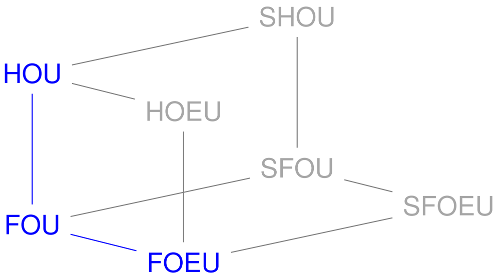
- First-order (order-)sorted syntactic unification.
- First-order (order-)sorted equational unification.
- higher-order equational unification,



What this Course Is (Not) About

There are many interesting topics not considered here, e.g.,

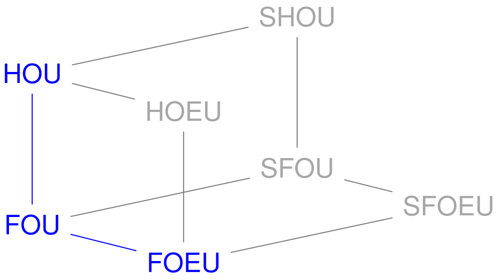
- First-order (order-)sorted syntactic unification.
- First-order (order-)sorted equational unification.
- higher-order equational unification,
- (order-)sorted higher-order equational unification,



What this Course Is (Not) About

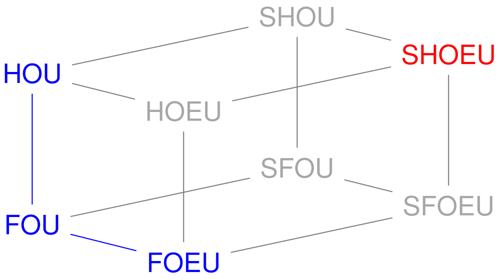
There are many interesting topics not considered here, e.g.,

- First-order (order-)sorted syntactic unification.
- First-order (order-)sorted equational unification.
- higher-order equational unification,
- (order-)sorted higher-order equational unification,
- special unification algorithms, related problems



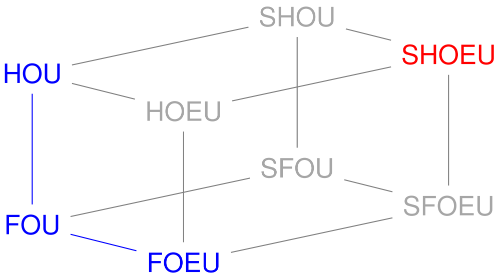
What this Course Is (Not) About

(Order-)sorted higher-order equational unification has not been investigated.



What this Course Is (Not) About

Warning! This "unification cube" is just an illustration of relations between *certain* problems, not a reflection of the *whole* unification field!



Reading: Main Sources

F. Baader and W. Snyder.
Unification Theory.
In A. Robinson and A. Voronkov, editors, *Handbook of Automated Reasoning*, pages 447–533. Elsevier, 2001.

F. Baader and J. Siekmann.
Unification Theory.
In D. Gabbay, C. Hogger and A. Robinson, editors, *Handbook of Logic in Artificial Intelligence and Logic Programming*, Oxford University Press, 1994.

W. Snyder and J. Gallier.
Higher-Order Unification Revisited: Complete Sets of Transformations.
J. Symbolic Computation, **8**(1–2), 101–140, 1989.

Reading: Additional Literature

- F. Baader and T. Nipkow.
  Term Rewriting and All That.
  Cambridge University Press, 1998.
- G. Dowek.
  Higher Order Unification and Matching.
  In: *Handbook of Automated Reasoning*, Elsevier, 2001.
- C. Kirchner (ed.)
  Unification.
  Academic Press, London, 1990.
- C. Kirchner and H. Kirchner.
  Rewriting, Solving, Proving.
- K. Knight.
  Unification: A Multidisciplinary Survey.
  ACM Computing Surveys, 21(1), 1989.

Results from various papers.

Brief History

1920s: Emil Posts diary and notes contain the first hint of the concept of a unification algorithm that computes a most general representative as opposed to all possible instantiations.

Brief History

1920s: Emil Posts diary and notes contain the first hint of the concept of a unification algorithm that computes a most general representative as opposed to all possible instantiations.

