

Unification (computer science)



In [logic](#) and [computer science](#), **unification** is an algorithmic process of [solving equations](#) between symbolic [expressions](#).

Depending on which expressions (also called *terms*) are allowed to occur in an equation set (also called *unification problem*), and which expressions are considered equal, several frameworks of unification are distinguished. If higher-order variables, that is, variables representing [functions](#), are allowed in an expression, the process is called **higher-order unification**, otherwise **first-order unification**. If a solution is required to make both sides of each equation literally equal, the process is called **syntactic** or **free unification**, otherwise **semantic** or **equational unification**, or **E-unification**, or **unification modulo theory**.

A *solution* of a unification problem is denoted as a [substitution](#), that is, a mapping assigning a symbolic value to each variable of the problem's expressions. A unification algorithm should compute for a given problem a *complete*, and *minimal* substitution set, that is, a set covering all its solutions, and containing no redundant members. Depending on the framework, a complete and minimal substitution set may have at most one, at most finitely many, or possibly infinitely many members, or may not exist at all.^{[[note 1](#)][1]} In some frameworks it is generally impossible to decide whether any solution exists. For first-order syntactical unification, Martelli and Montanari^[2] gave an algorithm that reports unsolvability or computes a complete and minimal singleton substitution set containing the so-called **most general unifier**.

For example, using x, y, z as variables, the singleton equation set $\{ \text{cons}(x, \text{cons}(x, \text{nil})) = \text{cons}(2, y) \}$ is a syntactic first-order unification problem that has the substitution $\{ x \mapsto 2, y \mapsto \text{cons}(2, \text{nil}) \}$ as its only solution. The syntactic first-order unification problem $\{ y = \text{cons}(2, y) \}$ has no solution over the set of [finite terms](#); however, it has the single solution $\{ y \mapsto \text{cons}(2, \text{cons}(2, \text{cons}(2, \dots))) \}$ over the set of [infinite trees](#). The semantic first-order unification problem $\{ a \cdot x = x \cdot a \}$ has each substitution of the form $\{ x \mapsto a \cdot \dots \cdot a \}$ as a solution in a [semigroup](#), i.e. if (\cdot) is considered [associative](#); the same problem, viewed in an [abelian group](#), where (\cdot) is considered also [commutative](#), has any substitution at all as a solution. The singleton set $\{ a = y(x) \}$ is a syntactic second-order unification problem, since y is a function variable. One solution is $\{ x \mapsto a, y \mapsto (\text{identity function}) \}$; another one is $\{ y \mapsto (\text{constant function mapping each value to } a), x \mapsto (\text{any value}) \}$.

The first formal investigation of unification can be attributed to [John Alan Robinson](#),^{[3][4]} who used first-order syntactical unification as a basic building block of his [resolution](#) procedure for first-order logic, a great step forward in [automated reasoning](#) technology, as it eliminated one source of combinatorial explosion: searching for instantiation of terms. Today, automated reasoning is still the main application area of unification. Syntactical first-order unification is used in [logic programming](#) and programming language [type system](#) implementation, especially in [Hindley–Milner](#) based [type inference](#) algorithms. Semantic unification is used in [SMT solvers](#), [term](#)

rewriting algorithms and cryptographic protocol analysis. Higher-order unification is used in proof assistants, for example Isabelle and Twelf, and restricted forms of higher-order unification (**higher-order pattern unification**) are used in some programming language implementations, such as lambdaProlog, as higher-order patterns are expressive, yet their associated unification procedure retains theoretical properties closer to first-order unification.

^ Common formal definitions

Prerequisites

Formally, a unification approach presupposes

- An infinite set \mathcal{V} of **variables**. For higher-order unification, it is convenient to choose \mathcal{V} disjoint from the set of [lambda-term bound variables](#).
- A set \mathcal{T} of **terms** such that $\mathcal{V} \subseteq \mathcal{T}$. For first-order unification and higher-order unification, \mathcal{T} is usually the set of [first-order terms](#) (terms built from variable and function symbols) and [lambda terms](#) (terms containing some higher-order variables), respectively.
- A mapping $\text{vars} : \mathcal{T} \rightarrow \mathcal{P}(\mathcal{V})$, assigning to each term t the set $\text{vars}(t)$ of **free variables** occurring in t .
- An [equivalence relation](#) \equiv on \mathcal{T} , indicating which terms are considered equal. For higher-order unification, usually $t \equiv s$ if t and s are [alpha equivalent](#). For first-order E-unification, \equiv reflects the background knowledge about certain function symbols; for example, if comm is considered commutative, $\text{comm}(x, y) \equiv \text{comm}(y, x)$ if comm results from $\text{comm}(x, y)$ by swapping the arguments of comm at some (possibly all) occurrences. ^[note 2] If there is no background knowledge at all, then only literally, or syntactically, identical terms are considered equal; in this case, \equiv is called the **free theory** (because it is a [free object](#)), the **empty theory** (because the set of equational [sentences](#), or the background knowledge, is empty), the **theory of uninterpreted functions** (because unification is done on uninterpreted [terms](#)), or the **theory of constructors** (because all function symbols just build up data terms, rather than operating on them).

First-order term

Main article: [Term \(logic\)](#)

Given a set \mathcal{V} of variable symbols, a set \mathcal{C} of constant symbols and sets \mathcal{F}_n of n -ary function symbols, also called operator symbols, for each natural number n , the set of (unsorted first-order) terms \mathcal{T} is [recursively defined](#) to be the smallest set with the following properties:^[5]

- every variable symbol is a term: $v \in \mathcal{T}$,
- every constant symbol is a term: $c \in \mathcal{T}$,
- from every n terms t_1, \dots, t_n , and every n -ary function symbol $f \in \mathcal{F}_n$, a larger term $f(t_1, \dots, t_n) \in \mathcal{T}$ can be built.

For example, if x is a variable symbol, a is a constant symbol, and f is a binary function symbol, then $f(x, a)$, and (hence) $f(x, a, b)$ by the first, second, and third term building rule, respectively. The latter term is usually written as $f(x, a, b)$, using [infix notation](#) and the more common operator symbol $+$ for convenience.

Higher-order term



Main article: [Lambda calculus](#)

Substitution



Main article: [Substitution \(logic\)](#)

A **substitution** is a mapping from variables to terms; the notation σ refers to a substitution mapping each variable x to the term $\sigma(x)$, for $x \in V$, and every other variable to itself. **Applying** that substitution to a term t is written in [postfix notation](#) as $t\sigma$; it means to (simultaneously) replace every occurrence of each variable x in the term t by $\sigma(x)$. The result $t\sigma$ of applying a substitution σ to a term t is called an **instance** of that term t . As a first-order example, applying the substitution $\{ x \mapsto h(a, y), z \mapsto b \}$ to the term

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

yields

$$f(h(a, y), a, g(b), y).$$

Generalization, specialization



If a term t has an instance equivalent to a term u , that is, if $t\sigma \equiv u$ for some substitution σ , then t is called **more general** than u , and u is called **more special** than, or **subsumed** by, t . For example, $x \oplus a$ is more general than $a \oplus b$ if \oplus is [commutative](#), since then $(x \oplus a)\{x \mapsto b\} = b \oplus a \equiv a \oplus b$.

