Unification (computer science)

In logic and computer science, specifically automated reasoning, **unification** is an algorithmic process of solving equations between symbolic expressions, each of the form *Left-hand side = Right-hand side*. For example, using *x*,*y*,*z* as variables, and taking *f* to be an uninterpreted function, the singleton equation set { *f*(1,*y*) = *f*(*x*,2) } is a syntactic first-order unification problem that has the substitution { $x \mapsto 1, y \mapsto 2$ } as its only solution.

Conventions differ on what values variables may assume and which expressions are considered equivalent. In first-order syntactic unification, variables range over first-order terms and equivalence is syntactic. This version of unification has a unique "best" answer and is used in logic programming and programming language type system implementation, especially in Hindley–Milner based type inference algorithms. In higher-order unification, possibly restricted to **higher-order pattern unification**, terms may include lambda expressions, and equivalence is up to beta-reduction. This version is used in proof assistants and higher-order logic programming, for example Isabelle, Twelf, and lambdaProlog. Finally, in semantic unification or E-unification, equality is subject to background knowledge and variables range over a variety of domains. This version is used in SMT solvers, term rewriting algorithms, and cryptographic protocol analysis.

Formal definition

A *unification problem* is a finite set $E=\{ l_1 \doteq r_1, ..., l_n \doteq r_n \}$ of equations to solve, where l_i, r_i are in the set $\boldsymbol{T}$ of *terms* or *expressions*. Depending on which expressions or terms are allowed to occur in an equation set or 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*.

If the right side of each equation is closed (no free variables), the problem is called (pattern) *matching*. The left side (with variables) of each equation is called the *pattern*.[1]

Prerequisites

Formally, a unification approach presupposes

- An infinite set $\boldsymbol{V}$ of *variables*. For higher-order unification, it is convenient to choose $\boldsymbol{V}$ disjoint from the set of lambda-term bound variables.
- A set $\boldsymbol{T}$ of *terms* such that $\boldsymbol{V} \subseteq \boldsymbol{T}$. For first-order unification, $\boldsymbol{T}$ is usually the set of first-order terms (terms built from variable and function symbols). For higher-order unification $\boldsymbol{T}$ consists of first-order terms and lambda terms (terms containing some higher-order variables).
- A mapping $\mathbf{vars}\colon \boldsymbol{T} \to \mathbb{P}(\boldsymbol{V})$, assigning to each term $\boldsymbol{t}$ the set $\mathbf{vars}(\boldsymbol{t}) \subsetneq \boldsymbol{V}$ of *free variables* occurring in $\boldsymbol{t}$.
- A theory or equivalence relation $\equiv$ on $\boldsymbol{T}$, indicating which terms are considered equal. For first-order E-unification, $\equiv$ reflects the background knowledge about certain function symbols; for example, if $\oplus$ is considered commutative, $\boldsymbol{t} \equiv \boldsymbol{u}$ if $\boldsymbol{u}$ results from $\boldsymbol{t}$ by swapping the arguments of $\oplus$ at some (possibly all) occurrences. [note 1] In the most typical case that 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). For higher-order unification, usually $\boldsymbol{t} \equiv \boldsymbol{u}$ if $\boldsymbol{t}$ and $\boldsymbol{u}$ are alpha equivalent.

As an example of how the set of terms and theory affects the set of solutions, the syntactic first-order unification problem { *y* = *cons*(2,*y*) } has no solution over the set of finite terms. However, it has the single solution { *y* ↦ *cons*(2,*cons*(2,*cons*(2,...))) } over the set of infinite tree terms. Similarly, the semantic first-order unification problem { *a*·*x*

= $x \cdot a$ } has each substitution of the form { $x \mapsto a \cdot \ldots \cdot a$ } as a solution in a semigroup, i.e. if (·) is considered associative. But the same problem, viewed in an abelian group, where (·) is considered also commutative, has any substitution at all as a solution.

As an example of higher-order unification, the singleton set { $a = y(x)$ } is a syntactic second-order unification problem, since y is a function variable. One solution is { $x \mapsto a$, $y \mapsto$ (identity function) }; another one is { $y \mapsto$ (constant function mapping each value to a), $x \mapsto$ *(any value)* }.

Substitution

A *substitution* is a mapping $\sigma : V \to T$ from variables to terms; the notation $\{x_1 \mapsto t_1, \ldots, x_k \mapsto t_k\}$ refers to a substitution mapping each variable x_i to the term t_i, for $i = 1, \ldots, k$, and every other variable to itself; the x_i must be pairwise distinct. *Applying* that substitution to a term t is written in postfix notation as $t\{x_1 \mapsto t_1, \ldots, x_k \mapsto t_k\}$; it means to (simultaneously) replace every occurrence of each variable x_i in the term t by t_i. The result $t\tau$ of applying a substitution τ 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

$$f(\quad \mathbf{x} \quad, a, g(\,\mathbf{z}\,), y)$$

yields

$$f(\,\mathbf{h(a,y)}\,, a, g(\,\mathbf{b}\,), y).$$

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 ≡ 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 ≡, 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 2] and vice versa,[note 3] 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 subsumed by $t\tau$ for each term t. We also say that τ is more general than σ. More formally, take a nonempty infinite set V of auxiliary variables such that no equation $l_i \doteq r_i$ in the unification problem contains variables from V. Then a substitution σ is subsumed by another substitution τ if there is a substitution θ such that for all terms $X \notin V$, $X\sigma \equiv X\tau\theta$.[2] For instance $\{x \mapsto a, y \mapsto a\}$ is subsumed by $\tau = \{x \mapsto y\}$, using $\theta = \{y \mapsto a\}$, but $\sigma = \{x \mapsto a\}$ is not subsumed by $\tau = \{x \mapsto y\}$, as $f(x,y)\sigma = f(a,y)$ is not an instance of $f(x,y)\tau = f(y,y)$.[3]