1930: The first explicit account of unification algorithm was given in Jacques Herbrand's doctoral thesis. It was the first published unification algorithm and was based on a technique later rediscovered by Alberto Martelli and Ugo Montanari, still in use today.

Brief History

1920s: Emil Posts diary and notes contain the first hint of the concept of a unification algorithm that computes a most general representative as opposed to all possible instantiations.

1930: The first explicit account of unification algorithm was given in Jacques Herbrand's doctoral thesis. It was the first published unification algorithm and was based on a technique later rediscovered by Alberto Martelli and Ugo Montanari, still in use today.

1962: First implementation of unification algorithm at Bell Labs, as a part of the proof procedure that combined Prawitz's and Davis-Putnam methods.

Brief History

1920s: Emil Posts diary and notes contain the first hint of the concept of a unification algorithm that computes a most general representative as opposed to all possible instantiations.

1930: The first explicit account of unification algorithm was given in Jacques Herbrand's doctoral thesis. It was the first published unification algorithm and was based on a technique later rediscovered by Alberto Martelli and Ugo Montanari, still in use today.

1962: First implementation of unification algorithm at Bell Labs, as a part of the proof procedure that combined Prawitz's and Davis-Putnam methods.

1964: Jim Guard's team at Applied Logic Corporation started working on higher-order versions of unification.

Brief History

1965: Alan Robinson introduced unification as the basic operation of his resolution principle, and gave a formal account of an algorithm that computes a most general unifier for first-order terms. This paper (A Machine Oriented Logic Based on the Resolution Principle, J. ACM) has been the most influential paper in the field. The name "unification" was first used in this work.

Brief History

1965: Alan Robinson introduced unification as the basic operation of his resolution principle, and gave a formal account of an algorithm that computes a most general unifier for first-order terms. This paper (A Machine Oriented Logic Based on the Resolution Principle, J. ACM) has been the most influential paper in the field. The name "unification" was first used in this work.

1966: W.E Gould showed that a minimal set of most general unifiers does not exist for ω-order logics.

Brief History

1965: Alan Robinson introduced unification as the basic operation of his resolution principle, and gave a formal account of an algorithm that computes a most general unifier for first-order terms. This paper (A Machine Oriented Logic Based on the Resolution Principle, J. ACM) has been the most influential paper in the field. The name "unification" was first used in this work.

1966: W.E Gould showed that a minimal set of most general unifiers does not exist for ω-order logics.

1967: Donald Knuth and Peter Bendix independently reinvented "unification" and “most general unifier” as a tool for testing term rewriting systems for local confluence by computing critical pairs.

Brief History

1972: Gerard Huet and Claudio Lucchesi showed undecidability of higher-order unification. Warren Goldfarb sharpened the result later (in 1981).

Brief History

1972: Gerard Huet and Claudio Lucchesi showed undecidability of higher-order unification. Warren Goldfarb sharpened the result later (in 1981).

1972: Gordon Plotkin showed how to build certain equational axioms into the inference rule for proving (resolution) without loosing completeness, replacing syntactic unification by unification modulo the equational theory induced by the axioms to be built in.

Brief History

1972: Gerard Huet and Claudio Lucchesi showed undecidability of higher-order unification. Warren Goldfarb sharpened the result later (in 1981).

1972: Gordon Plotkin showed how to build certain equational axioms into the inference rule for proving (resolution) without loosing completeness, replacing syntactic unification by unification modulo the equational theory induced by the axioms to be built in.

1972: Huet developed a constrained resolution method for higher-order theorem proving, based on an ω-order unification algorithm. Peter Andrews and the collaborators later implemented the method in the TPS system.

Brief History

1972: Gerard Huet and Claudio Lucchesi showed undecidability of higher-order unification. Warren Goldfarb sharpened the result later (in 1981).

1972: Gordon Plotkin showed how to build certain equational axioms into the inference rule for proving (resolution) without loosing completeness, replacing syntactic unification by unification modulo the equational theory induced by the axioms to be built in.

1972: Huet developed a constrained resolution method for higher-order theorem proving, based on an ω-order unification algorithm. Peter Andrews and the collaborators later implemented the method in the TPS system.

1976: Huet further developed this work in his Thèse d'État. A fundamental contribution in the field of first- and higher-order unification theory.