If \equiv is literal (syntactic) identity of terms, a term may be both more general and more special than another one only if both terms differ just in their variable names, not in their syntactic structure; such terms are called **variants**, or **renamings** of each other. For example, $f(x_1, a, g(z_1), y_1)$ is a variant of $f(x_2, a, g(z_2), y_2)$, since

$$f(x_1, a, g(z_1), y_1)\{x_1 \mapsto x_2, y_1 \mapsto y_2, z_1 \mapsto z_2\} = f(x_2, a, g(z_2), y_2)$$

and

$$f(x_2, a, g(z_2), y_2)\{x_2 \mapsto x_1, y_2 \mapsto y_1, z_2 \mapsto z_1\} = f(x_1, a, g(z_1), y_1).$$

However, $f(x_1, a, g(z_1), y_1)$ is *not* a variant of $f(x_2, a, g(x_2), x_2)$, since no substitution can transform the latter term into the former one. The latter term is therefore properly more special than the former one.

For arbitrary \equiv , a term may be both more general and more special than a structurally different term. For example, if \oplus is [idempotent](#), that is, if always $x \oplus x \equiv x$, then the term $x \oplus y$ is more general than z ,^[note 3] and vice versa,^[note 4] although $x \oplus y$ and z are of different structure.

A substitution σ is **more special** than, or **subsumed** by, a substitution τ if $t\sigma$ is more special than $t\tau$ for each term t . We also say that τ is more general than σ . For instance $\{x \mapsto a, y \mapsto a\}$ is more special than $\tau = \{x \mapsto y\}$, but $\sigma = \{x \mapsto a\}$ is not, as $f(x, y)\sigma = f(a, y)$ is not more special than $f(x, y)\tau = f(y, y)$.^[6]

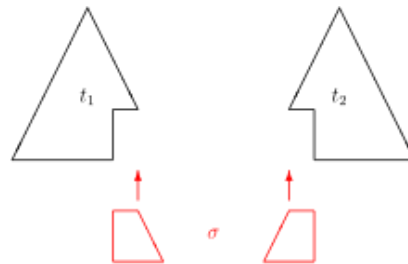
Unification problem, solution set



A **unification problem** is a finite set $\{l_1 \doteq r_1, \dots, l_n \doteq r_n\}$ of potential equations, where $l_i, r_i \in T$. A substitution σ is a **solution** of that problem if $l_i\sigma \doteq r_i\sigma$ for $i = 1, \dots, n$. Such a substitution is also called a **unifier** of the unification problem. For example, if \oplus is [associative](#), the unification problem $\{x \oplus a \doteq a \oplus x\}$ has the solutions $\{x \mapsto a\}$, $\{x \mapsto a \oplus a\}$, $\{x \mapsto a \oplus a \oplus a\}$, etc., while the problem $\{x \oplus a \doteq a\}$ has no solution.

For a given unification problem, a set S of unifiers is called **complete** if each solution substitution is subsumed by some substitution $\sigma \in S$; the set S is called **minimal** if none of its members subsumes another one.

^ Syntactic unification of first-order terms



Schematic triangle diagram of syntactically unifying terms t_1 and t_2 by a substitution σ

Syntactic unification of first-order terms is the most widely used unification framework. It is based on T being the set of *first-order terms* (over some given set V of variables, C of constants and F_n of n -ary function symbols) and on \equiv being *syntactic equality*. In this framework, each solvable unification problem $\{l_1 \doteq r_1, \dots, l_n \doteq r_n\}$ has a complete, and obviously minimal, [singleton](#) solution set $\{\sigma\}$. Its member σ is called the **most general unifier (mgu)** of the problem. The terms on the left and the right hand side of each potential equation become syntactically equal when the mgu is applied i.e. $l_1\sigma = r_1\sigma \wedge \dots \wedge l_n\sigma = r_n\sigma$. Any unifier of the problem is subsumed^[note 5] by the mgu σ . The mgu is unique up to variants: if S_1 and S_2 are both complete and minimal solution sets of the same syntactical unification problem, then $S_1 = \{\sigma_1\}$ and $S_2 = \{\sigma_2\}$ for some substitutions σ_1 and σ_2 , and $x\sigma_1$ is a variant of $x\sigma_2$ for each variable x occurring in the problem.

For example, the unification problem $\{x \doteq z, y \doteq f(x)\}$ has a unifier $\{x \mapsto z, y \mapsto f(z)\}$, because

$x \{ x \mapsto z, y \mapsto f(z) \} = z = z \{ x \mapsto z, y \mapsto f(z) \}$, and
 $y \{ x \mapsto z, y \mapsto f(z) \} = f(z) = f(x) \{ x \mapsto z, y \mapsto f(z) \}$.

This is also the most general unifier. Other unifiers for the same problem are e.g. $\{ x \mapsto f(x_1), y \mapsto f(f(x_1)), z \mapsto f(x_1) \}$, $\{ x \mapsto f(f(x_1)), y \mapsto f(f(f(x_1))), z \mapsto f(f(x_1)) \}$, and so on; there are infinitely many similar unifiers.

As another example, the problem $g(x, x) \doteq f(y)$ has no solution with respect to \equiv being literal identity, since any substitution applied to the left and right hand side will keep the outermost g and f , respectively, and terms with different outermost function symbols are syntactically different.

A unification algorithm



Robinson's 1965 unification algorithm

Symbols are ordered such that variables precede function symbols. Terms are ordered by increasing written length; equally long terms are ordered [lexicographically](#).^[7] For a set T of terms, its disagreement path p is the lexicographically least path where two member terms of T differ. Its disagreement set is the set of [subterms starting at \$p\$](#) , formally: $\{ t_p : t \in T \}$.^[8]

Algorithm:^[9]

```

Given a set  $T$  of terms to be unified
Let  $\sigma$  initially be the identity substitution

do forever
  if  $T_\sigma$  is a singleton set then
    return  $\sigma$ 
  fi

  let  $D$  be the disagreement set of  $T_\sigma$ 
  let  $s, t$  be the two lexicographically least terms in  $D$ 

  if  $s$  is not a variable or  $s$  occurs in  $t$  then
    return "NONUNIFIABLE"
  fi

   $\sigma := \sigma \{ s \mapsto t \}$ 
done
  
```

The first algorithm given by Robinson (1965) was rather inefficient; cf. box. The following faster algorithm originated from Martelli, Montanari (1982).^[10] This paper also lists preceding attempts to find an efficient syntactical unification algorithm,^{[11][12][13][14][15][16]} and states that linear-time algorithms were discovered independently by Martelli, Montanari (1976)^[13] and Paterson, Wegman (1978).^{[14][17]}