Solution set

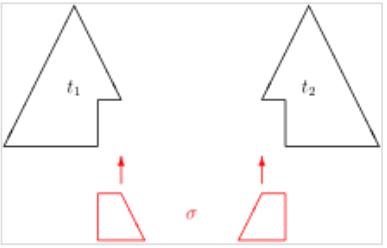
A substitution σ is a *solution* of the unification problem E if $l_i\sigma \equiv r_i\sigma$ for $i = 1, \ldots, n$. Such a substitution is also called a *unifier* of E. For example, if $\oplus$ is associative, the unification problem { $x \oplus a \doteq a \oplus x$ } has the solutions {$x \mapsto a$}, {$x \mapsto a \oplus a$}, {$x \mapsto a \oplus a \oplus a$}, etc., while the problem { $x \oplus a \doteq a$ } has no solution.

For a given unification problem E, a set S of unifiers is called *complete* if each solution substitution is subsumed by some substitution in S. A complete substitution set always exists (e.g. the set of all solutions), but in some frameworks (such as unrestricted higher-order unification) the problem of determining whether any solution exists (i.e., whether the complete substitution set is nonempty) is undecidable.

The set S is called *minimal* if none of its members subsumes another one. Depending on the framework, a complete and minimal substitution set may have zero, one, finitely many, or infinitely many members, or may not exist at all due to an infinite chain of redundant members.[4] Thus, in general, unification algorithms compute a finite approximation of the complete set, which may or may not be minimal, although most algorithms avoid redundant unifiers when possible.[2] For first-order syntactical unification, Martelli and Montanari[5] gave an algorithm that reports unsolvability or computes a single unifier that by itself forms a complete and minimal substitution set, called the **most general unifier**.

Syntactic unification of first-order terms

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 ≡ being *syntactic equality*. In this framework, each solvable unification problem $\{l_1 \doteq r_1, \ldots, l_n \doteq r_n\}$ 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. $l_1\sigma = r_1\sigma \wedge \ldots \wedge l_n\sigma = r_n\sigma$. Any unifier of the problem is subsumed[note 4] 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.



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

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

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

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 ≡ 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.

Unification algorithms

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.[6] 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: $\{ t|_p : t \in T \}$.[7]

Algorithm:[8]

```
Given a set T of terms to be unified
Let σ initially be the identity substitution
do forever
    if Tσ is a singleton set then
        return σ
```

```
    fi
    let D be the disagreement set of Tσ
    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
    σ := σ{s ↦ t}
done
```

Jacques Herbrand discussed the basic concepts of unification and sketched an algorithm in 1930.[9][10][11] But most authors attribute the first unification algorithm to John Alan Robinson (cf. box).[12][13][note 5] Robinson's algorithm had worst-case exponential behavior in both time and space.[11][15] Numerous authors have proposed more efficient unification algorithms.[16] Algorithms with worst-case linear-time behavior were discovered independently by Martelli & Montanari (1976) and Paterson & Wegman (1976).[note 6] Baader & Snyder (2001) uses a similar technique as Paterson-Wegman, hence is linear,[17] but like most linear-time unification algorithms is slower than the Robinson version on small sized inputs due to the overhead of preprocessing the inputs and postprocessing of the output, such as construction of a DAG representation. de Champeaux (2022) is also of linear complexity in the input size but is competitive with the Robinson algorithm on small size inputs. The speedup is obtained by using an object-oriented representation of the predicate calculus that avoids the need for pre- and post-processing, instead making variable objects responsible for creating a substitution and for dealing with aliasing. de Champeaux claims that the ability to add functionality to predicate calculus represented as programmatic objects provides opportunities for optimizing other logic operations as well.[15]

The following algorithm is commonly presented and originates from Martelli & Montanari (1982).[note 7] Given a finite set $G = \{s_1 \doteq t_1, \ldots, s_n \doteq t_n\}$ of potential equations, the algorithm applies rules to transform it to an equivalent set of equations of the form { $x_1 \doteq u_1$, ..., $x_m \doteq u_m$ } where x_1, ..., x_m are distinct variables and u_1, ..., 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 $\bot$; 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.

$$G \cup \{t \doteq t\} \Rightarrow G \qquad \textit{delete}$$

$$G \cup \{f(s_0, \ldots, s_k) \doteq f(t_0, \ldots, t_k)\} \Rightarrow G \cup \{s_0 \doteq t_0, \ldots, s_k \doteq t_k\} \qquad \textit{decompose}$$

$$G \cup \{f(s_0, \ldots, s_k) \doteq g(t_0, \ldots, t_m)\} \Rightarrow \bot \quad \text{if } f \neq g \text{ or } k \neq m \qquad \textit{conflict}$$

$$G \cup \{f(s_0, \ldots, s_k) \doteq x\} \Rightarrow G \cup \{x \doteq f(s_0, \ldots, s_k)\} \qquad \textit{swap}$$

$$G \cup \{x \doteq t\} \Rightarrow G\{x \mapsto t\} \cup \{x \doteq t\} \quad \text{if } x \notin \mathbf{vars}(t) \text{ and } x \in \mathbf{vars}(G) \qquad \textit{eliminate}^{[note\ 8]}$$

$$G \cup \{x \doteq f(s_0, \ldots, s_k)\} \Rightarrow \bot \quad \text{if } x \in \mathbf{vars}(f(s_0, \ldots, s_k)) \qquad \textit{check}$$

Occurs check

An attempt to unify a variable x with a term containing x as a strict subterm $x \doteq f(..., x, ...)$ 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 $x \doteq f(..., x, ...)$ has no solution; hence the *eliminate* rule may only be applied if $x \notin vars(t)$. 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 #Unification of infinite terms 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.

