# Unification Theory

Franz Baader
Wayne Snyder

**SECOND READERS:** Paliath Narendran, Manfred Schmidt-Schauss, and Klaus Schulz.

## Contents

<table>
<thead>
<tr>
<th>Chapter</th>
<th>Title</th>
<th>Pages</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Introduction</td>
<td>447</td>
</tr>
<tr>
<td>1.1</td>
<td>What is unification?</td>
<td>447</td>
</tr>
<tr>
<td>1.2</td>
<td>History and applications</td>
<td>448</td>
</tr>
<tr>
<td>1.3</td>
<td>Approach</td>
<td>450</td>
</tr>
<tr>
<td>2</td>
<td>Syntactic unification</td>
<td>450</td>
</tr>
<tr>
<td>2.1</td>
<td>Definitions</td>
<td>451</td>
</tr>
<tr>
<td>2.2</td>
<td>Unification of terms</td>
<td>452</td>
</tr>
<tr>
<td>2.3</td>
<td>Unification of term dags</td>
<td>459</td>
</tr>
<tr>
<td>3</td>
<td>Equational unification</td>
<td>469</td>
</tr>
<tr>
<td>3.1</td>
<td>Basic notions</td>
<td>469</td>
</tr>
<tr>
<td>3.2</td>
<td>New issues</td>
<td>473</td>
</tr>
<tr>
<td>3.3</td>
<td>Reformulations</td>
<td>475</td>
</tr>
<tr>
<td>3.4</td>
<td>Survey of results for specific theories</td>
<td>482</td>
</tr>
<tr>
<td>4</td>
<td>Syntactic methods for $E$-unification</td>
<td>488</td>
</tr>
<tr>
<td>4.1</td>
<td>$E$-unification in arbitrary theories</td>
<td>488</td>
</tr>
<tr>
<td>4.2</td>
<td>Restrictions on $E$-unification in arbitrary theories</td>
<td>495</td>
</tr>
<tr>
<td>4.3</td>
<td>Narrowing</td>
<td>495</td>
</tr>
<tr>
<td>4.4</td>
<td>Strategies and refinements of basic narrowing</td>
<td>499</td>
</tr>
<tr>
<td>5</td>
<td>Semantic approaches to $E$-unification</td>
<td>503</td>
</tr>
<tr>
<td>5.1</td>
<td>Unification modulo $ACU$, $ACUI$, and $AG$: an example</td>
<td>504</td>
</tr>
<tr>
<td>5.2</td>
<td>The class of commutative/monoidal theories</td>
<td>508</td>
</tr>
<tr>
<td>5.3</td>
<td>The corresponding semiring</td>
<td>510</td>
</tr>
<tr>
<td>5.4</td>
<td>Results on unification in commutative theories</td>
<td>511</td>
</tr>
<tr>
<td>6</td>
<td>Combination of unification algorithms</td>
<td>513</td>
</tr>
<tr>
<td>6.1</td>
<td>A general combination method</td>
<td>514</td>
</tr>
<tr>
<td>6.2</td>
<td>Proving correctness of the combination method</td>
<td>517</td>
</tr>
<tr>
<td>7</td>
<td>Further topics</td>
<td>519</td>
</tr>
<tr>
<td>Bibliography</td>
<td></td>
<td>521</td>
</tr>
<tr>
<td>Index</td>
<td></td>
<td>531</td>
</tr>
</tbody>
</table>
1. Introduction

Unification is a fundamental process upon which many methods for automated deduction are based. Unification theory abstracts from the specific applications of this process: it provides formal definitions for important notions like instantiation, most general unifier, etc., investigates properties of these notions, and provides and analyzes unification algorithms that can be used in different contexts. In this introductory section, we will first present the concept of unification in an informal way, then make some historical remarks on where unification was originally introduced, and finally explain our approach to writing this chapter.