Given a finite set $G = \{ s_1 \doteq t_1, \dots, s_n \doteq t_n \}$ of potential equations, the algorithm applies rules to transform it to an equivalent set of equations of the form $\{ x_1 \doteq u_1, \dots, x_m \doteq u_m \}$ where x_1, \dots, x_m are distinct

variables and u_1, \dots, u_m are terms containing none of the x_i . A set of this form can be read as a substitution. If there is no solution the algorithm terminates with \perp ; other authors use " Ω ", " $\{\}$ ", or "*fail*" in that case. The operation of substituting all occurrences of variable x in problem G with term t is denoted $G\{x \mapsto t\}$. For simplicity, constant symbols are regarded as function symbols having zero arguments.

$$\begin{aligned}
 G \cup \{t \doteq t\} &\Rightarrow G \\
 G \cup \{f(s_0, \dots, s_k) \doteq f(t_0, \dots, t_k)\} &\Rightarrow G \cup \{s_0 \doteq t_0, \dots, s_k \doteq t_k\} \\
 G \cup \{f(s_0, \dots, s_k) \doteq g(t_0, \dots, t_m)\} &\Rightarrow \perp && \text{if } f \neq g \text{ or } k \neq m \\
 G \cup \{f(s_0, \dots, s_k) \doteq x\} &\Rightarrow G \cup \{x \doteq f(s_0, \dots, s_k)\} \\
 G \cup \{x \doteq t\} &\Rightarrow G\{x \mapsto t\} \cup \{x \doteq t\} && \text{if } x \notin \text{vars}(t) \text{ and } x \in \text{vars}(G) \\
 G \cup \{x \doteq f(s_0, \dots, s_k)\} &\Rightarrow \perp && \text{if } x \in \text{vars}(f(s_0, \dots, s_k))
 \end{aligned}$$

Occurs check

Main article: [Occurs check](#)

An attempt to unify a variable x with a term containing x as a strict subterm $x \doteq f(\dots, x, \dots)$ would lead to an infinite term as solution for x , since x would occur as a subterm of itself. In the set of (finite) first-order terms as defined above, the equation $x \doteq f(\dots, x, \dots)$ has no solution; hence the *eliminate* rule may only be applied if $x \notin \text{vars}(t)$. Since that additional check, called *occurs check*, slows down the algorithm, it is omitted e.g. in most Prolog systems. From a theoretical point of view, omitting the check amounts to solving equations over infinite trees, see [below](#).

Proof of termination

For the proof of termination of the algorithm consider a triple $\langle n_{var}, n_{lhs}, n_{eqn} \rangle$ where n_{var} is the number of variables that occur more than once in the equation set, n_{lhs} is the number of function symbols and constants on the left hand sides of potential equations, and n_{eqn} is the number of equations. When rule *eliminate* is applied, n_{var} decreases, since x is eliminated from G and kept only in $\{x \doteq t\}$. Applying any other rule can never increase n_{var} again. When rule *decompose*, *conflict*, or *swap* is applied, n_{lhs} decreases, since at least the left hand side's outermost f disappears. Applying any of the remaining rules *delete* or *check* can't increase n_{lhs} , but decreases n_{eqn} . Hence, any rule application decreases the triple $\langle n_{var}, n_{lhs}, n_{eqn} \rangle$ with respect to the [lexicographical order](#), which is possible only a finite number of times.

Conor McBride observes^[18] that “by expressing the structure which unification exploits” in a [dependently typed](#) language such as [Epigram](#), [Robinson's](#) algorithm can be made [recursive on the number of variables](#), in which case a separate termination proof becomes unnecessary.

Examples of syntactic unification of first-order terms

In the Prolog syntactical convention a symbol starting with an upper case letter is a variable name; a symbol that starts with a lowercase letter is a function symbol; the comma is used as the logical *and* operator. For mathematical notation, x,y,z are used as variables, f,g as function symbols, and a,b as constants.

Prolog notation	Mathematical notation	Unifying substitution	Explanation
<code>a = a</code>	$\{ a = a \}$	$\{ \}$	Succeeds. (tautology)
<code>a = b</code>	$\{ a = b \}$	\perp	a and b do not match
<code>X = X</code>	$\{ x = x \}$	$\{ \}$	Succeeds. (tautology)
<code>a = X</code>	$\{ a = x \}$	$\{ x \mapsto a \}$	x is unified with the constant a
<code>X = Y</code>	$\{ x = y \}$	$\{ x \mapsto y \}$	x and y are aliased
<code>f(a, X) = f(a, b)</code>	$\{ f(a, x) = f(a, b) \}$	$\{ x \mapsto b \}$	function and constant symbols match, x is unified with the constant b
<code>f(a) = g(a)</code>	$\{ f(a) = g(a) \}$	\perp	f and g do not match
<code>f(X) = f(Y)</code>	$\{ f(x) = f(y) \}$	$\{ x \mapsto y \}$	x and y are aliased
<code>f(X) = g(Y)</code>	$\{ f(x) = g(y) \}$	\perp	f and g do not match
<code>f(X) = f(Y, Z)</code>	$\{ f(x) = f(y, z) \}$	\perp	Fails. The f function symbols have different arity
<code>f(g(X)) = f(Y)</code>	$\{ f(g(x)) = f(y) \}$	$\{ y \mapsto g(x) \}$	Unifies y with the term <code>g(x)</code>
<code>f(g(X), X) = f(Y, a)</code>	$\{ f(g(x), x) = f(y, a) \}$	$\{ x \mapsto a, y \mapsto g(a) \}$	Unifies x with constant a , and y with the term <code>g(a)</code>
<code>X = f(X)</code>	$\{ x = f(x) \}$	should be \perp	Returns \perp in first-order logic and many modern Prolog dialects (enforced by the occurs check). Succeeds in traditional Prolog and in Prolog II, unifying x with infinite term <code>x=f(f(f(f(...))))</code> .
<code>X = Y, Y = a</code>	$\{ x = y, y = a \}$	$\{ x \mapsto a, y \mapsto a \}$	Both x and y are unified with the constant a
<code>a = Y, X = Y</code>	$\{ a = y, x = y \}$	$\{ x \mapsto a, y \mapsto a \}$	As above (order of equations in set doesn't matter)
<code>X = a, b = X</code>	$\{ x = a, b = x \}$	\perp	Fails. a and b do not match, so x can't be unified with both

Two terms with an exponentially larger tree for their least common instance. Its [dag](#) representation (rightmost, orange part) is still of linear size.

