An Efficient Unification Algorithm

ALBERTO MARTELLI
Consiglio Nazionale delle Ricerche
and
UGO MONTANARI
Università di Pisa

The unification problem in first-order predicate calculus is described in general terms as the solution of a system of equations, and a nondeterministic algorithm is given. A new unification algorithm, characterized by having the acyclicity test efficiently embedded into it, is derived from the nondeterministic one, and a PASCAL implementation is given. A comparison with other well-known unification algorithms shows that the algorithm described here performs well in all cases.

Categories and Subject Descriptors: F.2.2 [**Analysis of Algorithms and Problem Complexity**]: Nonnumerical Algorithms and Problems—*complexity of proof procedures*; F.4.1 [**Mathematical Logic and Formal Languages**]: Mathematical Logic—*mechanical theorem proving*; I.2.3 [**Artificial Intelligence**]: Deduction and Theorem Proving—*resolution*

General Terms: Algorithms, Languages, Performance, Theory

1. INTRODUCTION

In its essence, the unification problem in first-order logic can be expressed as follows: Given two terms containing some variables, find, if it exists, the simplest substitution (i.e., an assignment of some term to every variable) which makes the two terms equal. The resulting substitution is called the *most general unifier* and is unique up to variable renaming.

Unification was first introduced by Robinson [17, 18] as the central step of the inference rule called resolution. This single, powerful rule can replace all the axioms and inference rules of the first-order predicate calculus and thus was immediately recognized as especially suited to mechanical theorem provers. In fact, a number of systems based on resolution were built and tried on a variety of different applications [5]. Even though further research made it apparent that resolution systems are difficult to direct during proof search and thus are often prone to combinatorial explosion [6], new impetus to the research in this area was given by Kowalski's idea of interpreting predicate logic as a programming language [10]. Here predicate logic clauses are seen as procedure declarations, and procedure invocation represents a resolution step. From this viewpoint, theorem provers can be regarded as interpreters for programs written in predicate logic, and this analogy suggests efficient implementations [3, 25].

Authors' present addresses: A. Martelli, Istituto di Scienze della Informazione, Università di Torino, Corso M. d'Azeglio 42, I-10125 Torino, Italy; U. Montanari, Istituto di Scienze della Informazione, Università di Pisa, Corso Italia 40, I-56100 Pisa, Italy.

Resolution, however, is not the only application of the unification algorithm. In fact, its pattern matching nature can be exploited in many cases where symbolic expressions are dealt with, such as, for instance, in interpreters for equation languages [4, 11], in systems using a database organized in terms of productions [19], in type checkers for programming languages with a complex type structure [14], and in the computation of critical pairs for term rewriting systems [9].

The unification algorithm constitutes the heart of all the applications listed above, and thus its performance affects in a crucial way the global efficiency of each. The unification algorithm as originally proposed can be extremely inefficient; therefore, many attempts have been made to find more efficient algorithms [2, 7, 13, 15, 16, 22]. Unification algorithms have also been extended to the case of higher order logic [8] and to deal directly with associativity and commutativity [20]. The problem was also tackled from a computational complexity point of view, and linear algorithms were proposed independently by Martelli and Montanari [13] and Paterson and Wegman [15].

In the next section we give some basic definitions by representing the unification problem as the solution of a system of equations. A nondeterministic algorithm, which comprehends as special cases most known algorithms, is then defined and proved correct. In Section 3 we present a new version of this algorithm obtained by grouping together all equations with some member in common, and we derive from it a first version of our unification algorithm.

In Sections 4 and 5 we present the main ideas which make the algorithm efficient, and the last details are described in Section 6 by means of a PASCAL implementation.

Finally, in Section 7, the performance of this algorithm is compared with that of two well-known algorithms, Huet's [7] and Paterson and Wegman's [15]. This analysis shows that our algorithm has uniformly good performance for all classes of data considered.

2. UNIFICATION AS THE SOLUTION OF A SET OF EQUATIONS: A NONDETERMINISTIC ALGORITHM

In this section we introduce the basic definitions and give a few theorems which are useful in proving the correctness of the algorithms. Our ay of stating the unification problem is slightly more general than the classical one due to Robinson [18] and directly suggests a number of possible solution methods.

Let

$$A = \bigcup_{i=0,1,\ldots} A_i \qquad (A_i \cap A_j = \emptyset,\ i \neq j)$$

be a ranked alphabet, where A_i contains the i-adic function symbols (the elements of A_0 are constant symbols). Furthermore, let V be the alphabet of the variables. The *terms* are defined recursively as follows:

(1) constant symbols and variables are terms;
(2) if $t_1, \ldots, t_n$ $(n \geq 1)$ are terms and $f \in A_n$, then $f(t_1, \ldots, t_n)$ is a term.