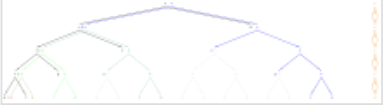
Conor McBride observes[18] that "by expressing the structure which unification exploits" in a dependently typed language such as Epigram, Robinson's unification algorithm can be made recursive on the number of variables, in which case a separate termination proof becomes unnecessary.

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
`a = a`	{ *a* = *a* }	{}	Succeeds. (tautology)
`a = b`	{ *a* = *b* }	⊥	*a* and *b* do not match
`X = X`	{ *x* = *x* }	{}	Succeeds. (tautology)
`a = X`	{ *a* = *x* }	{ *x* ↦ *a* }	*x* is unified with the constant *a*
`X = Y`	{ *x* = *y* }	{ *x* ↦ *y* }	*x* and *y* are aliased
`f(a,X) = f(a,b)`	{ *f*(*a*,*x*) = *f*(*a*,*b*) }	{ *x* ↦ *b* }	function and constant symbols match, *x* is unified with the constant *b*
`f(a) = g(a)`	{ *f*(*a*) = *g*(*a*) }	⊥	*f* and *g* do not match
`f(X) = f(Y)`	{ *f*(*x*) = *f*(*y*) }	{ *x* ↦ *y* }	*x* and *y* are aliased
`f(X) = g(Y)`	{ *f*(*x*) = *g*(*y*) }	⊥	*f* and *g* do not match
`f(X) = f(Y,Z)`	{ *f*(*x*) = *f*(*y*,*z*) }	⊥	Fails. The *f* function symbols have different arity
`f(g(X)) = f(Y)`	{ *f*(*g*(*x*)) = *f*(*y*) }	{ *y* ↦ *g*(*x*) }	Unifies *y* with the term $g(x)$
`f(g(X),X) = f(Y,a)`	{ *f*(*g*(*x*),*x*) = *f*(*y*,*a*) }	{ *x* ↦ *a*, *y* ↦ *g*(*a*) }	Unifies *x* with constant *a*, and *y* with the term $g(a)$
`X = f(X)`	{ *x* = *f*(*x*) }	should be ⊥	Returns ⊥ 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 `x=f(f(f(f(...))))`.
`X = Y, Y = a`	{ *x* = *y*, *y* = *a* }	{ *x* ↦ *a*, *y* ↦ *a* }	Both *x* and *y* are unified with the constant *a*
`a = Y, X = Y`	{ *a* = *y*, *x* = *y* }	{ *x* ↦ *a*, *y* ↦ *a* }	As above (order of equations in set doesn't matter)
`X = a, b = X`	{ *x* = *a*, *b* = *x* }	⊥	Fails. *a* and *b* do not match, so *x* can't be unified with both

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 $(((a * z) * y) * x) * w \doteq w * (x * (y * (z * a)))$ has the most general unifier



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

$\{z \mapsto a, y \mapsto a * a, x \mapsto (a * a) * (a * a), w \mapsto ((a * a) * (a * a)) * ((a * a) * (a * a))\}$, 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. Specifically, unification is a basic building block of resolution, a rule of inference for determining formula satisfiability. In Prolog, the equality symbol = implies first-order syntactic unification. It represents the mechanism of binding the contents of variables and can be viewed as a kind of one-time assignment.

In Prolog:

1. A variable can be unified with a constant, a term, or another 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 constants can be unified only 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.
4. Most operations, including `+`, `-`, `*`, `/`, are not evaluated by =. So for example `1+2 = 3` is not satisfiable because they are syntactically different. The use of integer arithmetic constraints #= introduces a form of E-unification for which these operations are interpreted and evaluated.[20]

Application: type inference

Type inference algorithms are typically based on unification, particularly Hindley-Milner type inference which is used by the functional languages Haskell and ML. For example, when attempting to infer the type of the Haskell expression `True : ['x']`, the compiler will use the type `a -> [a] -> [a]` of the list construction function `(:)`, the type `Bool` of the first argument `True`, and the type `[Char]` of the second argument `['x']`. The polymorphic type variable `a` will be unified with `Bool` and the second argument `[a]` will be unified with `[Char]`. `a` cannot be both `Bool` and `Char` at the same time, therefore this expression is not correctly typed.

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.

Application: Feature Structure Unification

Unification has been used in different research areas of computational linguistics.[21][22]

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* → *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* → *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: $s_1 \cap s_2$. [23] 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. [24] 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. [25] 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" (https://doi.org/10.1016%2F0304-3975%2883%2990059-2). *Theoret. Comput. Sci.* **25** (2): 95–169. doi:10.1016/0304-3975(83)90059-2 (https://doi.org/10.1016%2F0304-3975%2883%2990059-2).
- 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" (https://doi.org/10.1016%2F0304-3975%2886%2990027-7). *Theoretical Computer Science*. **46**: 141–158. doi:10.1016/0304-3975(86)90027-7 (https://doi.org/10.1016%2F0304-3975%2886%2990027-7).