The most general unifier of a syntactic first-order unification problem of [size](#) n may have a size of 2^n . For example, the problem `x = f(x, f(x, f(x, ...)))` has the most general unifier `{x = f(x, f(x, f(x, ...)))}`, cf. picture. In order to avoid exponential time complexity caused by such blow-up, advanced unification algorithms work on [directed acyclic graphs](#) (dags) rather than trees.^[19]

Application: unification in logic programming

The concept of unification is one of the main ideas behind [logic programming](#), best known through the language [Prolog](#). It represents the mechanism of binding the contents of variables and can be viewed as a kind of one-time assignment. In Prolog, this operation is denoted by the equality symbol `=`, but is also done when instantiating variables (see below). It is also used in other languages by the use of the equality symbol `=`, but also in conjunction with many operations including `+`, `-`, `*`, `/`. [Type inference](#) algorithms are typically based on unification.

In Prolog:

1. A [variable](#) which is uninstantiated—i.e. no previous unifications were performed on it—can be unified with an atom, a term, or another uninstantiated variable, thus effectively becoming its alias. In many modern Prolog dialects and in [first-order logic](#), a variable cannot be unified with a term that contains it; this is the so-called [occurs check](#).
2. Two atoms can only be unified if they are identical.
3. Similarly, a term can be unified with another term if the top function symbols and [arities](#) of the terms are identical and if the parameters can be unified simultaneously. Note that this is a recursive behavior.

Application: type inference

Unification is used during type inference, for instance in the functional programming language [Haskell](#). On one hand, the programmer does not need to provide type information for every function, on the other hand it is used to detect typing errors. The Haskell expression `True : ['a', 'b', 'c']` is not correctly typed. The list construction function `(:)` is of type `a -> [a] -> [a]`, and for the first argument `True` the polymorphic type variable `a` has to be unified with `True`'s type, `Bool`. The second argument, `['a', 'b', 'c']`, is of type `[Char]`, but `a` cannot be both `Bool` and `Char` at the same time.

Like for Prolog, an algorithm for type inference can be given:

1. Any type variable unifies with any type expression, and is instantiated to that expression. A specific theory might restrict this rule with an occurs check.
2. Two type constants unify only if they are the same type.
3. Two type constructions unify only if they are applications of the same type constructor and all of their component types recursively unify.

Due to its declarative nature, the order in a sequence of unifications is (usually) unimportant.

Note that in the terminology of [first-order logic](#), an atom is a basic proposition and is unified similarly to a Prolog term.

^ Order-sorted unification



[Order-sorted logic](#) allows one to assign a *sort*, or *type*, to each term, and to declare a sort s_1 a *subsort* of another sort s_2 , commonly written as $s_1 \subseteq s_2$. For example, when reasoning about biological creatures, it is useful to declare a sort *dog* to be a subsort of a sort *animal*. Wherever a term of some sort s is required, a term of any subsort of s may be supplied instead. For example, assuming a function declaration *mother*: $animal \rightarrow animal$, and a constant declaration *lassie*: *dog*, the term *mother(lassie)* is perfectly valid and has the sort *animal*. In order to supply the information that the mother of a dog is a dog in turn, another declaration *mother*: $dog \rightarrow dog$ may be issued; this is called *function overloading*, similar to [overloading in programming languages](#).

[Walther](#) gave a unification algorithm for terms in order-sorted logic, requiring for any two declared sorts s_1, s_2 their intersection $s_1 \cap s_2$ to be declared, too: if x_1 and x_2 is a variable of sort s_1 and s_2 , respectively, the equation $x_1 \doteq x_2$ has the solution $\{x_1 = x, x_2 = x\}$, where $x: s_1 \cap s_2$.^[20] After incorporating this algorithm into a clause-based automated theorem prover, he could solve a benchmark problem by translating it into order-sorted logic, thereby boiling it down an order of magnitude, as many unary predicates turned into sorts.

Smolka generalized order-sorted logic to allow for [parametric polymorphism](#).^[21] In his framework, subsort declarations are propagated to complex type expressions. As a programming example, a parametric sort $list(X)$ may be declared (with X being a type parameter as in a [C++ template](#)), and from a subsort declaration $int \subseteq float$ the relation $list(int) \subseteq list(float)$ is automatically inferred, meaning that each list of integers is also a list of floats.

Schmidt-Schauß generalized order-sorted logic to allow for term declarations.^[22] As an example, assuming subsort declarations $even \subseteq int$ and $odd \subseteq int$, a term declaration like $\forall i: int. (i + i) : even$ allows to declare a property of integer addition that could not be expressed by ordinary overloading.

^ Unification of infinite terms



Background on infinite trees:

- B. Courcelle (1983). "Fundamental Properties of Infinite Trees" [↗](#) (PDF). *Theoret. Comput. Sci.* **25** (2): 95–169. doi:10.1016/0304-3975(83)90059-2 [↗](#). Archived from [the original](#) [↗](#) (PDF) on 2014-04-21. Retrieved 2013-06-28.
- Michael J. Maher (Jul 1988). "Complete Axiomatizations of the Algebras of Finite, Rational and Infinite Trees". *Proc. IEEE 3rd Annual Symp. on Logic in Computer Science, Edinburgh*. pp. 348–357.
- Joxan Jaffar; Peter J. Stuckey (1986). "Semantics of Infinite Tree Logic Programming". *Theoretical Computer Science*. **46**: 141–158. doi:10.1016/0304-3975(86)90027-7 [↗](#).

Unification algorithm, Prolog II:

- A. Colmerauer (1982). K.L. Clark; S.-A. Tarnlund (eds.). *Prolog and Infinite Trees*. Academic Press.
- Alain Colmerauer (1984). "Equations and Inequations on Finite and Infinite Trees". In ICOT (ed.). *Proc. Int. Conf. on Fifth Generation Computer Systems*. pp. 85–99.

Applications:

- Francis Giannesini; Jacques Cohen (1984). "Parser Generation and Grammar Manipulation using Prolog's Infinite Trees" [↗](#). *J. Logic Programming*. **3** (3): 253–265. doi:10.1016/0743-1066(84)90013-X [↗](#).

^ E-unification

E-unification is the problem of finding solutions to a given set of [equations](#), taking into account some equational background knowledge E . The latter is given as a set of universal [equalities](#). For some particular sets E , equation solving [algorithms](#) (a.k.a. *E-unification algorithms*) have been devised; for others it has been proven that no such algorithms can exist.

For example, if a and b are distinct constants, the [equation](#) has no solution with respect to purely [syntactic unification](#), where nothing is known about the operator . However, if the is known to be [commutative](#), then the substitution $\{x \mapsto b, y \mapsto a\}$ solves the above equation, since

$$\begin{aligned}
 & \text{ } \{x \mapsto b, y \mapsto a\} \\
 = & \text{ } && \text{by substitution application} \\
 = & \text{ } && \text{by commutativity of } \\
 = & \text{ } \{x \mapsto b, y \mapsto a\} && \text{by (converse) substitution application}
 \end{aligned}$$

The background knowledge E could state the commutativity of by the universal equality " " for all u, v ".