1.1. What is unification?

Very generally speaking, unification tries to identify two symbolic expressions by replacing certain sub-expressions (variables) by other expressions. To be more concrete, one usually considers terms that are built from function symbols (say $f$, $a$, and $b$, where $f$ is binary and $a, b$ are nullary) and variable symbols (say $x$ and $y$). The unification problem for the terms $s = f(a, x)$ and $t = f(y, b)$ is concerned with the following question: is it possible to replace the variables $x, y$ in $s$ and $t$ by terms such that the two terms obtained this way are (syntactically) equal. In this example, if we substitute $b$ for $x$ and $a$ for $y$, we obtain the unified term $f(a, b)$. This substitution is denoted as $u := \{x \mapsto b, y \mapsto a\}$, and its application to terms is written suffix, i.e., $s u = f(a, b) = t u$. Note that different occurrences of the same variable in a unification problem must always be replaced by the same term. For this reason, the terms $s' = f(a, x)$ and $t' = f(x, b)$ cannot be unified since this would require the occurrence of $x$ in $s'$ to be replaced by $b$, and the occurrence of $x$ in $t'$ to be replaced by the different constant $a$.

In most applications of unification, one is not just interested in the decision problem for unification, which simply asks for a “yes” or “no” answer to the question of whether two terms $s$ and $t$ are unifiable. If they are unifiable, one would like to construct a solution, i.e., a substitution that identifies $s$ and $t$. Such a substitution is called a unifier of $s$ and $t$. In general, a unification problem may have infinitely many solutions; e.g., $f(x, y)$ and $f(y, x)$ can be unified by replacing $x$ and $y$ by the same term $s$ (and there are infinitely many terms available). Fortunately, the applications of unification in automated deduction do not require the computation of all unifiers. It is sufficient to consider the so-called most general unifier, i.e., a unifier such that every other unifier can be obtained by instantiation. In the above example, $\sigma := \{x \mapsto y\}$ is such a most general unifier since for all terms $s$ we have $\{x \mapsto s, y \mapsto s\} = \sigma \{y \mapsto s\}$. A unification algorithm should thus not only decide solvability of a given unification problem: if the problem is solvable, it should also compute a most general unifier. As we will show, there exist very efficient algorithms for this purpose.

Unification as described until now is called syntactic unification of first-order terms. “Syntactic” means that the terms must be made syntactically equal, whereas
"first-order" expresses the fact that we do not allow for higher-order variables, i.e., variables for functions. For example, the terms $f(x,a)$ and $g(a,x)$ obviously cannot be made syntactically equal by first-order unification. However, $f(x,a)$ and $G(a,x)$ can be made equal by higher-order unification if $G$ is a (higher-order) variable, which may be replaced by $f$. We will not consider higher-order unification in more detail since it is treated in [Dowek 2001] (Chapter 16 of this Handbook). However, equational unification—as opposed to syntactic unification—of first-order terms will be one of the most important topics of this chapter. Instead of requiring that the terms are made syntactically equal, equational unification is concerned with making terms equivalent with respect to a congruence induced by certain equational axioms $E$. In this case, one talks about $E$-unification or unification modulo $E$. For example, if $E = \{ f(a,a) \approx g(a,a) \}$, then the terms $f(x,a)$ and $g(a,x)$, which are not (syntactically) unifiable, are $E$-unifiable: for the substitution $\sigma := \{ x \mapsto a \}$, we have $f(x,a)\sigma = f(a,a) = E g(a,a) = g(a,x)\sigma$, where $=E$ denotes the equational theory induced by $E$. For equational unification, things are not as nice as for syntactic unification. In fact, depending on the theory $E$ in question, $E$-unifiability may be undecidable, and even if it is decidable, solvable unification problems need not have a most general $E$-unifier. Research on equational unification is, on the one hand, interested in classifying equational theories of interest according to their behavior in this respect. On the other hand, it develops general approaches and algorithms that apply to whole classes of theories.