Brief History

1978: Jörg Siekmann in his thesis introduced unification hierarchy and suggested that unification theory was worthy of study as a field in its own right.

Brief History

1978: Jörg Siekmann in his thesis introduced unification hierarchy and suggested that unification theory was worthy of study as a field in its own right.

1980s: Further improvement of unification algorithms, starting series of Unification Workshops (UNIF).

Brief History

1978: Jörg Siekmann in his thesis introduced unification hierarchy and suggested that unification theory was worthy of study as a field in its own right.

1980s: Further improvement of unification algorithms, starting series of Unification Workshops (UNIF).

1990s: Maturing the field, broadening application areas, combination method of Franz Baader and Klaus Schulz.

Brief History

- 1978: Jörg Siekmann in his thesis introduced unification hierarchy and suggested that unification theory was worthy of study as a field in its own right.
- 1980s: Further improvement of unification algorithms, starting series of Unification Workshops (UNIF).
- 1990s: Maturing the field, broadening application areas, combination method of Franz Baader and Klaus Schulz.
- 2006: Colin Stirling proved decidability of higher-order matching (for the classical case), an open problem for 30 years.

Brief History

- 1978: Jörg Siekmann in his thesis introduced unification hierarchy and suggested that unification theory was worthy of study as a field in its own right.
- 1980s: Further improvement of unification algorithms, starting series of Unification Workshops (UNIF).
- 1990s: Maturing the field, broadening application areas, combination method of Franz Baader and Klaus Schulz.
- 2006: Colin Stirling proved decidability of higher-order matching (for the classical case), an open problem for 30 years.
- 2014: Artur Jeż proved decidability of context unification, an open problem for more than 20 years.

Terms

Alphabet:

- A set of fixed arity function symbols $\mathcal{F}$.
- A countable set of variables $\mathcal{V}$.
- $\mathcal{F}$ and $\mathcal{V}$ are disjoint.

Terms over $\mathcal{F}$ and $\mathcal{V}$:

$$t ::= x \mid f(t_1, \ldots, t_n),$$

where

- $n \geq 0$,
- x is a variable,
- f is an n-ary function symbol.

Terms

Conventions, notation:

- Constants: 0-ary function symbols.
- x, y, z denote variables.
- a, b, c denote constants.
- f, g, h denote arbitrary function symbols.
- s, t, r denote terms.
- Parentheses omitted in terms with the empty list of arguments: a instead of $a()$.

Terms

Conventions, notation:

- Ground terms: terms without variables.
- $\mathcal{T}(\mathcal{F},\mathcal{V})$: the set of terms over $\mathcal{F}$ and $\mathcal{V}$.
- $\mathcal{T}(\mathcal{F})$: the set of ground terms over $\mathcal{F}$.
- 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.

Terms

Example

- $f(x, g(x, a), y)$ is a term, where f is ternary, g is binary, a is a constant.
- $vars(f(x, g(x, a), y)) = \{x, y\}$.
- $f(b, g(b, a), c)$ is a ground term.
- $vars(f(b, g(b, a), c)) = \emptyset$.

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.

Notation

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

Substitutions

Substitution Application

Applying a substitution σ to a term t:

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

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)

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

$$\begin{aligned} dom(\{x \mapsto f(a,b), y \mapsto g(c)\}) &= \{x, y\} \\ ran(\{x \mapsto f(a,b), y \mapsto g(c)\}) &= \{f(a,b), g(c)\} \\ vran(\{x \mapsto f(a,b), y \mapsto g(c)\}) &= \varnothing \text{ (ground substitution)} \end{aligned}$$

$$\begin{aligned} dom(\varepsilon) &= \varnothing \\ ran(\varepsilon) &= \varnothing \\ vran(\varepsilon) &= \varnothing \end{aligned}$$

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: $\sigma\vartheta$.
- $t(\sigma\vartheta) = (t\sigma)\vartheta$.
- Informal algorithm for constructing the representation of the composition $\sigma\vartheta$:
  1. σ and ϑ are given by their representation.
  2. Apply ϑ to every term in $ran(\sigma)$ to obtain σ_1.
  3. Remove from ϑ any binding $x \mapsto t$ with $x \in dom(\sigma)$ to obtain ϑ_1.
  4. Remove from σ_1 any trivial binding $x \mapsto x$ to obtain σ_2.
  5. Take the union of the sets of bindings σ_2 and ϑ_1.