Particular background knowledge sets E

Used naming conventions

$\forall u, v, w:$	$\square = \square$	A	Associativity of \square
$\forall u, v:$	$\square = \square$	C	Commutativity of \square
$\forall u, v, w:$	$\square = \square$	D_l	Left distributivity of \square over \square
$\forall u, v, w:$	$\square = \square$	D_r	Right distributivity of \square over \square
$\forall u:$	$\square = u$	I	Idempotence of \square
$\forall u:$	$\square = u$	N_l	Left neutral element n with respect to \square
$\forall u:$	$\square = u$	N_r	Right neutral element n with respect to \square

It is said that *unification is decidable* for a theory, if a unification algorithm has been devised for it that terminates for *any* input problem. It is said that *unification is semi-decidable* for a theory, if a unification algorithm has been devised for it that terminates for any *solvable* input problem, but may keep searching forever for solutions of an unsolvable input problem.

Unification is decidable for the following theories:

- A ^[23]
- A, C ^[24]
- A, C, I ^[25]
- A, C, N_l ^{[note 7][25]}
- A, I ^[26]
- A, N_l, N_r (monoid)^[27]
- C ^[28]
- Boolean rings^{[29][30]}
- Abelian groups, even if the signature is expanded by arbitrary additional symbols (but not axioms)^[31]
- K4 modal algebras^[32]

Unification is semi-decidable for the following theories:

- A, D_l, D_r ^[33]
- A, C, D_l ^{[note 7][34]}
- Commutative rings^[31]

One-sided paramodulation



If there is a [convergent term rewriting system](#) R available for E , the **one-sided paramodulation** algorithm^[35] can be used to enumerate all solutions of given equations.

One-sided paramodulation rules

$$G \cup \{ f(s_1, \dots, s_n) \doteq; \Rightarrow G \cup \{ s_1 \doteq t_1, \dots, s_n \doteq t_n \}; S$$

$f(t_1, \dots, t_n) \} S$			decompose
$G \cup \{ x \doteq t \} ; S \Rightarrow$	$G \{ x \mapsto t \} ; S \{ x \mapsto t \} \cup$	if the variable x doesn't occur in t	eliminate
$G \cup \{ f(s_1, \dots, s_n) \doteq t \} ; S \Rightarrow$	$G \cup \{ s_1 \doteq u_1, \dots, s_n \doteq u_n, r \doteq t \} ; S$	if $f(u_1, \dots, u_n) \rightarrow r$ is a rule from R	mutate
$G \cup \{ f(s_1, \dots, s_n) \doteq y \} ; S \Rightarrow$	$G \cup \{ s_1 \doteq y_1, \dots, s_n \doteq y_n, y \doteq f(y_1, \dots, y_n) \} ; S$	if y_1, \dots, y_n are new variables	imitate

Starting with G being the unification problem to be solved and S being the identity substitution, rules are applied nondeterministically until the empty set appears as the actual G , in which case the actual S is a unifying substitution. Depending on the order the paramodulation rules are applied, on the choice of the actual equation from G , and on the choice of R 's rules in *mutate*, different computations paths are possible. Only some lead to a solution, while others end at a $G \neq \{\}$ where no further rule is applicable (e.g. $G = \{ f(\dots) \doteq g(\dots) \}$).

Example term rewrite system R

- 1 $app(nil, z) \rightarrow z$
- 2 $app(x.y, z) \rightarrow x.app(y, z)$

For an example, a term rewrite system R is used defining the *append* operator of lists built from *cons* and *nil*; where $cons(x, y)$ is written in infix notation as $x.y$ for brevity; e.g. $app(a.b.nil, c.d.nil) \rightarrow a.app(b.nil, c.d.nil) \rightarrow a.b.app(nil, c.d.nil) \rightarrow a.b.c.d.nil$ demonstrates the concatenation of the lists $a.b.nil$ and $c.d.nil$, employing the rewrite rule 2, 2, and 1. The equational theory E corresponding to R is the [congruence closure](#) of R , both viewed as binary relations on terms. For example, $app(a.b.nil, c.d.nil) \equiv a.b.c.d.nil \equiv app(a.b.c.d.nil, nil)$. The paramodulation algorithm enumerates solutions to equations with respect to that E when fed with the example R .

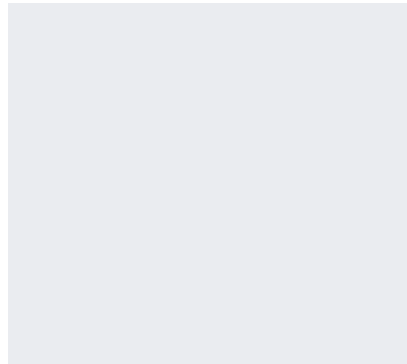
A successful example computation path for the unification problem $\{ app(x, app(y, x)) \doteq a.a.nil \}$ is shown below. To avoid variable name clashes, rewrite rules are consistently renamed each time before their use by rule *mutate*; v_2, v_3, \dots are computer-generated variable names for this purpose. In each line, the chosen equation from G is highlighted in red. Each time the *mutate* rule is applied, the chosen rewrite rule (1 or 2) is indicated in parentheses. From the last line, the unifying substitution $S = \{ y \mapsto nil, x \mapsto a.nil \}$ can be obtained. In fact, $app(x, app(y, x)) \{ y \mapsto nil, x \mapsto a.nil \} = app(a.nil, app(nil, a.nil)) \equiv app(a.nil, a.nil) \equiv a.app(nil, a.nil) \equiv a.a.nil$ solves the given problem. A second successful computation path, obtainable by choosing "mutate(1), mutate(2), mutate(2), mutate(1)" leads to the substitution $S = \{ y \mapsto a.a.nil, x \mapsto nil \}$; it is not shown here. No other path leads to a success.

Example unifier computation

Used rule		G	S
		$\{ app(x, app(y, x)) \doteq a.a.nil \}$	$\{\}$
mutate(2)	\Rightarrow	$\{ x \doteq v_2.v_3, app(y, x) \doteq v_4, v_2.app(v_3, v_4) \doteq a.a.nil \}$	$\{\}$
decompose	\Rightarrow	$\{ x \doteq v_2.v_3, app(y, x) \doteq v_4, v_2 \doteq a, app(v_3, v_4) \doteq a.nil \}$	$\{\}$