A *substitution* ϑ is a mapping from variables to terms, with $\vartheta(x) = x$ almost everywhere. A substitution can be represented by a finite set of ordered pairs

$\vartheta = \{(t_1, x_1), (t_2, x_2), \ldots, (t_m, x_m)\}$ where t_i are terms and x_i are distinct variables, $i = 1, \ldots, m$. To apply a substitution ϑ to a term t, we simultaneously substitute all occurrences in t of every variable x_i in a pair (t_i, x_i) of ϑ with the corresponding term t_i. We call the resulting term t_ϑ.

For instance, given a term $t = f(x_1, g(x_2), a)$ and a substitution $\vartheta = \{(h(x_2), x_1), (b, x_2)\}$, we have $t_\vartheta = f(h(x_2), g(b), a)$ and $t_{\vartheta\vartheta} = f(h(b), g(b), a)$.

The standard unification problem can be written as an equation

$$t' = t''.$$

A solution of the equation, called a *unifier*, is any substitution ϑ, if it exists, which makes the two terms identical. For instance, two unifiers of the equation $f(x_1, h(x_1), x_2) = f(g(x_3), x_4, x_3)$ are $\vartheta_1 = \{(g(x_3), x_1), (x_3, x_2), (h(g(x_3)), x_4)\}$ and $\vartheta_2 = \{(g(a), x_1), (a, x_2), (a, x_3), (h(g(a)), x_4)\}$.

In what follows it is convenient also to consider sets of equations

$$t'_j = t''_j, \qquad j = 1, \ldots, k.$$

Again, a *unifier* is any substitution which makes all pairs of terms t'_j, t''_j identical simultaneously.

Now we are interested in finding transformations which produce *equivalent* sets of equations, namely, transformations which preserve the sets of all unifiers. Let us introduce the following two transformations:

(1) *Term Reduction.* Let

$$f(t'_1, t'_2, \ldots, t'_n) = f(t''_1, t''_2, \ldots, t''_n), \qquad f \in A_n, \tag{1}$$

be an equation where both terms are not variables and where the two root function symbols are equal. The new set of equations is obtained by replacing this equation with the following ones:

$$\begin{aligned} t'_1 &= t''_1 \\ t'_2 &= t''_2 \\ &\vdots \\ t'_n &= t''_n. \end{aligned} \tag{2}$$

If $n = 0$, then f is a constant symbol, and the equation is simply erased.

(2) *Variable Elimination.* Let $x = t$ be an equation where x is a variable and t is any term (variable or not). The new set of equations is obtained by applying the substitution $\vartheta = \{(t, x)\}$ to both terms of all other equations in the set (without erasing $x = t$).

We can prove the following theorems:

THEOREM 2.1. *Let S be a set of equations and let $f'(t'_1, \ldots, t'_n) = f''(t''_1, \ldots, t''_n)$ be an equation of S. If $f' \neq f''$, then S has no unifier. Otherwise, the new set of equations S', obtained by applying term reduction to the given equation, is equivalent to S.*

PROOF. If $f' \neq f''$, then no substitution can make the two terms identical. If $f' = f''$, any substitution which satisfies (2) also satisfies (1), and conversely for the recursive definition of term. □

THEOREM 2.2. *Let S be a set of equations, and let us apply variable elimination to some equation $x = t$, getting a new set of equations S'. If variable x occurs in t (but t is not x), then S has no unifier; otherwise, S and S' are equivalent.*

PROOF. Equation $x = t$ belongs both to S and to S', and thus any unifier ϑ (if it exists) of S or of S' must unify x and t; that is, x_ϑ and t_ϑ are identical. Now let $t_1 = t_2$ be any other equation of S, and let $t'_1 = t'_2$ be the corresponding equation in S'. Since t'_1 and t'_2 have been obtained by substituting t for every occurrence of x in t_1 and t_2, respectively, we have $t_{1_\vartheta} = t'_{1_\vartheta}$ and $t_{2_\vartheta} = t'_{2_\vartheta}$. Thus, any unifier of S is also a unifier of S' and vice versa. Furthermore, if variable x occurs in t (but t is not x), then no substitution ϑ can make x and t identical, since x_ϑ becomes a subterm of t_ϑ, and thus S has no unifier. □

A set of equations is said to be *in solved form* iff it satisfies the following conditions:

(1) the equations are $x_j = t_j, j = 1, \ldots, k$;
(2) every variable which is the left member of some equation occurs only there.

A set of equations in solved form has the obvious unifier

$$\vartheta = \{(t_1, x_1), (t_2, x_2), \ldots, (t_k, x_k)\}.$$

If there is any other unifier, it can be obtained as

$$\sigma = \{(t_{1_\alpha}, x_1), (t_{2_\alpha}, x_2), \ldots, (t_{k_\alpha}, x_k)\} \cup \alpha$$

where α is any substitution which does not rewrite variables $x_1, \ldots, x_k$. Thus ϑ is called a *most general unifier* (*mgu*).