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" (https://doi.org/10.1016%2F0743-1066%2884%2990013-X). *Journal of Logic Programming*. **1** (3): 253–265. doi:10.1016/0743-1066(84)90013-X (https://doi.org/10.1016%2F0743-1066%2884%2990013-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 $x * a \doteq y * b$ 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

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

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

Particular background knowledge sets E

Used naming conventions

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

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[26]
- A,C[27]
- A,C,I[28]
- A,C,N_l[note 9][28]
- A,I[29]
- A,N_l,N_r (monoid)[30]
- C[31]
- Boolean rings[32][33]
- Abelian groups, even if the signature is expanded by arbitrary additional symbols (but not axioms)[34]
- K4 modal algebras[35]

Unification is semi-decidable for the following theories:

- A,D_l,D_r[36]
- A,C,D_l[note 9][37]
- Commutative rings[34]

One-sided paramodulation

If there is a convergent term rewriting system R available for E, the *one-sided paramodulation* algorithm[38] can be used to enumerate all solutions of given equations.

One-sided paramodulation rules

$G \cup \{ f(s_1,\ldots,s_n) \doteq f(t_1,\ldots,t_n) \}\ ;\ S$	$\Rightarrow$	$G \cup \{ s_1 \doteq t_1, \ldots, s_n \doteq t_n \}\ ;\ S$		*decompose*
$G \cup \{ x \doteq t \}\ ;\ S$	$\Rightarrow$	$G\{x \mapsto t\}\ ;\ S\{x \mapsto t\} \cup \{x \mapsto t\}$	if the variable x doesn't occur in t	*eliminate*
$G \cup \{ f(s_1,\ldots,s_n) \doteq t \}\ ;\ S$	$\Rightarrow$	$G \cup \{ s_1 \doteq u_1, \ldots, s_n \doteq u_n, r \doteq t \}\ ;\ S$	if $f(u_1,\ldots,u_n) \to r$ is a rule from R	*mutate*
$G \cup \{ f(s_1,\ldots,s_n) \doteq y \}\ ;\ S$	$\Rightarrow$	$G \cup \{ s_1 \doteq y_1, \ldots, s_n \doteq y_n, y \doteq f(y_1,\ldots,y_n) \}\ ;\ S$	if $y_1,\ldots,y_n$ 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. $G = \{ f(...) \doteq g(...) \}$).

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, ... 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

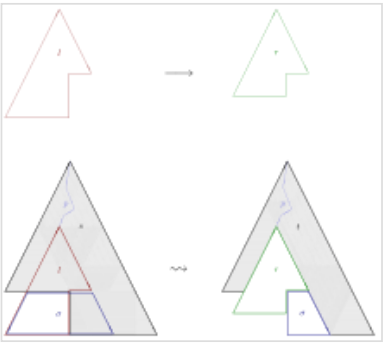
Used rule		**G**	**S**
		$\{ app(x,app(y,x)) \doteq a.a.nil \}$	{}
mutate(2)	⇒	$\{ x \doteq v_2.v_3, app(y,x) \doteq v_4, v_2.app(v_3,v_4) \doteq a.a.nil \}$	{}
decompose	⇒	$\{ x \doteq v_2.v_3, app(y,x) \doteq v_4, v_2 \doteq a, app(v_3,v_4) \doteq a.nil \}$	{}
eliminate	⇒	$\{ app(y,v_2.v_3) \doteq v_4, v_2 \doteq a, app(v_3,v_4) \doteq a.nil \}$	$\{ x \mapsto v_2.v_3 \}$
eliminate	⇒	$\{ app(y,a.v_3) \doteq v_4, app(v_3,v_4) \doteq a.nil \}$	$\{ x \mapsto a.v_3 \}$
mutate(1)	⇒	$\{ y \doteq nil, a.v_3 \doteq v_5, v_5 \doteq v_4, app(v_3,v_4) \doteq a.nil \}$	$\{ x \mapsto a.v_3 \}$
eliminate	⇒	$\{ y \doteq nil, a.v_3 \doteq v_4, app(v_3,v_4) \doteq a.nil \}$	$\{ x \mapsto a.v_3 \}$
eliminate	⇒	$\{ a.v_3 \doteq v_4, app(v_3,v_4) \doteq a.nil \}$	$\{ y \mapsto nil, x \mapsto a.v_3 \}$
mutate(1)	⇒	$\{ a.v_3 \doteq v_4, v_3 \doteq nil, v_4 \doteq v_6, v_6 \doteq a.nil \}$	$\{ y \mapsto nil, x \mapsto a.v_3 \}$
eliminate	⇒	$\{ a.v_3 \doteq v_4, v_3 \doteq nil, v_4 \doteq a.nil \}$	$\{ y \mapsto nil, x \mapsto a.v_3 \}$
eliminate	⇒	$\{ a.nil \doteq v_4, v_4 \doteq a.nil \}$	$\{ y \mapsto nil, x \mapsto a.nil \}$
eliminate	⇒	$\{ a.nil \doteq a.nil \}$	$\{ y \mapsto nil, x \mapsto a.nil \}$
decompose	⇒	$\{ a \doteq a, nil \doteq nil \}$	$\{ y \mapsto nil, x \mapsto a.nil \}$
decompose	⇒	$\{ nil \doteq nil \}$	$\{ y \mapsto nil, x \mapsto a.nil \}$
decompose	⇒	{}	$\{ y \mapsto nil, x \mapsto a.nil \}$

Narrowing

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 $l \to r$ 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 $t_1 \rightsquigarrow t_2 \rightsquigarrow ... \rightsquigarrow t_n$ can be thought of as a sequence of rewrite steps $t_1 \to t_2 \to ... \to t_n$, but with the initial term t_1 being further and further instantiated, as necessary to make each of the used rules applicable.