eliminate	\Rightarrow	$\{ \text{app}(y, v_2.v_3) \doteq v_4, \textcolor{red}{v_2} \doteq \textcolor{red}{a}, \text{app}(v_3, v_4) \doteq a.\text{nil} \}$	$\{ x \mapsto v_2.v_3 \}$
eliminate	\Rightarrow	$\{ \textcolor{red}{\text{app}(y, a.v_3)} \doteq v_4, \text{app}(v_3, v_4) \doteq a.\text{nil} \}$	$\{ x \mapsto a.v_3 \}$
mutate(1)	\Rightarrow	$\{ y \doteq \text{nil}, a.v_3 \doteq v_5, \textcolor{red}{v_5} \doteq \textcolor{red}{v_4}, \text{app}(v_3, v_4) \doteq a.\text{nil} \}$	$\{ x \mapsto a.v_3 \}$
eliminate	\Rightarrow	$\{ \textcolor{red}{y} \doteq \textcolor{red}{\text{nil}}, a.v_3 \doteq v_4, \text{app}(v_3, v_4) \doteq a.\text{nil} \}$	$\{ x \mapsto a.v_3 \}$
eliminate	\Rightarrow	$\{ a.v_3 \doteq v_4, \textcolor{red}{\text{app}(v_3, v_4)} \doteq \textcolor{red}{a.\text{nil}} \}$	$\{ y \mapsto \text{nil}, x \mapsto a.v_3 \}$
mutate(1)	\Rightarrow	$\{ a.v_3 \doteq v_4, v_3 \doteq \text{nil}, \textcolor{red}{v_4} \doteq \textcolor{red}{v_6}, v_6 \doteq a.\text{nil} \}$	$\{ y \mapsto \text{nil}, x \mapsto a.v_3 \}$
eliminate	\Rightarrow	$\{ a.v_3 \doteq v_4, \textcolor{red}{v_3} \doteq \textcolor{red}{\text{nil}}, v_4 \doteq a.\text{nil} \}$	$\{ y \mapsto \text{nil}, x \mapsto a.v_3 \}$
eliminate	\Rightarrow	$\{ a.\text{nil} \doteq v_4, \textcolor{red}{v_4} \doteq \textcolor{red}{a.\text{nil}} \}$	$\{ y \mapsto \text{nil}, x \mapsto a.\text{nil} \}$
eliminate	\Rightarrow	$\{ \textcolor{red}{a.\text{nil}} \doteq \textcolor{red}{a.\text{nil}} \}$	$\{ y \mapsto \text{nil}, x \mapsto a.\text{nil} \}$
decompose	\Rightarrow	$\{ \textcolor{red}{a} \doteq \textcolor{red}{a}, \text{nil} \doteq \text{nil} \}$	$\{ y \mapsto \text{nil}, x \mapsto a.\text{nil} \}$
decompose	\Rightarrow	$\{ \textcolor{red}{\text{nil}} \doteq \textcolor{red}{\text{nil}} \}$	$\{ y \mapsto \text{nil}, x \mapsto a.\text{nil} \}$
decompose	\Rightarrow	$\{ \}$	$\{ y \mapsto \text{nil}, x \mapsto a.\text{nil} \}$

Narrowing



Triangle diagram of narrowing step $s \rightsquigarrow t$ at position p in term s , with unifying substitution σ (bottom row), using a rewrite rule $l \rightarrow r$ (top row)

If R is a **convergent term rewriting system** for E , an approach alternative to the previous section consists in successive application of "**narrowing steps**"; this will eventually enumerate all solutions of a given equation. A narrowing step (cf. picture) consists in

- choosing a nonvariable subterm of the current term,
- **syntactically unifying** it with the left hand side of a rule from R , and
- replacing the instantiated rule's right hand side into the instantiated term.

Formally, if $l \rightarrow r$ is a **renamed copy** of a rewrite rule from R , having no variables in common with a term s , and the **subterm** $s|_p$ is not a variable and is unifiable with l via the **mgu** σ , then s can be **narrowed** to the term $t = s\sigma[r\sigma]_p$, i.e. to the term $s\sigma$, with the subterm at p **replaced** by $r\sigma$. The situation that s can be narrowed to t is commonly denoted as $s \rightsquigarrow t$. Intuitively, a sequence of narrowing steps $t_1 \rightsquigarrow t_2 \rightsquigarrow \dots \rightsquigarrow t_n$ can be thought of as a

sequence of rewrite steps $t_1 \rightarrow t_2 \rightarrow \dots \rightarrow t_n$, but with the initial term t_1 being further and further instantiated, as necessary to make each of the used rules applicable.

The [above](#) example paramodulation computation corresponds to the following narrowing sequence ("↓" indicating instantiation here):

$$\begin{array}{ccc}
 \text{app}(x, \text{app}(y, x)) & & \\
 \downarrow \quad \downarrow & & \\
 \text{app}(v_2.v_3, \text{app}(y, v_2.v_3)) \rightarrow v_2.\text{app}(v_3, \text{app}(y, v_2.v_3)) & & x \mapsto v_2.v_3 \\
 \downarrow & & y \mapsto \text{nil} \\
 v_2.\text{app}(v_3, \text{app}(\text{nil}, v_2.v_3)) \rightarrow v_2.\text{app}(v_3, v_2.v_3) & & \\
 \downarrow \quad \downarrow & & v_3 \mapsto \text{nil} \\
 v_2.\text{app}(\text{nil}, v_2.\text{nil}) \rightarrow v_2.v_2.\text{nil} & &
 \end{array}$$

The last term, $v_2.v_2.\text{nil}$ can be syntactically unified with the original right hand side term $a.a.\text{nil}$.

The *narrowing lemma*^[36] ensures that whenever an instance of a term s can be rewritten to a term t by a convergent term rewriting system, then s and t can be narrowed and rewritten to a term s' and t' , respectively, such that t' is an instance of s' .

Formally: whenever $s\sigma \rightarrow t$ holds for some substitution σ , then there exist terms s', t' such that $s \rightsquigarrow s'$ and $t \rightarrow t'$ and $s'\tau = t'$ for some substitution τ .

^ Higher-order unification



Many applications require one to consider the unification of typed lambda-terms instead of first-order terms. Such unification is often called *higher-order unification*. A well studied branch of higher-order unification is the problem of unifying simply typed lambda terms modulo the equality determined by $\alpha\beta\eta$ conversions. Such unification problems do not have most general unifiers. While higher-order unification is [undecidable](#),^{[37][38][39]} [Gérard Huet](#) gave a [semi-decidable](#) (pre-)unification algorithm^[40] that allows a systematic search of the space of unifiers (generalizing the unification algorithm of Martelli-Montanari^[2] with rules for terms containing higher-order variables) that seems to work sufficiently well in practice. Huet^[41] and Gilles Dowek^[42] have written articles surveying this topic.

[Dale Miller](#) has described what is now called *higher-order pattern unification*.^[43] This subset of higher-order unification is decidable and solvable unification problems have most-general unifiers. Many computer systems that contain higher-order unification, such as the higher-order logic programming languages [λProlog](#) and [Twelf](#), often implement only the pattern fragment and not full higher-order unification.