The following nondeterministic algorithm shows how a set of equations can be transformed into an equivalent set of equations in solved form.

Algorithm 1

Given a set of equations, repeatedly perform any of the following transformations. If no transformation applies, stop with success.

(a) Select any equation of the form

$$t = x$$

where t is not a variable and x is a variable, and rewrite it as

$$x = t.$$

(b) Select any equation of the form

$$x = x$$

where x is variable, and erase it.

(c) Select any equation of the form

$$t' = t''$$

where t' and t'' are not variables. If the two root function symbols are different, stop with failure; otherwise, apply term reduction.

(d) Select any equation of the form

$$x = t$$

where x is a variable which occurs somewhere else in the set of equations and where $t \neq x$. If x occurs in t, then stop with failure; otherwise, apply variable elimination.

As an example, let us consider the following set of equations:

$$g(x_2) = x_1;$$
$$f(x_1, h(x_1), x_2) = f(g(x_3), x_4, x_3).$$

By applying transformation (c) of Algorithm 1 to the second equation we get

$$g(x_2) = x_1;$$
$$x_1 = g(x_3);$$
$$h(x_1) = x_4;$$
$$x_2 = x_3.$$

By applying transformation (d) to the second equation we get

$$g(x_2) = g(x_3);$$
$$x_1 = g(x_3);$$
$$h(g(x_3)) = x_4;$$
$$x_2 = x_3.$$

We now apply transformation (c) to the first equation and transformation (a) to the third equation:

$$x_2 = x_3;$$
$$x_1 = g(x_3);$$
$$x_4 = h(g(x_3));$$
$$x_2 = x_3.$$

Finally, by applying transformation (d) to the first equation and transformation (b) to the last equation, we get the set of equations in solved form:

$$x_2 = x_3;$$
$$x_1 = g(x_3);$$
$$x_4 = h(g(x_3)).$$

Therefore, an mgu of the given system is

$$\vartheta = \{(g(x_3), x_1), (x_3, x_2), (h(g(x_3)), x_4)\}.$$

The following theorem proves the correctness of Algorithm 1.

THEOREM 2.3. *Given a set of equations S,*

(*i*) *Algorithm* 1 *always terminates, no matter which choices are made.*
(*ii*) *If Algorithm* 1 *terminates with failure, S has no unifier. If Algorithm* 1 *terminates with success, the set S has been transformed into an equivalent set in solved form.*

PROOF.

(i) Let us define a function F mapping any set of equations S into a triple of natural numbers (n_1, n_2, n_3). The first number, n_1, is the number of variables in S which *do not* occur only once as the left-hand side of some equation. The second number, n_2, is the total number of occurrences of function symbols in S. The third number, n_3, is the sum of the numbers of equations in S of type $x = x$ and of type $t = x$, where x is a variable and t is not. Let us define a total ordering on such triples as follows:

$$(n_1', n_2', n_3') > (n_1'', n_2'', n_3'') \quad \text{if } n_1' > n_1''$$
$$\text{or } n_1' = n_1'' \text{ and } n_2' > n_2''$$
$$\text{or } n_1' = n_1'' \text{ and } n_2' = n_2'' \text{ and } n_3' > n_3''.$$

With the above ordering, N^3 becomes a well-founded set, that is, a set where no infinite decreasing sequence exists. Thus, if we prove that any transformation of Algorithm 1 transforms a set S in a set S' such that $F(S') < F(S)$, we have proved the termination. In fact, transformations (a) and (b) always decrease n_3 and, possibly, n_1. Transformation (c) can possibly increase n_3 and decrease n_1, but it surely decreases n_2 (by two). Transformation (d) can possibly change n_3 and increase n_2, but it surely decreases n_1.

(ii) If Algorithm 1 terminates with failure, the thesis immediately follows from Theorems 2.1 and 2.2. If Algorithm 1 terminates with success, the resulting set of equations S' is equivalent to the given set S. In fact, transformations (a) and (b) clearly do not change the set of unifiers, while for transformations (c) and (d) this fact is stated in Theorems 2.1 and 2.2. Finally, S' is in solved form. In fact, if (a), (b), and (c) cannot be applied, it means that the equations are all in the form $x = t$, with $t \neq x$. If (d) cannot be applied, that means that every variable which is the left-hand side of some equation occurs only there. □

The above nondeterministic algorithm provides a widely general version from which most unification algorithms [2, 3, 7, 13, 15, 16, 18, 22–24] can be derived by specifying the order in which the equations are selected and by defining suitable concrete data structures. For instance, Robinson's algorithm [18] might be obtained by considering the set of equations as a stack.