Jump to RDA

Substitutions

Example (Composition)

1. $\sigma = \{x \mapsto f(y), y \mapsto z\}$
   $\vartheta = \{x \mapsto a, y \mapsto b, z \mapsto y\}$
2. $\sigma_1 = \{x \mapsto f(y)\vartheta, y \mapsto z\vartheta\} = \{x \mapsto f(b), y \mapsto y\}$
3. $\vartheta_1 = \{z \mapsto y\}$
4. $\sigma_2 = \{x \mapsto f(b)\}$
5. $\sigma\vartheta = \{x \mapsto f(b), z \mapsto y\}$

Composition is not commutative:

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

Substitutions

Elementary Properties of Substitutions

Theorem

- *Composition of substitutions is associative.*
- *For all $\mathcal{X} \subseteq \mathcal{V}$, t and σ, if $vars(t) \subseteq \mathcal{X}$ then $t\sigma = t\sigma|_{\mathcal{X}}$.*
- *For all σ, ϑ, and t, if $t\sigma = t\vartheta$ then $t\sigma|_{vars(t)} = t\vartheta|_{vars(t)}$*

Proof.

Exercise. □

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, Inverse

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
- $\{x_1, \ldots, x_n\} = \{y_1, \ldots, y_n\}$.

The *inverse* of σ, denoted σ^{-1}, is the substitution

$$\sigma^{-1} = \{y_1 \mapsto x_1, y_2 \mapsto x_2, \ldots, y_n \mapsto x_n\}$$

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: $\vartheta\vartheta = \sigma \neq \vartheta$.

Theorem

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

Proof.

Exercise. $\square$

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

- 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:

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

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

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

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 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.

Example

$$f(x, g(a), g(z))$$

$$\uparrow$$

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

$$\uparrow$$

Example

$f(x, g(a), g(z))$

$\uparrow$

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

$\uparrow$

Example

$$f(x, g(a), g(z))$$

$$\uparrow$$

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

$$\uparrow$$

Example

$$f(x, g(a), g(z))$$

$$\uparrow$$

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

$$\uparrow$$

Example

$$f(g(y), g(a), g(z))$$

$$\uparrow$$

$$f(g(y), g(y), g(g(g(y))))$$

$$\uparrow$$

Example

$$f(g(y), g(a), g(z))$$

$$\uparrow$$

$$f(g(y), g(y), g(g(g(y))))$$

$$\uparrow$$

$$\{x \mapsto g(y)\}$$

Example

$$f(g(y), g(a), g(z))$$

$$\uparrow$$

$$f(g(y), g(y), g(g(g(y))))$$

$$\uparrow$$

$$\{x \mapsto g(y)\}$$

Example

$$f(g(y), g(a), g(z))$$

$$\uparrow$$

$$f(g(y), g(y), g(g(g(y))))$$

$$\uparrow$$

$$\{x \mapsto g(y)\}$$

Example

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

$$\uparrow$$

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

$$\uparrow$$

$$\{x \mapsto g(a)\}$$

Example

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

$$\uparrow$$

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

$$\uparrow$$

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

Example

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

$$\uparrow$$

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

$$\uparrow$$

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

Example

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

$$\uparrow$$

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

$$\uparrow$$

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

Example

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

$$\uparrow$$

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

$$\uparrow$$

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

Example

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

$$\uparrow$$

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

$$\uparrow$$

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

Example

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

$$\uparrow$$

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

$$\uparrow$$

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

Example

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

$$\uparrow$$

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

$$\uparrow$$

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

Interesting Questions

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?

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?

Complexity:

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

Answers

On the coming slides, for various unification algorithms.

Implementation: Unification by Recursive Descent

Implementation of the naive algorithm:

- Term representation: either by explicit pointer structures or by built-in recursive data types (depending on the implementation language).
- Substitution representation: a list of pairs of terms.
- Application of a substitution: constructing a new term or replacing a variable with a new term.
- The left-to-right search for disagreements: implemented by recursive descent through the terms.