1.2. History and applications

The name "unification" and the first formal investigation of this notion is due to J.A. Robinson [1965], who introduced unification as the basic operation of his resolution principle, showed that unifiable terms have a most general unifier, and described an algorithm for computing this unifier. In the propositional case, the resolution principle can be described as follows, see also [Bachmair and Ganzinger 2001] (Chapter 2 of this Handbook). Assume that clauses $C \lor p$ and $C' \lor \neg p$ have already been derived (where $C, C'$ are sub-clauses and $p$ is a propositional variable). Then one can also deduce $C \lor C'$. In the first-order case, the clauses one starts with may contain variables. Herbrand's famous theorem implies that finitely many ground instances (i.e., instances obtained by substituting all variables by terms without variables) are sufficient to show unsatisfiability of a given unsatisfiable set of clauses by propositional reasoning (e.g., propositional resolution). The problem is, however, to find the appropriate instantiations. Early theorem provers approached this problem by a breadth-first enumeration of all possible ground instantiations, which led to an immediate combinatorial explosion [Robinson 1963]. Theorem provers based on the resolution principle need not search blindly for the right instantiations: they can compute them via syntactic unification. For example, assume the clauses $C \lor P(s)$ and $C' \lor \neg P(t)$ are given. Obviously, the resolution rule applies to ground instances of these clauses iff in these instances the predicate $P$ contains the same term, i.e., iff the substitution used in the instantiation process is a (syntactic) unifier of $s$ and $t$. 
Instead of using all ground unifiers for instantiation, Robinson proposed to lift the resolution principle to terms with variables, and apply only the most general unifier $\sigma$ of $s$ and $t$. In the example, this yields the resolvent $(C \lor C')\sigma$. The completeness proof for propositional resolution can be lifted to non-ground resolution by using the fact that every ground unifier of $s, t$ is an instance of the most general unifier. In fact, the notion "most general unifier" was defined in this way just to make this lifting possible.

Similar ideas for determining appropriate instantiations have been proposed prior to Robinson by Post, Herbrand [1930a, 1930b, 1967, 1971] (in the investigation of his property $A$), Prawitz [1960], and Guard [1964, 1969]. However, in this previous work, the notions "unification" and "most general unifier" are not singled out as interesting concepts of their own (they don’t even receive a name). Prawitz only considers the function-free case (in which unification is rather trivial), and Herbrand also first presents his approach for this restricted case. The description by Herbrand of the unification algorithm for the general case (which appears to be the first published account of such an algorithm, and which is similar to the transformation-based algorithm by Martelli and Montanari [1982]) is rather informal, and there is no proof of correctness.\footnote{Strictly speaking, Herbrand’s unification algorithm is not an algorithm for simple syntactic unification, but an algorithm for unification with so-called linear constant restrictions (see section 3.3.2). This is due to the fact that he does not Skolemize his formulae, and thus he has both universal and existential quantifiers in the quantifier prefix.}

The notions “unification” and “most general unifier” were independently re-invented by Knuth and Bendix [1970] as a tool for testing term rewriting systems for local confluence by computing critical pairs. Again, the definition of the most general unifier makes sure that every critical situation is an instance of a critical pair, and thus it is sufficient to test the critical pairs for confluence, see [Dershowitz and Plaisted 2001] (Chapter 9 of this Handbook). \textit{Equational unification} was introduced both in resolution-based theorem proving and in term rewriting as a means for treating certain troublesome equational axioms (like associativity and commutativity) in a special manner. In automated theorem proving, it quickly became apparent that the equality relation requires a special treatment (see [Degtyarev and Voronkov 2001a, Nieuwenhuis and Rubio 2001], Chapters 10 and 7 of this Handbook), since a simple integration of axioms that describe the properties of equality (in principle, being a congruence relation) often leads to an unacceptable increase in the search space. Whereas the first approaches providing such a special treatment of equality replaced only the axiomatization of equality by special inference rules, Plotkin [1972] proposed to go one step further. In his approach, also certain axioms that use equality (like $f(x, y) \approx f(y, x)$ and $f(f(x, y), z) \approx f(x, f(y, z)))$ can be built into the inference rule (namely resolution). This is achieved by replacing the use of syntactic unification in the resolution step by equational unification, i.e., unification modulo the equational theory induced by the axioms to be built in.