3. AN ALGORITHM WHICH EXPLOITS A PARTIAL ORDERING AMONG SETS OF VARIABLES

3.1 Basic Definitions

In this section we present an extension of the previous formalism to model our algorithm more closely. We first introduce the concept of multiequation. A *multiequation* is the generalization of an equation, and it allows us to group together many terms which should be unified. To represent multiequations we use the notation $S = M$ where the left-hand side S is a nonempty set of variables and the right-hand side M is a multiset[1] of nonvariable terms. An example is

$$\{x_1, x_2, x_3\} = (t_1, t_2).$$

[1] A multiset is a family of elements in which no ordering exists but in which many identical elements may occur.

The solution (unifier) of a multiequation is any substitution which makes all terms in the left- and right-hand sides identical.

A multiequation can be seen as a way of grouping many equations together. For instance, the set of equations

$$x_1 = x_2;$$

$$x_3 = x_1;$$

$$t_1 = x_1;$$

$$x_2 = t_2;$$

$$t_1 = t_2$$

can be transformed into the above multiequation, since every unifier of this set of equations makes the terms of all equations identical. To be more precise, given a set of equations SE, let us define a relation R_{SE} between pairs of terms as follows: $t_1 \, R_{SE} \, t_2$ iff the equation $t_1 = t_2$ belongs to SE. Let $\bar{R}_{SE}$ be the reflexive, symmetric, and transitive closure of R_{SE}.

Now we can say that a set of equations SE *corresponds* to a multiequation $S = M$ iff all terms of SE belong to $S \cup M$ and for every t_r and $t_s \in S \cup M$ we have $t_r \, \bar{R}_{SE} \, t_s$.

It is easy to see that many different sets of equations may correspond to a given multiequation and that all these sets are equivalent. Thus the set of solutions (unifiers) of a multiequation coincides with the set of solutions of any corresponding set of equations.

Similar definitions can be given for a set of multiequations Z by introducing a relation R_Z between pairs of terms which belong to the same multiequation. A set of equations SE *corresponds* to a set of multiequations Z iff

$$t_i \, \bar{R}_{SE} \, t_j \Leftrightarrow t_i \, \bar{R}_Z \, t_j$$

for all terms t_i, t_j of SE or Z.

3.2 Transformations of Sets of Multiequations

We now introduce a few transformations of sets of multiequations, which are generalizations of the transformations presented in Section 2.

We first define the *common part* and the *frontier* of a multiset of terms (variables or not). The common part of a multiset of terms M is a term which, intuitively, is obtained by superimposing all terms of M and by taking the part which is common to all of them starting from the root. For instance, given the multiset of terms

$$(f(x_1, g(a, f(x_5, b))), f(h(c), g(x_2, f(b, x_5))), f(h(x_4), g(x_6, x_3))),$$

the common part is

$$f(x_1, g(x_2, x_3)).$$

The frontier is a set of multiequations, where every multiequation is associated with a leaf of the common part and consists of all subterms (one for each term of

M) corresponding to that leaf. The frontier of the above multiset of terms is

$$\{\{x_1\} = (h(c), h(x_4)),$$
$$\{x_2, x_6\} = (a),$$
$$\{x_3\} = (f(x_5, b), f(b, x_5))\}.$$

Note that if there is a clash of function symbols among some terms of a multiset of terms M, then M has no common part and frontier. In this case the terms of M are not unifiable.

The common part and the frontier can be defined more precisely by means of a function DEC which takes a multiset of terms M as argument and returns either "failure," in which case M has neither common part nor frontier, or a pair $\langle C(M), F(M)\rangle$ where $C(M)$ is the common part of M and $F(M)$ is the frontier of M.

In the definition of DEC we use the following notation:

head(t) is the root function symbol of term t, for $t \notin V$.

P_i is the ith projection, defined by

$$P_i(f(t_1, \ldots, t_n)) = t_i \quad \text{for} \quad f \in A_n \quad \text{and} \quad 1 \le i \le n;$$

makemulteq is a function which transforms a multiset of terms M into a multiequation whose left-hand side is the set of all variables in M and whose right-hand side is the multiset of all terms in M which are not variables; and

$\uplus$ is the union for multisets.

DEC(M) = **if** $\exists t \in M, t \in V$
then $\langle t, \{\text{makemulteq}(M)\}\rangle$
else if $\exists n, \exists f \in A_n, \forall t \in M, \text{head}(t) = f$
then if $n = 0$
then $\langle f, \varnothing\rangle$
else if $\forall i\ (1 \le i \le n)$, DEC($M_i$) $\neq$ failure
where $M_i = \uplus_{t \in M} P_i(t)$
then $\langle f(C(M_1), \ldots, C(M_n)), \cup_{i=1}^{n} F(M_i)\rangle$
else failure
else failure.