Example

The Recursive Descent Algorithm we are going to describe will correspond to a slightly modified version of the naive algorithm:

$$f(x, g(a), g(z))$$

$$\uparrow$$

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

$$\uparrow$$

Example

The Recursive Descent Algorithm we are going to describe will correspond to a slightly modified version of the naive algorithm:

$$f(x, g(a), g(z))$$

$$\uparrow$$

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

$$\uparrow$$

Example

The Recursive Descent Algorithm we are going to describe will correspond to a slightly modified version of the naive algorithm:

$$f(x, g(a), g(z))$$

$$\uparrow$$

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

$$\uparrow$$

Example

The Recursive Descent Algorithm we are going to describe will correspond to a slightly modified version of the naive algorithm:

$$f(x, g(a), g(z))$$

$$\uparrow$$

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

$$\uparrow$$

Example

The Recursive Descent Algorithm we are going to describe will correspond to a slightly modified version of the naive algorithm:

$$f(g(y), g(a), g(z))$$
$$\uparrow$$

$$f(g(y), g(y), g(g(x)))$$
$$\uparrow$$

Example

The Recursive Descent Algorithm we are going to describe will correspond to a slightly modified version of the naive algorithm:

$$f(g(y), g(a), g(z))$$

$$\uparrow$$

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

$$\uparrow$$

$$\{x \mapsto g(y)\}$$

Example

The Recursive Descent Algorithm we are going to describe will correspond to a slightly modified version of the naive algorithm:

$$f(g(y), \color{blue}{g}(a), g(z))$$

$$\uparrow$$

$$f(g(y), \color{blue}{g}(y), g(g(x)))$$

$$\uparrow$$

$$\{x \mapsto g(y)\}$$

Example

The Recursive Descent Algorithm we are going to describe will correspond to a slightly modified version of the naive algorithm:

$$f(g(y), g(a), g(z))$$

$$\uparrow$$

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

$$\uparrow$$

$$\{x \mapsto g(y)\}$$

Example

The Recursive Descent Algorithm we are going to describe will correspond to a slightly modified version of the naive algorithm:

$$f(g(y), g(a), g(z))$$

$$\uparrow$$

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

$$\uparrow$$

$$\{x \mapsto g(a)\}$$

Example

The Recursive Descent Algorithm we are going to describe will correspond to a slightly modified version of the naive algorithm:

$$f(g(y), g(a), g(z))$$

$$\uparrow$$

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

$$\uparrow$$

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

Example

The Recursive Descent Algorithm we are going to describe will correspond to a slightly modified version of the naive algorithm:

$$f(g(y), g(a), g(z))$$

$$\uparrow$$

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

$$\uparrow$$

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

Example

The Recursive Descent Algorithm we are going to describe will correspond to a slightly modified version of the naive algorithm:

$$f(g(y), g(a), g(z))$$

$$\uparrow$$

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

$$\uparrow$$

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

Example

The Recursive Descent Algorithm we are going to describe will correspond to a slightly modified version of the naive algorithm:

$$f(g(y), g(a), g(z))$$

$$\uparrow$$

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

$$\uparrow$$

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

Example

The Recursive Descent Algorithm we are going to describe will correspond to a slightly modified version of the naive algorithm:

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

$$\uparrow$$

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

$$\uparrow$$

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

Example

The Recursive Descent Algorithm we are going to describe will correspond to a slightly modified version of the naive algorithm:

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

$$\uparrow$$

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

$$\uparrow$$

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

Example

The Recursive Descent Algorithm we are going to describe will correspond to a slightly modified version of the naive algorithm:

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

$$\uparrow$$

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

$$\uparrow$$

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

Unification by Recursive Descent

Input: Terms s and t
Output: An mgu of s and t
Global: Substitution σ. Initialized to ε

`Unify` (s,t)
begin
  if s *is a variable* **then** $s := s\sigma$; $t := t\sigma$
  Print$(s, ' \doteq^? ', t, '\sigma = ', \sigma)$
  if s *is a variable and* $s = t$ **then** `Do nothing`
  else if $s = f(s_1, \ldots, s_n)$ **and** $t = g(t_1, \ldots, t_m)$, $n, m \geq 0$ **then**
    if $f = g$ **then for** $i := 1$ **to** n **do** Unify(s_i, t_i)
    else Exit with failure
  else if s *is not a variable* **then** `Unify` (t, s)
  else if s *occurs in* t **then** Exit with failure
  else $\sigma := \sigma\{s \mapsto t\}$
end

Algorithm 1: Recursive descent algorithm

Recursive Descent Algorithm

- Implementation of substitution composition: Without the steps 3 and 4 of the composition algorithm. ▸ Jump to composition
- Reason: When a binding $x \mapsto t$ created and applied, x does not appear in the terms anymore.