In term rewriting, axioms like commutativity (i.e., $f(x, y) \approx f(y, x)$) cannot be oriented into terminating rewrite rules. One way of solving this problem is to take such non-orientable identities completely out of the rewrite process, and perform
rewriting with respect to the remaining (orientable) rules modulo the unoriented ones. In this setting, critical pairs must now be computed by equational unification, i.e., modulo the unoriented identities, see, e.g., [Peterson and Stickel 1981, Jouannaud and Kirchner 1986] and [Dershowitz and Plaisted 2001] (Chapter 9 of this Handbook).

1.3. Approach

This chapter is not intended to give a complete coverage of all the results obtained in unification theory (see the overview articles [Jouannaud and Kirchner 1991, Baader and Siekmann 1994] for this purpose). Instead we try to cover a number of significant topics in more detail. This should give a feeling for unification research and its methodology, provide the most important references, and enable the reader to study recent research papers on the topic.

Notational and typographic conventions
We will try to keep as close as possible to the typographic conventions introduced by Dershowitz and Jouannaud [1991], which they also used in their survey article on rewrite systems [Dershowitz and Jouannaud 1990]. In particular, substitutions are written in suffix notation (i.e., $\sigma s$ instead of $\sigma(s)$), and consequently composition of substitution should be read from left to right (i.e., $\sigma \tau$ means: first apply $\sigma$ and then $\tau$).

Equational axioms (written $s \approx t$) that define equational theories will be called "identities," whereas unification problems consist of "equations" (written $s = ? t$ for syntactic unification and $s = ?^E t$ for unification modulo $E$). Thus, identities must hold, whereas equations must be solved.

2. Syntactic unification

As mentioned earlier, syntactic unification of first-order terms was introduced by Post and Herbrand in the early part of this century. Various researchers have studied the problem further [Champeaux 1986, Corbin and Bidoit 1983, Huet 1976, Martelli and Montanari 1982, Paterson and Wegman 1978, Robinson 1971, Venturini-Zilli 1975] and, among other results, it was shown that linear time algorithms for unification exist [Martelli and Montanari 1976, Paterson and Wegman 1978]. The corresponding lower complexity bound was shown by Dwork, Kanellakis and Mitchell [1984]: the unification problem is log-space complete for $P$, the class of polynomial-time solvable problems. In particular, this implies that it is very unlikely that an efficient parallel unification algorithm exists.

In this section we review the major approaches to syntactic unification.
2.1. Definitions

A signature is a (finite or countably infinite) set of function symbols \( \mathcal{F} \). We assume the reader is familiar with the term algebra \( T(\mathcal{F}, \mathcal{V}) \) generated by a signature function symbols \( \mathcal{F} \) and a (countably) infinite set of variables \( \mathcal{V} \); we shall call these \( \mathcal{F} \)-terms, or simply terms when \( \mathcal{F} \) is unimportant, and denote them by the letters \( l, r, s, t, u, \) and \( v \). Variables will be denoted by \( w, x, y, \) and \( z \). The set of variables occurring in a term \( t \) will be denoted by \( \text{Vars}(t) \), and this will be extended to sets of variables, equations, and sets of equations.

A substitution is a mapping from variables to terms which is almost everywhere equal to the identity, and will generally be represented by \( \sigma, \theta, \eta, \) or \( \rho \). The identity substitution is represented by \( \text{Id} \). The application of a substitution \( \sigma \) to a term \( t \), denoted \( t\sigma \), is defined by induction on the structure of terms:

\[
\begin{align*}
    t\sigma & := \begin{cases} 
    x\sigma & \text{if } t = x, \\
    f(t_1\sigma, \ldots, t_n\sigma) & \text{if } t = f(t_1, \ldots, t_n).
    \end{cases}
\end{align*}
\]

In the second case of this definition, \( n = 0 \) is allowed: in this case, \( f \) is a constant symbol and \( f\sigma = f \). Substitutions can also be applied to sets of terms, equations, and sets of equations, in the obvious fashion.

For a substitution \( \sigma \), the domain is the set of variables

\[
\text{Dom}(\sigma) := \{ x | x\sigma \neq x \},
\]

the range is the set of terms

\[
\text{Ran}(\sigma) := \bigcup_{x \in \text{Dom}(\sigma)} \{ x\sigma \},
\]

and the set of variables occurring in the range is \( \text{Vran}(\sigma) := \text{Vars}(\text{Ran}(\sigma)) \).