We can now define the following transformation:

Multiequation Reduction. Let Z be a set of multiequations containing a multiequation $S = M$ such that M is nonempty and has a common part C and a frontier F. The new set Z' of multiequations is obtained by replacing $S = M$ with the union of the multiequation $S = (C)$ and of all the multiequations of F:

$$Z' = (Z - \{S = M\}) \cup \{S = (C)\} \cup F.$$

THEOREM 3.1. *Let $S = M$ (M nonempty) be a multiequation of a set Z of multiequations. If M has no common part, or if some variable in S belongs to the left-hand side of some multiequation in the frontier F of M, then Z has no*

unifier. Otherwise, by applying multiequation reduction to the multiequation $S = M$ *we get an equivalent set* Z' *of multiequations.*

PROOF. If the common part of M does not exist, then the multiequation $S = M$ has no unifier, since two terms should be made equal having a different function symbol in the corresponding subterms. Moreover, if some variable x of S occurs in some left-hand side of the frontier, then it also occurs in some term t of M, and thus the equation $x = t$, with x occurring in t, belongs to a set of equations equivalent to Z. But, according to Theorem 2.2, this set has no unifier.

To prove that Z and Z' are equivalent, we show first that a unifier of Z is also a unifier of Z'. In fact, if a substitution ϑ makes all terms of M equal, it also makes equal all the corresponding subterms, in particular, all terms and variables which belong to left- and right-hand sides of the same multiequation in the frontier. The multiequation $S = (C)$ is also satisfied by construction. Conversely, if ϑ satisfies Z', then the multiequation $S = M$ is also satisfied. In fact, all terms in S and M are made equal—in their upper part (the common part) due to the multiequation $S = (C)$ and in their lower part (the subterms not included in the common part) due to the set of multiequations F. □

We say that a set Z of multiequations is *compact* iff

$$\forall (S = M), (S' = M') \in Z: S \cap S' = \emptyset.$$

We can now introduce a second transformation, which derives a compact set of multiequations.

Compactification. Let Z be a noncompact set of multiequations. Let R be a relation between pairs of multiequations of Z such that $(S = M)\ R\ (S' = M')$ iff $S \cap S' \neq \emptyset$, and let $\bar{R}$ be the transitive closure of R. The relation $\bar{R}$ partitions the set Z into equivalence classes. To obtain the final compact set Z', all multiequations belonging to the same equivalence class are merged; that is, they are transformed into single multiequations by taking the union of their left- and right-hand sides.

Clearly, Z and Z' are equivalent, because the relation $\bar{R}_Z$ between pairs of terms, defined in Section 3.1, does not change by passing from Z to Z'.

3.3 Solving Systems of Multiequations

For convenience, in what follows, we want to give a structure to a set of multiequations. Thus we introduce the concept of *system of multiequations.* A system R is a pair (T, U), where T is a sequence and U is a set of multiequations (either possibly empty), such that

(1) the sets of variables which constitute the left-hand sides of all multiequations in both T and U contain all variables and are disjoint;
(2) the right-hand sides of all multiequations in T consist of no more than one term; and
(3) all variables belonging to the left-hand side of some multiequation in T can *only* occur in the right-hand side of any preceding multiequation in T.

We now present an algorithm for solving a given system R of multiequations. When the computation starts, the T part is empty, and every step of the following

Algorithm 2 consists of "transferring" a multiequation from the U part, that is, the *unsolved* part, to the T part, that is, the *triangular* or *solved* part of R. When the U part of R is empty, the system is essentially solved. In fact, the solution can be obtained by substituting the variables backward. Notice that, by keeping a solved system in this triangular form, we can hope to find efficient algorithms for unification even when the mgu has a size which is exponential with respect to the size of the initial system. For instance, the mgu of the set of multiequations

$$\{\{x_1\} = \emptyset,$$
$$\{x_2\} = \emptyset,$$
$$\{x_3\} = \emptyset,$$
$$\{x_4\} = (h(x_3, h(x_2, x_2)), h(h(h(x_1, x_1), x_2), x_3))\}$$

is

$$\{(h(x_1, x_1), x_2), (h(h(x_1, x_1), h(x_1, x_1)), x_3),$$
$$(h(h(h(x_1, x_1), h(x_1, x_1)), h(h(x_1, x_1), h(x_1, x_1))), x_4)\}.$$

However, we can give an equivalent solved system with empty U part and whose T part is

$$(\{x_4\} = (h(x_3, x_3)),$$
$$\{x_3\} = (h(x_2, x_2)),$$
$$\{x_2\} = (h(x_1, x_1)),$$
$$\{x_1\} = \emptyset),$$

from which the mgu can be obtained by substituting backward.

Given a system $R = (T, U)$ with an empty T part, an equivalent system with an empty U part can be computed with the following algorithm.