The Recursive Descent Algorithm is essentially the Robinson's Unification Algorithm.

Example

$s = f(x, g(a), g(z))$, $t = f(g(y), g(y), g(g(x)))$, $\sigma = \varepsilon$.

Printing outputs are given in *blue*.

$$Unify(f(x, g(a), g(z)), f(g(y), g(y), g(g(x))))$$

$$f(x, g(a), g(z)) \doteq^? f(g(y), g(y), g(g(x))), \sigma = \varepsilon$$

$$Unify(x, g(y))$$

$$x \doteq^? g(y), \sigma = \varepsilon$$

$$Unify(g(a), g(y))$$

$$g(a) \doteq^? g(y), \sigma = \{x \mapsto g(y)\}$$

Continues on the next slide.

Example (Cont.)

$$\dfrac{\dfrac{\dfrac{\dfrac{Unify(a, y)}{a \doteq^? y, \sigma = \{x \mapsto g(y)\}} \qquad \dfrac{Unify(y, a)}{y \doteq^? a, \sigma = \{x \mapsto g(y)\}}}{}}{}}{}$$

$$Unify(a, y)$$

$$a \doteq^? y, \sigma = \{x \mapsto g(y)\}$$

$$Unify(y, a)$$

$$y \doteq^? a, \sigma = \{x \mapsto g(y)\}$$

$$Unify(g(z), g(g(x)))$$

$$g(z) \doteq^? g(g(x)), \sigma = \{x \mapsto g(a), y \mapsto a\}$$

$$Unify(z, g(x))$$

$$z \doteq^? g(g(a)), \sigma = \{x \mapsto g(a), y \mapsto a\}$$

Result: $\sigma = \{x \mapsto g(a),\ y \mapsto a,\ z \mapsto g(g(a))\}$

Properties of Recursive Descent Algorithm

- Goal: Prove logical properties of the Recursive Descent Algorithm.
- Method (rule-based approach):
  1. Describe an inference system for deriving solutions for unification problems.
  2. Show that the inference system simulates the actions of the Recursive Descent Algorithm.
  3. Prove logical properties of the inference system.

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 $\bot$ or a pair $P; S$ where
  - P is a multiset of unification problems,
  - S is a set of equations in solved form.
- $\bot$ represents failure.
- A unifier (or a solution) of a system $P; S$: A substitution that unifies each of the equations in P and S.
- $\bot$ 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:

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

Symbol Clash:

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

[1] $\uplus$ is multiset union.

The Inference System $\mathcal{U}$

Orient:

$$\{t \doteq^? x\} \uplus P'; S \Longrightarrow \{x \doteq^? t\} \cup P'; S,$$

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.

Simulating the Recursive Descent Algorithm by $\mathcal{U}$

s, t, σ when printed in the Recursive Descent Algorithm:

$$\begin{array}{lll} s_1 & t_1 & \varepsilon \\ s_2 & t_2 & \sigma_2 \\ s_3 & t_3 & \sigma_3 \\ \ldots & & \end{array}$$

Can be simulated by the sequence of transformations:

$$\begin{array}{ll} & \{s_1 \doteq^? t_1\}; \varnothing \\ \Longrightarrow & \{s_2 \doteq^? t_2\} \cup P_2; S_2 \\ \Longrightarrow & \{s_3 \doteq^? t_3\} \cup P_3; S_3 \\ \ldots & \end{array}$$

where $s_i \doteq^? t_i$ is the equation acted on by a rule, and σ_i is σ_{S_i}.

Simulating the Recursive Descent Algorithm by $\mathcal{U}$

Furthermore:

- If the call to Unify in RDA ends in failure, then the transformation sequence ends in $\bot$.
- If the call to Unify in RDA terminates with success, with a global substitution σ_n, then the transformation sequence ends in $\emptyset; S$ where $\sigma_S = \sigma_n$.