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

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

$$
\begin{array}{lll}
app(\ x \quad ,app(y,\ x \quad)) & & \\
\quad\downarrow \qquad\quad \downarrow & & x \mapsto v_2.v_3 \\
app(\ v_2.v_3\ ,app(y,\ v_2.v_3\ )) \to v_2.app(v_3,app(\ y \quad ,v_2.v_3)) & & \\
\qquad\qquad\qquad\qquad\quad \downarrow & & y \mapsto nil \\
\qquad\qquad v_2.app(v_3,app(\ nil\ ,v_2.v_3)) \to v_2.app(\ v_3\ ,v_2.\ v_3\ ) & & \\
\qquad\qquad\qquad\qquad\qquad\quad \downarrow \quad\ \downarrow & & v_3 \mapsto nil \\
\qquad\qquad\qquad\qquad v_2.app(\ nil\ ,v_2.\ nil\ ) \to v_2.v_2.nil & &
\end{array}
$$

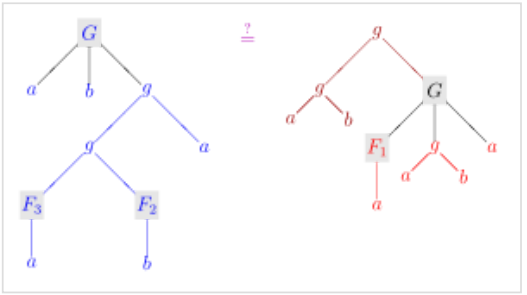
The last term, $v_2.v_2.nil$ can be syntactically unified with the original right hand side term $a.a.nil$.

The *narrowing lemma*[39] 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 $s\sigma \overset{*}{\to} t$ holds for some substitution σ, then there exist terms s', t' such that $s \overset{*}{\rightsquigarrow} s'$ and $t \overset{*}{\to} t'$ 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*. Higher-order unification is undecidable,[40][41][42] and such unification problems do not have most general unifiers. 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 {$f \mapsto \lambda x.\lambda y.\lambda z.\ d(b,a,c)$ }. A well studied branch of higher-order unification is the problem of unifying simply typed lambda terms modulo the equality determined by αβη conversions. Gérard Huet gave a semi-decidable (pre-)unification algorithm[43] that allows a systematic search of the space of unifiers (generalizing the unification algorithm of Martelli-Montanari[5] with rules for terms containing higher-order variables) that seems to work sufficiently well in practice. Huet[44] and Gilles Dowek[45] have written articles surveying this topic.



In Goldfarb's[40] reduction of Hilbert's 10th problem to second-order unifiability, the equation $X_1 * X_2 = X_3$ corresponds to the depicted unification problem, with function variables F_i corresponding to X_i and G fresh.

Several subsets of higher-order unification are well-behaved, in that they are decidable and have a most-general unifier for solvable problems. One such subset is the previously described first-order terms. **Higher-order pattern unification**, due to Dale Miller,[46] is another such subset. The higher-order logic programming languages λProlog and Twelf have switched from full higher-order unification to implementing only the pattern fragment; surprisingly pattern unification is sufficient for almost all programs, if each non-pattern unification problem is suspended until a subsequent substitution puts the unification into the pattern fragment. A superset of pattern unification called functions-as-constructors unification is also well-behaved.[47] The Zipperposition theorem prover has an algorithm integrating these well-behaved subsets into a full higher-order unification algorithm.[2]

In computational linguistics, one of the most influential theories of elliptical construction is that ellipses are represented by free variables whose values are then determined using Higher-Order Unification. For instance, the semantic representation of "Jon likes Mary and Peter does too" is $\mathrm{like}(j,\ m)\ \wedge\ \mathrm{R}(p)$ and the value of R (the semantic representation of the ellipsis) is determined by the equation $\mathrm{like}(j,\ m) = \mathrm{R}(j)$. The process of solving such equations is called Higher-Order Unification.[48]

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.[49]

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)
- Subsumption lattice, a lattice having unification as meet and anti-unification as join
- Ontology alignment (use *unification* with semantic equivalence)

Notes

1. E.g. $a \oplus (b \oplus f(x)) \equiv a \oplus (f(x) \oplus b) \equiv (b \oplus f(x)) \oplus a \equiv (f(x) \oplus b) \oplus a$
2. since $(x \oplus y)\{x \mapsto z, y \mapsto z\} = z \oplus z \equiv z$
3. since $z\ \{z \mapsto x \oplus y\} = x \oplus y$
4. formally: each unifier τ satisfies $\forall x\colon x\tau = (x\sigma)\rho$ for some substitution ρ
5. Robinson 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.[14]
6. Independent discovery is stated in Martelli & Montanari (1982) sect.1, p.259. The journal publisher received Paterson & Wegman (1978) in Sep.1976.
7. 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*.
8. Although the rule keeps $x \doteq t$ in G, it cannot loop forever since its precondition $x \in vars(G)$ is invalidated by its first application. More generally, the algorithm is guaranteed to terminate always, see below.
9. in the presence of equality C, equalities N_l and N_r are equivalent, similar for D_l and D_r

References

1. Dowek, Gilles (1 January 2001). "Higher-order unification and matching". *Handbook of automated reasoning* (https://web.archive.org/web/20190515113555/http://www.lsv.fr/~dowek/Publi/unification.ps). Elsevier Science Publishers B. V. pp. 1009–1062. ISBN 978-0-444-50812-6. Archived from the original (http://www.lsv.fr/~dowek/Publi/unification.ps) on 15 May 2019. Retrieved 15 May 2019.