Algorithm 2

(1) **repeat**
  (1.1) Select a multiequation $S = M$ of U with $M \neq \emptyset$.
  (1.2) Compute the common part C and the frontier F of M. If M has no common part, stop with failure (clash).
  (1.3) If the left-hand sides of the frontier of M contain some variable of S, stop with failure (cycle).
  (1.4) Transform U using multiequation reduction on the selected multiequation and compactification.
  (1.5) Let $S = \{x_1, \ldots, x_n\}$. Apply the substitution $\vartheta = \{(C, x_1), \ldots, (C, x_n)\}$ to all terms in the right-hand side of the multiequations of U.
  (1.6) Transfer the multiequation $S = (C)$ from U to the end of T.
  until the U part of R contains only multiequations, if any, with empty right-hand sides.
(2) Transfer all the multiequations of U (all with $M = \emptyset$) to the end of T, and stop with success.

Of course, if we want to use this algorithm for unifying two terms t_1 and t_2, we have to construct an initial system with empty T part and with the following U part:

$$\{\{x\} = (t_1, t_2), \{x_1\} = \emptyset, \{x_2\} = \emptyset, \ldots, \{x_n\} = \emptyset\}$$

where $x_1, x_2, \ldots, x_n$ are all the variables in t_1 and t_2 and x is a new variable which does not occur in t_1 and t_2. For instance, let $t_1 = f(x_1, g(x_2, x_3), x_2, b)$ and $t_2 = f(g(h(a, x_5), x_2), x_1, h(a, x_4), x_4)$. The initial system is as follows:

$$\begin{aligned} U\text{: } &\{\{x\} = (f(x_1, g(x_2, x_3), x_2, b), f(g(h(a, x_5), x_2), x_1, h(a, x_4), x_4)), \\ &\{x_1\} = \emptyset, \{x_2\} = \emptyset, \{x_3\} = \emptyset, \{x_4\} = \emptyset, \{x_5\} = \emptyset\}; \\ T\text{: } &(\,). \end{aligned} \tag{3}$$

After the first iteration of Algorithm 2 we get

$$\begin{aligned} U\text{: } &\{\{x_1\} = (g(h(a, x_5), x_2), g(x_2, x_3)), \\ &\{x_2\} = (h(a, x_4)), \\ &\{x_3\} = \emptyset, \\ &\{x_4\} = (b), \\ &\{x_5\} = \emptyset\}; \\ T\text{: } &(\{x\} = (f(x_1, x_1, x_2, x_4))). \end{aligned}$$

We now eliminate variable x_2, obtaining

$$\begin{aligned} U\text{: } &\{\{x_1\} = (g(h(a, x_5), h(a, x_4)), g(h(a, x_4), x_3)), \\ &\{x_3\} = \emptyset, \\ &\{x_4\} = (b), \\ &\{x_5\} = \emptyset\}; \\ T\text{: } &(\{x\} = (f(x_1, x_1, x_2, x_4)), \\ &\{x_2\} = (h(a, x_4))). \end{aligned}$$

By eliminating variable x_1, we get

$$\begin{aligned} U\text{: } &\{\{x_3\} = (h(a, x_4)), \\ &\{x_4, x_5\} = (b)\}; \\ T\text{: } &(\{x\} = (f(x_1, x_1, x_2, x_4)), \\ &\{x_2\} = (h(a, x_4)), \\ &\{x_1\} = (g(h(a, x_4), x_3))). \end{aligned}$$

Finally, by eliminating first the set $\{x_4, x_5\}$ and then $\{x_3\}$, we get the solved system

$$\begin{aligned} U\text{: } &\emptyset; \\ T\text{: } &(\{x\} = (f(x_1, x_1, x_2, x_4)), \\ &\{x_2\} = (h(a, x_4)), \\ &\{x_1\} = (g(h(a, x_4), x_3)), \\ &\{x_4, x_5\} = (b), \\ &\{x_3\} = (h(a, b))). \end{aligned}$$

We can now prove the correctness of Algorithm 2.

THEOREM 3.2. *Algorithm 2 always terminates. If it stops with failure, then the given system has no unifier. If it stops with success, the resulting system is equivalent to the given system and has an empty unsolved part.*

PROOF. All transformations obtain systems equivalent to the given one. In fact, in step (1.4) multiequation reduction obtains a set of equations which (according to Theorem 3.1) is equivalent, and compactification transforms it again into a system. Step (1.5) applies substitution only to the terms in U, and its feasibility can be proved as in Theorem 2.2. Step (1.6) can be applied since the multiequation $S = (C)$, introduced during multiequation reduction, has not been modified by compactification, due to the condition tested in step (1.3). For the same condition, transferring multiequation $S = (C)$ from U to T still leaves a system. Step (2) is clearly feasible.