Simulating the Recursive Descent Algorithm by $\mathcal{U}$

Furthermore:

- If the call to Unify in RDA ends in failure, then the transformation sequence ends in $\bot$.
- If the call to Unify in RDA terminates with success, with a global substitution σ_n, then the transformation sequence ends in $\emptyset; S$ where $\sigma_S = \sigma_n$.

This simulation can be achieved by

- treating P as a stack,
- always applying the rule to the top equation,
- only using **Trivial** when s is a variable.

There is only one rule applicable at each step under this control.

$\mathcal{U}$ — an abstract version of RDA.

Properties of $\mathcal{U}$: Termination

Lemma

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

$$P; \varnothing \Longrightarrow P_1; \sigma_1 \Longrightarrow P_2; \sigma_2 \Longrightarrow \cdots$$

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

Properties of $\mathcal{U}$: Termination

Proof.

Complexity measure on the multisets 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.

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

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 $\bot$ and $\emptyset; S$.
- 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.

□

Corollary

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

Proof.

σ_S unifies S. Induction using the previous lemma finishes the proof. □

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 $\sigma_S \lesssim \vartheta$.

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 $P; \varnothing \Longrightarrow \cdots \Longrightarrow \bot$.

Properties of $\mathcal{U}$: Correctness

Observations:

- 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.

Properties of $\mathcal{U}$: Correctness

Observations:

- Any substitution generated by $\mathcal{U}$ is a compact representation of the (infinite) set of all unifiers.
- The unifiers can be generated by composing all the possible substitutions with the mgu.
- Any two mgu's of a given pair of terms are instances of each other.
- The mgu's can be obtained from a single mgu by composition with variable renaming.
- By this operation it is possible to create an infinite number of mgu's.
- The finite search tree for $\mathcal{U}$ is not able to produce every idempotent 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

$f(x,y) \ll^? f(g(z),c)$ $\{x \mapsto g(z), y \mapsto c\}$	$f(x,y) \doteq^? f(g(z),c)$ $\{x \mapsto g(z), y \mapsto c\}$
$f(x,y) \ll^? f(g(z),x)$ $\{x \mapsto g(z), y \mapsto x\}$	$f(x,y) \doteq^? f(g(z),x)$ $\{x \mapsto g(z), y \mapsto g(z)\}$
$f(x,a) \ll^? f(b,y)$ No matcher	$f(x,a) \doteq^? f(b,y)$ $\{x \mapsto b, y \mapsto a\}$
$f(x,x) \ll^? f(x,a)$ No matcher	$f(x,x) \doteq^? f(x,a)$ $\{x \mapsto a\}$
$x \ll^? f(x)$ $\{x \mapsto f(x)\}$	$x \doteq^? f(x)$ 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 $\bot$ 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.
- $\bot$ has no matchers.

The Inference System $\mathcal{M}$

Five transformation rules on matching systems:[2]

Decomposition:

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

Symbol Clash:

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

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

The Inference System $\mathcal{M}$

Symbol-Variable Clash:

$$\{f(s_1, \ldots, s_n) \ll^? x\} \uplus P'; S \Longrightarrow \bot.$$

Merging Clash:

$$\{x \ll^? t_1\} \uplus P'; \{x \ll t_2\} \uplus S' \Longrightarrow \bot, \text{ if } t_1 \neq t_2.$$

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 $f(x,a)$:

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

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 $\perp$ or with $\varnothing; S$, with S in matched form.

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 $\perp$ 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 $\perp$ 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.

□

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*.*

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. □

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 $\sigma_S = \vartheta|_{v(P)}$.

Properties of $\mathcal{M}$: Correctness

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

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, $\sigma_S = \vartheta|_{v(P)}$. $\square$

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.

Complexity of Recursive Descent Unification

Can take exponential time and space.

Example

Let

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

Unifying s and t 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?

Complexity of Recursive Descent Unification

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, calling Unify on x_n and y_n, which by then are bound to terms with $2^{n+1} - 1$ symbols, will lead to exponential number of recursive calls.

How to Speed up Unification?

Develop

(a) more subtle data structures for terms.

(b) a different method for applying substitutions.

Details: The next lecture.