2. Vukmirović, Petar; Bentkamp, Alexander; Nummelin, Visa (14 December 2021). "Efficient Full Higher-Order Unification" (https://doi.org/10.46298%2Flmcs-17%284%3A18%292021). *Logical Methods in Computer Science*. **17** (4): 6919. arXiv:2011.09507 (https://arxiv.org/abs/2011.09507). doi:10.46298/lmcs-17(4:18)2021 (https://doi.org/10.46298%2Flmcs-17%284%3A18%292021).
3. Apt, Krzysztof R. (1997). *From logic programming to Prolog* (https://homepages.cwi.nl/~apt/book.ps) (1. publ ed.). London Munich: Prentice Hall. p. 24. ISBN 013230368X.
4. Fages, François; Huet, Gérard (1986). "Complete Sets of Unifiers and Matchers in Equational Theories" (https://doi.org/10.1016%2F0304-3975%2886%2990175-1). *Theoretical Computer Science*. **43**: 189–200. doi:10.1016/0304-3975(86)90175-1 (https://doi.org/10.1016%2F0304-3975%2886%2990175-1).
5. 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%2F357162.357169). S2CID 10921306 (https://api.semanticscholar.org/CorpusID:10921306).
6. Robinson (1965) nr.2.5, 2.14, p.25
7. Robinson (1965) nr.5.6, p.32
8. Robinson (1965) nr.5.8, p.32
9. J. Herbrand: Recherches sur la théorie de la démonstration. *Travaux de la société des Sciences et des Lettres de Varsovie*, Class III, Sciences Mathématiques et Physiques, 33, 1930.
10. Jacques Herbrand (1930). *Recherches sur la théorie de la demonstration* (http://www.numdam.org/issue/THESE_1930__110__1_0.pdf) (PDF) (Ph.D. thesis). A. Vol. 1252. Université de Paris. Here: p.96-97
11. Claus-Peter Wirth; Jörg Siekmann; Christoph Benzmüller; Serge Autexier (2009). Lectures on Jacques Herbrand as a Logician (SEKI Report). DFKI. arXiv:0902.4682 (https://arxiv.org/abs/0902.4682). Here: p.56
12. Robinson, J.A. (Jan 1965). "A Machine-Oriented Logic Based on the Resolution Principle" (https://doi.org/10.1145%2F321250.321253). *Journal of the ACM*. **12** (1): 23–41. doi:10.1145/321250.321253 (https://doi.org/10.1145%2F321250.321253). S2CID 14389185 (https://api.semanticscholar.org/CorpusID:14389185).; Here: sect.5.8, p.32
13. J.A. Robinson (1971). "Computational logic: The unification computation" (https://aitopics.org/download/classics:E35191E8). *Machine Intelligence*. **6**: 63–72.
14. David A. Duffy (1991). *Principles of Automated Theorem Proving*. New York: Wiley. ISBN 0-471-92784-8. Here: Introduction of sect.3.3.3 *"Unification"*, p.72.
15. de Champeaux, Dennis (Aug 2022). "Faster Linear Unification Algorithm" (https://raw.githubusercontent.com/ddccc/Unification/a5975a47bca1be3f7bf0afe9ad3595a707a29ea4/LinUnify3.pdf) (PDF). *Journal of Automated Reasoning*. **66**: 845–860. doi:10.1007/s10817-022-09635-1 (https://doi.org/10.1007%2Fs10817-022-09635-1).
16. Per Martelli & Montanari (1982):
    - Lewis Denver Baxter (Feb 1976). A practically linear unification algorithm (https://cs.uwaterloo.ca/research/tr/1976/CS-76-13.pdf) (PDF) (Res. Report). Vol. CS-76-13. Univ. of Waterloo, Ontario.
    - Gérard Huet (Sep 1976). *Resolution d'Equations dans des Langages d'Ordre 1,2,...ω* (These d'etat). Universite de Paris VII.
    - Martelli, Alberto & Montanari, Ugo (Jul 1976). Unification in linear time and space: A structured presentation (https://web.archive.org/web/20150115070153/http://puma.isti.cnr.it/publichtml/section_cnr_iei/cnr_iei_1976-B4-041.html) (Internal Note). Vol. IEI-B76-16. Consiglio Nazionale delle Ricerche, Pisa. Archived from the original (http://puma.isti.cnr.it/publichtml/section_cnr_iei/cnr_iei_1976-B4-041.html) on 2015-01-15.
    - Paterson, M.S.; Wegman, M.N. (May 1976). Chandra, Ashok K.; Wotschke, Detlef; Friedman, Emily P.; Harrison, Michael A. (eds.). *Linear unification*. Proceedings of the eighth annual ACM Symposium on Theory of Computing (STOC). ACM. pp. 181–186. doi:10.1145/800113.803646 (https://doi.org/10.1145%2F800113.803646).
    - Paterson, M.S.; Wegman, M.N. (Apr 1978). "Linear unification" (https://doi.org/10.1016%2F0022-0000%2878%2990043-0). *J. Comput. Syst. Sci*. **16** (2): 158–167. doi:10.1016/0022-0000(78)90043-0 (https://doi.org/10.1016%2F0022-0000%2878%2990043-0).
    - J.A. Robinson (Jan 1976). "Fast unification" (http://oda.mfo.de/bsz325106819.html). In Woodrow W. Bledsoe, Michael M. Richter (ed.). *Proc. Theorem Proving Workshop Oberwolfach*. Oberwolfach Workshop Report. Vol. 1976/3.
    - 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%2FBF02575754). S2CID 189789152 (https://api.semanticscholar.org/CorpusID:189789152).
17. Baader, Franz; Snyder, Wayne (2001). "Unification Theory" (https://www.cs.bu.edu/fac/snyder/publications/UnifChapter.pdf) (PDF). *Handbook of Automated Reasoning*. pp. 445–533. doi:10.1016/B978-044450813-3/50010-2 (https://doi.org/10.1016%2FB978-044450813-3%2F50010-2).