A substitution can be represented explicitly as a function by a set of bindings of variables in its domain, e.g.,

\[
\{ x_1 \mapsto s_1, \ldots, x_n \mapsto s_n \}.
\]

The restriction of a substitution \( \theta \) to a set of variables \( X \), denoted by \( \theta|_X \), is the substitution which is equal to the identity everywhere except over \( X \cap \text{Dom}(\sigma) \), where it is equal to \( \sigma \). Composition of two substitutions is written \( \sigma\theta \), and is defined by

\[
t\sigma\theta = (t\sigma)\theta.
\]

An algorithm for constructing the composition \( \sigma\theta \) of two substitutions represented as sets of bindings is as follows:

1. Apply \( \theta \) to every term in \( \text{Ran}(\sigma) \) to obtain \( \sigma_1 \);
2. Remove from \( \theta \) any binding \( x \mapsto t \), where \( x \in \text{Dom}(\sigma) \), to obtain \( \theta_1 \);
3. Remove from \( \sigma_1 \) any trivial binding \( x \mapsto x \), to obtain \( \sigma_2 \); and
4. Take the union of the two sets of bindings $\sigma_2$ and $\theta_1$.

It is also useful to be able to represent substitutions in their triangular form as a sequential list of bindings, e.g.,

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

which represents the composition of $n$ substitutions each consisting of a single binding:

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

A substitution is idempotent if $\sigma \sigma = \sigma$; it is easy to show that this is true iff $\text{Dom}(\sigma) \cap \forall \text{Ran}(\sigma) = \emptyset$.

A variable renaming substitution is defined as a substitution $\sigma$ such that $\text{Dom}(\sigma) = \text{Ran}(\sigma)$. (For example, $\{x \mapsto y, y \mapsto z, z \mapsto x\}$ is a variable renaming, whereas $\{x \mapsto y\}$ and $\{y \mapsto z, x \mapsto z\}$ are not.) For any such variable renaming $\rho = \{x_1 \mapsto y_1, \ldots, x_n \mapsto y_n\}$, we denote its inverse $\{y_1 \mapsto x_1, \ldots, y_n \mapsto x_n\}$ by $\rho^{-1}$.

Two substitutions are equal, denoted $\sigma = \theta$, if $x \sigma = x \theta$ for every variable $x$. We say that $\sigma$ is more general than $\theta$, denoted $\sigma \leq \theta$, if there exists an $\eta$ such that $\theta = \sigma \eta$. The relation $\leq$ is called the instantiation quasi-ordering. The corresponding equivalence relation (i.e., $\leq \cap \geq$) is denoted by $\equiv$; it can be shown [Lassez, Maher and Mariott 1987] that $\sigma \equiv \theta$ iff there exists some variable renaming $\rho$ such that $\sigma = \theta \rho$.

2.1. DEFINITION. A substitution $\sigma$ is a unifier of two terms $s$ and $t$ if $s \sigma = t \sigma$; it is a most general unifier (or mgu for short), if for every unifier $\theta$ of $s$ and $t$, $\sigma \leq \theta$. A unification problem for two terms $s$ and $t$ is represented by $s \neq t$.

A multiset is an unordered collection with possible duplicate elements. We denote the number of occurrences of an object $x$ in a multiset $M$ by $M(x)$, and define the multiset union $M \cup N$ as the multiset $Q$ such that $Q(x) = M(x) + N(x)$ for every $x$.

2.2. Unification of terms

In this section and the next, we present a series of algorithms for unification, each of which returns an mgu for two unifiable terms. Our approach will be two-sided: on the one hand we will present a series of practical algorithms, from the "naive" to the more sophisticated (and faster), in pseudo-code suitable for implementing in a programming language; and on the other we will present a "rule-based" approach which serves to clarify the essential properties of the process and also to prove the correctness of some of the practical algorithms.