If the algorithm stops with failure, then, by Theorem 3.1, the system presently denoted by R (equivalent to the given one) has no solution. Otherwise, the final system clearly has an empty U part. Finally, the algorithm always terminates since at every cycle some variable is eliminated from the U part. □

It is easy to see that, for a given system, the size of the final system depends heavily on the order of elimination of the multiequations. For instance, given the same system as discussed earlier,

$$\begin{aligned} U\colon\ & \{\{x_1\} = \varnothing, \\ & \{x_2\} = (h(x_1, x_1)), \\ & \{x_3\} = (h(x_2, x_2)), \\ & \{x_4\} = (h(x_3, x_3))\}; \\ T\colon\ & (\ ), \end{aligned}$$

and eliminating the variables in the order x_2, x_3, x_4, x_1, we get the final system

$$\begin{aligned} U\colon\ & \varnothing; \\ T\colon\ & (\{x_2\} = (h(x_1, x_1)), \\ & \{x_3\} = (h(h(x_1, x_1), h(x_1, x_1))), \\ & \{x_4\} = (h(h(h(x_1, x_1), h(x_1, x_1)), h(h(x_1, x_1), h(x_1, x_1)))), \\ & \{x_1\} = \varnothing). \end{aligned}$$

If instead we eliminate the variables in the order x_4, x_3, x_2, x_1, we get

$$\begin{aligned} U\colon\ & \varnothing; \\ T\colon\ & (\{x_4\} = (h(x_3, x_3)), \\ & \{x_3\} = (h(x_2, x_2)), \\ & \{x_2\} = (h(x_1, x_1)), \\ & \{x_1\} = \varnothing). \end{aligned}$$

3.4 The Unification Algorithm

Looking at Algorithm 2, it is clear that the main source of complexity is step (1.5), since it may make many copies of large terms. In the following—and this is

the heart of our algorithm—we show that, if the system has unifiers, then there always exists a multiequation in U (if not empty) such that by selecting it we do not need step (1.5) of the algorithm, since the variables in its left-hand side do not occur elsewhere in U. We need the following definition.

Given a system R, let us consider the subset V_U of variables obtained by making the union of all left-hand sides S_i of the multiequations in the U part of R. Since the sets S_i are disjoint, they determine a partition of V_U. We now define a relation on the classes S_i of this partition: we say that $S_i < S_j$ iff there exists a variable of S_i occurring in some term of M_j, where M_j is the right-hand side of the multiequation whose left-hand side is S_j. We write $<^*$ for the transitive closure of $<$.

Now we can prove the following theorem and corollary.

THEOREM 3.3. *If a system R has a unifier, then the relation $<^*$ is a partial ordering.*

PROOF. If $S_i < S_j$, then, in all unifiers of the system, the term substituted for every variable in S_i must be a strict subterm of the term substituted for every variable in S_j. Thus, if the system has a unifier, the graph of the relation $<$ cannot have cycles. Therefore, its transitive closure must be a partial ordering. □

COROLLARY. *If the system R has a unifier and its U part is nonempty, there exists a multiequation $S = M$ such that the variables in S do not occur elsewhere in U.*

PROOF. Let $S = M$ be a multiequation such that S is "on top" of the partial ordering $<^*$ (i.e., $\sim\exists S_i, S < S_i$). The variables in S occur neither in the other left-hand sides of U (since they are disjoint) nor in any right member M_i of U, since otherwise $S < S_i$. □

We can now refine the nondeterministic Algorithm 2 giving the general version of our unification algorithm for a system of multiequations $R = (T, U)$.

Algorithm 3: UNIFY, the Unification Algorithm

(1) **repeat**
  (1.1) Select a multiequation $S = M$ of U such that the variables in S do not occur elsewhere in U. If a multiequation with this property does not exist, stop with failure (cycle).
  (1.2) **if** M is empty
    then transfer this multiequation from U to the end of T.
    else begin
      (1.2.1) Compute the common part C and the frontier F of M. If M has no common part, stop with failure (clash).
      (1.2.2) Transform U using multiequation reduction on the selected multiequation and compactification.
      (1.2.3) Transfer the multiequation $S = (C)$ from U to the end of T.
      end
  until the U part of R is empty.
(2) **stop** with success.

A few comments are needed. Besides step (1.5) of Algorithm 2, we have also erased step (1.3) for the same reason. Furthermore, in Algorithm 2 we were forced to wait to transfer multiequations with empty right-hand sides since substitution in that case would have required a special treatment.

By applying Algorithm UNIFY to the system which was previously solved with Algorithm 2, we see that we must first eliminate variable x, then variable x_1, then variables x_2 and x_3 together, and, finally, variables x_4 and x_5 together, getting the following final system:

$$\begin{aligned} &U: \varnothing \\ &T: (\{x\} = (f(x_1, x_1, x_2, x_4)), \\ &\quad \{x_1\} = (g(x_2, x_3)), \\ &\quad \{x_2, x_3\} = (h(a, x_4)), \\ &\quad \{x_4, x_5\} = (b)). \end{aligned}$$

Note that the solution obtained using Algorithm UNIFY is more concise than the solution previously obtained using Algorithm 2, for two reasons. First, variables x_2 and x_3 have been recognized as equivalent; second, the right member of x_1 is more factorized. This improvement is not casual but is intrinsic in the ordering behavior of Algorithm UNIFY.