18. McBride, Conor (October 2003). "First-Order Unification by Structural Recursion" (http://strictlypositive.org/unify.ps.gz). *Journal of Functional Programming*. **13** (6): 1061–1076. CiteSeerX 10.1.1.25.1516 (https://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.25.1516). doi:10.1017/S0956796803004957 (https://doi.org/10.1017%2FS0956796803004957). ISSN 0956-7968 (https://search.worldcat.org/issn/0956-7968). S2CID 43523380 (https://api.semanticscholar.org/CorpusID:43523380). Retrieved 30 March 2012.
19. e.g. Paterson & Wegman (1978) sect.2, p.159
20. "Declarative integer arithmetic" (https://www.swi-prolog.org/pldoc/man?section=clpfd-integer-arith). *SWI-Prolog*. Retrieved 18 February 2024.
21. Jonathan Calder, Mike Reape, and Hank Zeevat,, An algorithm for generation in unification categorial grammar (https://www.aclweb.org/anthology/E89-1032). In Proceedings of the 4th Conference of the European Chapter of the Association for Computational Linguistics, pages 233-240, Manchester, England (10–12 April), University of Manchester Institute of Science and Technology, 1989.
22. Graeme Hirst and David St-Onge, [1] (http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.50.8426) Lexical chains as representations of context for the detection and correction of malapropisms, 1998.
23. Walther, Christoph (1985). "A Mechanical Solution of Schubert's Steamroller by Many-Sorted Resolution" (https://web.archive.org/web/20110708231225/http://www.inferenzsysteme.informatik.tu-darmstadt.de/media/is/publikationen/Schuberts_Steamroller_by_Many-Sorted_Resolution-AIJ-25-2-1985.pdf) (PDF). *Artif. Intell*. **26** (2): 217–224. doi:10.1016/0004-3702(85)90029-3 (https://doi.org/10.1016%2F0004-3702%2885%2990029-3). Archived from the original (http://www.inferenzsysteme.informatik.tu-darmstadt.de/media/is/publikationen/Schuberts_Steamroller_by_Many-Sorted_Resolution-AIJ-25-2-1985.pdf) (PDF) on 2011-07-08. Retrieved 2013-06-28.
24. Smolka, Gert (Nov 1988). *Logic Programming with Polymorphically Order-Sorted Types* (https://link.springer.com/content/pdf/10.1007/3-540-50667-5_58.pdf) (PDF). Int. Workshop Algebraic and Logic Programming. LNCS. Vol. 343. Springer. pp. 53–70. doi:10.1007/3-540-50667-5_58 (https://doi.org/10.1007%2F3-540-50667-5_58).
25. Schmidt-Schauß, Manfred (Apr 1988). *Computational Aspects of an Order-Sorted Logic with Term Declarations*. Lecture Notes in Artificial Intelligence (LNAI). Vol. 395. Springer.
26. Gordon D. Plotkin, *Lattice Theoretic Properties of Subsumption*, Memorandum MIP-R-77, Univ. Edinburgh, Jun 1970
27. Mark E. Stickel, *A Unification Algorithm for Associative-Commutative Functions*, Journal of the Association for Computing Machinery, vol.28, no.3, pp. 423–434, 1981
28. F. Fages, *Associative-Commutative Unification*, J. Symbolic Comput., vol.3, no.3, pp. 257–275, 1987
29. Franz Baader, *Unification in Idempotent Semigroups is of Type Zero*, J. Automat. Reasoning, vol.2, no.3, 1986
30. J. Makanin, *The Problem of Solvability of Equations in a Free Semi-Group*, Akad. Nauk SSSR, vol.233, no.2, 1977
31. F. Fages (1987). "Associative-Commutative Unification" (https://hal.inria.fr/inria-00076271/file/RR-0287.pdf) (PDF). *J. Symbolic Comput*. **3** (3): 257–275. doi:10.1016/s0747-7171(87)80004-4 (https://doi.org/10.1016%2Fs0747-7171%2887%2980004-4). S2CID 40499266 (https://api.semanticscholar.org/CorpusID:40499266).
32. Martin, U., Nipkow, T. (1986). "Unification in Boolean Rings". In Jörg H. Siekmann (ed.). *Proc. 8th CADE*. LNCS. Vol. 230. Springer. pp. 506–513.
33. A. Boudet; J.P. Jouannaud; M. Schmidt-Schauß (1989). "Unification of Boolean Rings and Abelian Groups" (https://doi.org/10.1016%2Fs0747-7171%2889%2980054-9). *Journal of Symbolic Computation*. **8** (5): 449–477. doi:10.1016/s0747-7171(89)80054-9 (https://doi.org/10.1016%2Fs0747-7171%2889%2980054-9).
34. Baader and Snyder (2001), p. 486.
35. F. Baader and S. Ghilardi, *Unification in modal and description logics (https://web.archive.org/web/20171223215706/https://pdfs.semanticscholar.org/492e/9f03ab7abd043ed0167dc7309552d21a88ef.pdf)*, Logic Journal of the IGPL 19 (2011), no. 6, pp. 705–730.
36. P. Szabo, *Unifikationstheorie erster Ordnung* (*First Order Unification Theory*), Thesis, Univ. Karlsruhe, West Germany, 1982
37. Jörg H. Siekmann, *Universal Unification*, Proc. 7th Int. Conf. on Automated Deduction, Springer LNCS vol.170, pp. 1–42, 1984
38. 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
39. Fay (1979). "First-Order Unification in an Equational Theory". *Proc. 4th Workshop on Automated Deduction*. pp. 161–167.
40. Warren D. Goldfarb (1981). "The Undecidability of the Second-Order Unification Problem" (https://doi.org/10.1016%2F0304-3975%2881%2990040-2). *TCS*. **13** (2): 225–230. doi:10.1016/0304-3975(81)90040-2 (https://doi.org/10.1016%2F0304-3975%2881%2990040-2).