In computational linguistics, one of the most influential theories of [ellipsis](#) is that ellipses are represented by free variables whose values are then determined using Higher-Order Unification (HOU). For instance, the semantic representation of "Jon likes Mary and Peter does too" is $\text{like}(j, m) \wedge R(p)$ and the value of R (the semantic

representation of the ellipsis) is determined by the equation $\text{like}(j, m) = R(j)$. The process of solving such equations is called Higher-Order Unification.^[44]

For example, the unification problem $\{ f(a, b, a) \doteq d(b, a, c) \}$, where the only variable is f , has the solutions $\{ f \mapsto \lambda x. \lambda y. \lambda z. d(y, x, c) \}$, $\{ f \mapsto \lambda x. \lambda y. \lambda z. d(y, z, c) \}$, $\{ f \mapsto \lambda x. \lambda y. \lambda z. d(y, a, c) \}$, $\{ f \mapsto \lambda x. \lambda y. \lambda z. d(b, x, c) \}$, $\{ f \mapsto \lambda x. \lambda y. \lambda z. d(b, z, c) \}$ and $\{ f \mapsto \lambda x. \lambda y. \lambda z. d(b, a, c) \}$.

Wayne Snyder gave a generalization of both higher-order unification and E-unification, i.e. an algorithm to unify lambda-terms modulo an equational theory.^[45]

^ See also



- [Rewriting](#)
- [Admissible rule](#)
- [Explicit substitution](#) in [lambda calculus](#)
- Mathematical [equation solving](#)
- [Dis-unification](#): solving inequations between symbolic expression
- [Anti-unification](#): computing a least general generalization (lgg) of two terms, dual to computing a most general instance (mgu)
- [Ontology alignment](#) (use *unification* with [semantic equivalence](#))

^ Notes
















1. ^ in this case, still a complete substitution set exists (e.g. the set of all solutions at all); however, each such set contains redundant members.
2. ^ E.g. $a \oplus (b \oplus f(x)) \equiv a \oplus (f(x) \oplus b) \equiv (b \oplus f(x)) \oplus a \equiv (f(x) \oplus b) \oplus a$
3. ^ since
4. ^ since $z \{ z \mapsto x \oplus y \} = x \oplus y$
5. ^ formally: each unifier τ satisfies $\forall x: x\tau = (x\sigma)_Q$ for some substitution ρ
6. ^ Although the rule keeps $\lambda \text{doteq} t$ in G , it cannot loop forever since its precondition $x \in \text{vars}(G)$ is invalidated by its first application. More generally, the algorithm is guaranteed to terminate always, see [below](#).
7. ^ [a b](#) in the presence of equality C , equalities N_l and N_r are equivalent, similar for D_l and D_r

^ References



1. ^ Fages, François; Huet, Gérard (1986). "Complete Sets of Unifiers and Matchers in Equational Theories".

Theoretical Computer Science. **43**: 189–200. doi:[10.1016/0304-3975\(86\)90175-1](https://doi.org/10.1016/0304-3975(86)90175-1) .







2. ^ [a b c](#) Martelli, Alberto; Montanari, Ugo (Apr 1982). "An Efficient Unification Algorithm". *ACM Trans. Program. Lang. Syst.* **4** (2): 258–282. doi:[10.1145/357162.357169](https://doi.org/10.1145/357162.357169) .
3. ^ [a b c d](#) J.A. Robinson (Jan 1965). "A Machine-Oriented Logic Based on the Resolution Principle". *Journal of the ACM*. **12** (1): 23–41. doi:[10.1145/321250.321253](https://doi.org/10.1145/321250.321253) ; Here: sect.5.8, p.32
4. ^ J.A. Robinson (1971). "[Computational logic: The unification computation](#)"  (PDF). *Machine Intelligence*. **6**: 63–72.
5. ^ [C.C. Chang](#); [H. Jerome Keisler](#) (1977). *Model Theory*. Studies in Logic and the Foundation of Mathematics. **73**. North Holland.; here: Sect.1.3
6. ^ K.R. Apt. "From Logic Programming to Prolog", p. 24. Prentice Hall, 1997.
7. ^ Robinson (1965);^[3] nr.2.5, 2.14, p.25
8. ^ Robinson (1965);^[3] nr.5.6, p.32
9. ^ Robinson (1965);^[3] nr.5.8, p.32
10. ^ Alg.1, p.261. Their rule **(a)** corresponds to rule **swap** here, **(b)** to **delete**, **(c)** to both **decompose** and **conflict**, and **(d)** to both **eliminate** and **check**.
11. ^ Lewis Denver Baxter (Feb 1976). [A practically linear unification algorithm](#)  (PDF) (Res. Report). CS-76-13. Univ. of Waterloo, Ontario.
12. ^ [Gérard Huet](#) (Sep 1976). *Resolution d'Equations dans des Langages d'Ordre 1,2,... ω* (These d'etat). Universite de Paris VII.
13. ^ [a b](#) Alberto Martelli & Ugo Montanari (Jul 1976). [Unification in linear time and space: A structured presentation](#)  (Internal Note). IEI-B76-16. Consiglio Nazionale delle Ricerche, Pisa.
14. ^ [a b c](#) [Michael Stewart Paterson](#) and M.N. Wegman (Apr 1978). "[Linear unification](#)"  (PDF). *J. Comput. Syst. Sci.* **16** (2): 158–167. doi:[10.1016/0022-0000\(78\)90043-0](https://doi.org/10.1016/0022-0000(78)90043-0) .
15. ^ [J.A. Robinson](#) (Jan 1976). "[Fast unification](#)" . In [Woodrow W. Bledsoe](#), Michael M. Richter (ed.). *Proc. Theorem Proving Workshop Oberwolfach*. Oberwolfach Workshop Report. 1976/3.
16. ^ M. Venturini-Zilli (Oct 1975). "Complexity of the unification algorithm for first-order expressions". *Calcolo*. **12** (4): 361–372. doi:[10.1007/BF02575754](https://doi.org/10.1007/BF02575754) .
17. ^ See Martelli, Montanari (1982),^[2] sect.1, p.259. Paterson's and Wegman's paper is dated 1978; however, the journal publisher received it in Sep.1976.
18. ^ McBride, Conor (October 2003). "[First-Order Unification by Structural Recursion](#)" . *Journal of Functional Programming*. **13** (6): 1061–1076. CiteSeerX [10.1.1.25.1516](https://citeseerx.ist.psu.edu/viewdoc/doi?doi=10.1.1.25.1516) . doi:[10.1017/S0956796803004957](https://doi.org/10.1017/S0956796803004957) . ISSN [0956-7968](#) . Retrieved 30 March 2012.
19. ^ e.g. Paterson, Wegman (1978),^[14] sect.2, p.159