2.2.1. A naive algorithm

The simplest algorithm for unification is perhaps one that is taught in many introductory courses in AI:
Write down two terms and set markers (e.g., two index fingers) at the beginning of the terms. Then:

1. Move the markers together, one symbol at a time, until both move off the end of the term (success!), or until they point to two different symbols;

2. If the two symbols are both non-variables, then fail; otherwise, one is a variable (call it \( x \)) and the other is the first symbol of a subterm (call it \( t \)):
   (a) If \( x \) occurs in \( t \), then fail;
   (b) Otherwise, write down "\( x \mapsto t \)" as part of the solution, replace \( x \) everywhere by \( t \) (including in the solution), and return to (1).

This simple algorithm methodically finds disagreements in the two terms to be unified, and attempts to repair them by binding variables to terms: it fails when function symbols clash, or when an attempt is made to unify a variable with a term containing that variable (which is impossible). Already present in this simple algorithm are several interesting issues:

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

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

**Complexity:** How much space does this take, and how much time?

In the remainder of this section we will consider these issues in detail while developing our sequence of algorithms.

### 2.2.2. Unification by recursive descent

If we take our naive algorithm and implement it as simply as possible in a programming language, then we would represent terms using either explicit pointer structures (as in C or Pascal) or built-in recursive data types (as in ML and Lisp), and represent substitutions as lists of pairs of terms. Application of a substitution would involve constructing a new term or replacing a variable with a new term. The left-to-right search for disagreements would then be implemented by recursive descent through the terms as shown in Figure 1.

(In an actual implementation, the case "Unify( \( t, s \) )" could be moved up before the first "else if" and simply swap \( s \) and \( t \) if the former is not a variable.) The only detail that might cause some confusion is the exact method for implementing the composition in the last line. This was described in section 2.1; however, in our naive unification algorithm, we omitted the second and third steps from the informal algorithm for composition, and this may be done as well here, due to a simple but important fact about these algorithms: when a binding \( x \mapsto t \) is created and applied, \( x \) will never appear in another term considered by the algorithm—\( x \) has been "eliminated" and occurs only once, in the solution.

This algorithm is essentially the one first described by Robinson [1965], and has been almost universally used in symbolic computation systems.
global $\sigma$ : substitution;  \{ Initialized to Id \}

Unify( s : term; t : term )
begin
    if s is a variable then  \{ Instantiate variables \}
        $s := s\sigma$;
    if t is a variable then
        $t := t\sigma$;
    if s is a variable and $s = t$ then
        \{ Do nothing \}
    else if $s = f(s_1, \ldots, s_n)$ and $t = g(t_1, \ldots, t_m)$ for $n, m \geq 0$ then begin
        if $f = g$ then
            for $i := 1$ to $n$ do
                Unify( $s_i$, $t_i$ );
        else Exit with failure  \{ Symbol clash \}
    end
    else if s is not a variable then
        Unify( t, s );
    else if s occurs in t then
        Exit with failure;  \{ Occurs check \}
    else $\sigma := \sigma\{ s \mapsto t \}$;
end;

Figure 1: Unification by recursive descent

2.2.3. A *rule-based* approach $\mathcal{U}$

In order to explore some of the logical properties of this algorithm, we now present a simple inference system for deriving solutions for unification problems.

An idempotent substitution \{ $x_1 \mapsto t_1, \ldots, x_n \mapsto t_n$ \} may be represented by a set of equations \{ $x_1 \approx t_1, \ldots, x_n \approx t_n$ \} in *solved form*, i.e., where each $x_i$ has a single occurrence in the set. For any idempotent substitution $\sigma$, the corresponding solved form set will be denoted by $[\sigma]$, and for any set of equations $S$ in solved form, the corresponding substitution will be denoted by $\sigma_S$.

A *system* is either the symbol $\perp$ (representing failure) or a pair consisting of a multiset $P$ of unification problems and a set $S$ of equations in solved form. We will use $\Gamma$ to denote an arbitrary system. A substitution is said to be a unifier (or solution) of a system $P; S$ if it unifies each of the equations in $P$ and $S$; the system $\perp$ has no unifiers.

The inference system $\mathcal{U}$ consists of the following transformations on systems:\footnote{The symbol $\cup$ below when applied to $P$ is *multiset union*.}