41. Gérard P. Huet (1973). "The Undecidability of Unification in Third Order Logic" (https://doi.org/10.1016%2FS0019-9958%2873%2990301-X). *Information and Control*. **22** (3): 257–267. doi:10.1016/S0019-9958(73)90301-X (https://doi.org/10.1016%2FS0019-9958%2873%2990301-X).
42. Claudio Lucchesi: The Undecidability of the Unification Problem for Third Order Languages (Research Report CSRR 2059; Department of Computer Science, University of Waterloo, 1972)
43. Gérard Huet: A Unification Algorithm for typed Lambda-Calculus []
44. Gérard Huet: Higher Order Unification 30 Years Later (http://portal.acm.org/citation.cfm?id=695200)
45. Gilles Dowek: Higher-Order Unification and Matching. Handbook of Automated Reasoning 2001: 1009–1062
46. Miller, Dale (1991). "A Logic Programming Language with Lambda-Abstraction, Function Variables, and Simple Unification" (http://www.lix.polytechnique.fr/Labo/Dale.Miller/papers/jlc91.pdf) (PDF). *Journal of Logic and Computation*. **1** (4): 497–536. doi:10.1093/logcom/1.4.497 (https://doi.org/10.1093%2Flogcom%2F1.4.497).
47. Libal, Tomer; Miller, Dale (May 2022). "Functions-as-constructors higher-order unification: extended pattern unification" (https://doi.org/10.1007%2Fs10472-021-09774-y). *Annals of Mathematics and Artificial Intelligence*. **90** (5): 455–479. doi:10.1007/s10472-021-09774-y (https://doi.org/10.1007%2Fs10472-021-09774-y).
48. 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 (https://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.55.9018).
49. Wayne Snyder (Jul 1990). "Higher order E-unification". *Proc. 10th Conference on Automated Deduction*. LNAI. Vol. 449. Springer. pp. 573–587.

Further reading

- Franz Baader and Wayne Snyder (2001). "Unification Theory" (http://www.cs.bu.edu/~snyder/publications/UnifChapter.pdf) Archived (https://web.archive.org/web/20150608053650/http://www.cs.bu.edu/~snyder/publications/UnifChapter.pdf) 2015-06-08 at the Wayback Machine. 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" (http://www.lsv.fr/~dowek/Publi/unification.ps) Archived (https://web.archive.org/web/20190515113555/http://www.lsv.fr/~dowek/Publi/unification.ps) 2019-05-15 at the Wayback Machine. In *Handbook of Automated Reasoning*.
- Franz Baader and Tobias Nipkow (1998). *Term Rewriting and All That* (https://www.in.tum.de/~nipkow/TRaAT/). Cambridge University Press.
- Franz Baader and Jörg H. Siekmann (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" (http://www.isi.edu/natural-language/people/unification-knight.pdf) (PDF). *ACM Computing Surveys*. **21** (1): 93–124. CiteSeerX 10.1.1.64.8967 (https://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.64.8967). doi:10.1145/62029.62030 (https://doi.org/10.1145%2F62029.62030). S2CID 14619034 (https://api.semanticscholar.org/CorpusID:14619034).
- Gérard Huet and Derek C. Oppen (1980). "Equations and Rewrite Rules: A Survey" (http://infolab.stanford.edu/pub/cstr/reports/cs/tr/80/785/CS-TR-80-785.pdf). 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 (https://doi.org/10.1145%2F1089208.1089210). S2CID 17033087 (https://api.semanticscholar.org/CorpusID:17033087).
- Claude Kirchner and Hélène Kirchner. *Rewriting, Solving, Proving*. In preparation.