20. [^] [Walther, Christoph](#) (1985). "A Mechanical Solution of Schubert's Steamroller by Many-Sorted Resolution" [↗](#) (PDF). *Artif. Intell.* **26** (2): 217–224. doi:[10.1016/0004-3702\(85\)90029-3](#) [↗](#).
21. [^] [Smolka, Gert](#) (Nov 1988). *Logic Programming with Polymorphically Order-Sorted Types* [↗](#) (PDF). Int. Workshop Algebraic and Logic Programming. LNCS. **343**. Springer. pp. 53–70. doi:[10.1007/3-540-50667-5_58](#) [↗](#).
22. [^] [Schmidt-Schauß, Manfred](#) (Apr 1988). *Computational Aspects of an Order-Sorted Logic with Term Declarations*. LNAI. **395**. Springer.
23. [^] [Gordon D. Plotkin](#), *Lattice Theoretic Properties of Subsumption*, Memorandum MIP-R-77, Univ. Edinburgh, Jun 1970
24. [^] [Mark E. Stickel](#), *A Unification Algorithm for Associative-Commutative Functions*, J. Assoc. Comput. Mach., vol.28, no.3, pp. 423–434, 1981
25. [^] [a](#) [b](#) [F. Fages](#), *Associative-Commutative Unification*, J. Symbolic Comput., vol.3, no.3, pp. 257–275, 1987
26. [^] [Franz Baader](#), *Unification in Idempotent Semigroups is of Type Zero*, J. Automat. Reasoning, vol.2, no.3, 1986
27. [^] [J. Makanin](#), *The Problem of Solvability of Equations in a Free Semi-Group*, Akad. Nauk SSSR, vol.233, no.2, 1977
28. [^] [F. Fages](#) (1987). "Associative-Commutative Unification". *J. Symbolic Comput.* **3** (3): 257–275. doi:[10.1016/s0747-7171\(87\)80004-4](#) [↗](#).
29. [^] [Martin, U.](#), [Nipkow, T.](#) (1986). "Unification in Boolean Rings". In [Jörg H. Siekmann](#) (ed.). *Proc. 8th CADE*. LNCS. **230**. Springer. pp. 506–513.
30. [^] [A. Boudet](#); [J.P. Jouannaud](#); [M. Schmidt-Schauß](#) (1989). "[Unification of Boolean Rings and Abelian Groups](#)" [↗](#) (PDF). *Journal of Symbolic Computation.* **8** (5): 449–477. doi:[10.1016/s0747-7171\(89\)80054-9](#) [↗](#).
31. [^] [a](#) [b](#) [Baader and Snyder](#) (2001), p. 486.
32. [^] [F. Baader and S. Ghilardi](#), *Unification in modal and description logics* [↗](#), Logic Journal of the IGPL 19 (2011), no. 6, pp. 705–730.
33. [^] [P. Szabo](#), *Unifikationstheorie erster Ordnung (First Order Unification Theory)*, Thesis, Univ. Karlsruhe, West Germany, 1982
34. [^] [Jörg H. Siekmann](#), *Universal Unification*, Proc. 7th Int. Conf. on Automated Deduction, Springer LNCS vol.170, pp. 1–42, 1984
35. [^] [N. Dershowitz and G. Sivakumar](#), *Solving Goals in Equational Languages*, Proc. 1st Int. Workshop on Conditional Term Rewriting Systems, Springer LNCS vol.308, pp. 45–55, 1988
36. [^] [Fay](#) (1979). "First-Order Unification in an Equational Theory". *Proc. 4th Workshop on Automated Deduction*. pp. 161–167.

37. [^] Warren D. Goldfarb (1981). "The Undecidability of the Second-Order Unification Problem"  (PDF). *TCS*. **13** (2): 225–230. doi:10.1016/0304-3975(81)90040-2 .
38. [^] Gérard P. Huet (1973). "The Undecidability of Unification in Third Order Logic"  (PDF). *Information and Control*. **22** (3): 257–267. doi:10.1016/S0019-9958(73)90301-X .
39. [^] Claudio Lucchesi: The Undecidability of the Unification Problem for Third Order Languages (Research Report CSRR 2059; Department of Computer Science, University of Waterloo, 1972)
40. [^] Gérard Huet: A Unification Algorithm for typed Lambda-Calculus 
41. [^] Gérard Huet: Higher Order Unification 30 Years Later 
42. [^] Gilles Dowek: Higher-Order Unification and Matching. Handbook of Automated Reasoning 2001: 1009–1062
43. [^] Miller, Dale (1991). "A Logic Programming Language with Lambda-Abstraction, Function Variables, and Simple Unification"  (PDF). *Journal of Logic and Computation*. **1** (4): 497–536. doi:10.1093/logcom/1.4.497 .
44. [^] Gardent, Claire; Kohlhase, Michael; Konrad, Karsten (1997). "A Multi-Level, Higher-Order Unification Approach to Ellipsis". *Submitted to European Association for Computational Linguistics (EACL)*. CiteSeerX 10.1.1.55.9018 .
45. [^] Wayne Snyder (Jul 1990). "Higher order E-unification". *Proc. 10th Conference on Automated Deduction*. LNAI. **449**. Springer. pp. 573–587.

[^] Further reading



- Franz Baader and Wayne Snyder (2001). "Unification Theory" . In John Alan Robinson and Andrei Voronkov, editors, *Handbook of Automated Reasoning*, volume I, pages 447–533. Elsevier Science Publishers.
- Gilles Dowek (2001). "Higher-order Unification and Matching" . In *Handbook of Automated Reasoning*.
- Franz Baader and Tobias Nipkow (1998). *Term Rewriting and All That* . Cambridge University Press.
- Franz Baader and Jörg H. Siekmann [de] (1993). "Unification Theory". In *Handbook of Logic in Artificial Intelligence and Logic Programming*.
- Jean-Pierre Jouannaud and Claude Kirchner (1991). "Solving Equations in Abstract Algebras: A Rule-Based Survey of Unification". In *Computational Logic: Essays in Honor of Alan Robinson*.
- Nachum Dershowitz and Jean-Pierre Jouannaud, *Rewrite Systems*, in: Jan van Leeuwen (ed.), *Handbook of Theoretical Computer Science*, volume B *Formal Models and Semantics*, Elsevier, 1990, pp. 243–320
- Jörg H. Siekmann (1990). "Unification Theory". In Claude Kirchner (editor) *Unification*. Academic Press.
- Kevin Knight (Mar 1989). "Unification: A Multidisciplinary Survey"  (PDF). *ACM Computing Surveys*. **21** (1): 93–124. CiteSeerX 10.1.1.64.8967 . doi:10.1145/62029.62030 .

- [Gérard Huet](#) and [Derek C. Oppen](#) (1980). "[Equations and Rewrite Rules: A Survey](#)". Technical report. Stanford University.
- Raulefs, Peter; Siekmann, Jörg; Szabó, P.; Unvericht, E. (1979). "[A short survey on the state of the art in matching and unification problems](#)". *ACM SIGSAM Bulletin*. **13** (2): 14–20. doi:10.1145/1089208.1089210.
- Claude Kirchner and Hélène Kirchner. *Rewriting, Solving, Proving*. In preparation.



Last edited 18 days ago by an anonymous user



WIKIPEDIA

Content is available under [CC BY-SA 3.0](#) unless otherwise noted.

[Terms of Use](#) • [Privacy](#) • [Desktop](#)