To summarize, Algorithm UNIFY is based mainly on the two ideas of keeping the solution in a factorized form and of selecting at each step a multiequation in such a way that no substitution ever has to be applied. Because of these two facts, the size of the final system cannot be larger than that of the initial one. Furthermore, the operation of selecting a multiequation fails if there are cycles among variables, and thus the so-called occur-check is built into the algorithm, instead of being performed at the last step as in other algorithms [2, 7].

4. EFFICIENT MULTIEQUATION SELECTION

In this section we show how to implement efficiently the operation of selecting a multiequation "on top" of the partial ordering in step (1.1) of Algorithm 3.

The idea is to associate with every multiequation a *counter* which contains the number of other occurrences in U of the variables in its left-hand side. This counter is initialized by scanning the whole U part at the beginning. Of course, a multiequation whose counter is set to zero is on top of the partial ordering.

For instance, let us again consider system (3):

$$\begin{aligned} U: \{&[0]\ \{x\} = (f(x_1, g(x_2, x_3), x_2, b), f(g(h(a, x_5), x_2), x_1, \\ &\qquad\qquad\qquad\qquad\qquad\qquad h(a, x_4), x_4)), \\ &[2]\ \{x_1\} = \varnothing, \\ &[3]\ \{x_2\} = \varnothing, \\ &[1]\ \{x_3\} = \varnothing, \\ &[2]\ \{x_4\} = \varnothing, \\ &[1]\ \{x_5\} = \varnothing\}; \\ T: &(\ ). \end{aligned}$$

Here square brackets enclose the counters associated with each multiequation. Since only the first multiequation has its counter set to zero, it is selected to be transferred. Counters of the other multiequations are easily updated by decre-

menting them whenever an occurrence of the corresponding variable appears in the left-hand side of a multiequation in the frontier computed in step (1.2.1). When two or more multiequations in U are merged in the compactification phase, the counter associated with the new multiequation is obviously set to a value which is the sum of the contents of the old counters.

The next steps are as follows:

$$\begin{aligned}
U:\ &\{[0]\ \{x_1\} = (g(h(a, x_5), x_2), g(x_2, x_3)),\\
&\ [2]\ \{x_2\} = (h(a, x_4)),\\
&\ [1]\ \{x_3\} = \varnothing,\\
&\ [1]\ \{x_4\} = (b),\\
&\ [1]\ \{x_5\} = \varnothing\};\\
T:\ &(\{x\} = (f(x_1, x_1, x_2, x_4))).
\end{aligned}$$

$$\begin{aligned}
U:\ &\{[0]\ \{x_2, x_3\} = (h(a, x_4), h(a, x_5)),\\
&\ [1]\ \{x_4\} = (b),\\
&\ [1]\ \{x_5\} = \varnothing\};\\
T:\ &(\{x\} = (f(x_1, x_1, x_2, x_4)),\\
&\ \{x_1\} = (g(x_2, x_3))).
\end{aligned}$$

$$\begin{aligned}
U:\ &\{[0]\ \{x_4, x_5\} = (b)\};\\
T:\ &(\{x\} = (f(x_1, x_1, x_2, x_4)),\\
&\ \{x_1\} = (g(x_2, x_3)),\\
&\ \{x_2, x_3\} = (h(a, x_4))).
\end{aligned}$$

$$\begin{aligned}
U:\ &\varnothing;\\
T:\ &(\{x\} = (f(x_1, x_1, x_2, x_4)),\\
&\ \{x_1\} = (g(x_2, x_3)),\\
&\ \{x_2, x_3\} = (h(a, x_4)),\\
&\ \{x_4, x_5\} = (b)).
\end{aligned}$$

5. IMPROVING THE UNIFICATION ALGORITHM FOR NONUNIFYING DATA

In the case of nonunifying data, Algorithm 3 can stop with failure in two ways: either in step (1.1) if a cycle has been detected, or in step (1.2.1) if a clash occurs. In this section we show how to anticipate the latter kind of failure without altering the structure of the algorithm.

Let us first give the following definition. Two terms are *consistent* iff either at least one of them is a variable or they are both nonvariable terms with the same root function symbol and pairwise consistent arguments. This definition can be extended to the case of more than two terms by saying that they are consistent iff all pairs of terms are consistent. For instance, the three terms $f(x, g(a, y))$, $f(b, x)$, and $f(x, y)$ are consistent although they are not unifiable.

We now modify Algorithm UNIFY by requiring all terms in the right-hand side of a multiequation to be consistent, for every multiequation. Thus, we stop with clash failure as soon as this requirement is not satisfied. This new version of the algorithm is still correct since, if there are two inconsistent terms in the same multiequation, they will never unify.

In this way, clashes are detected earlier. In fact, in the Algorithm 3 version of UNIFY a clash can be detected while computing the common part and the frontier of the right-hand side of the selected multiequation, whereas in the new version of UNIFY the same error is detected in the compactification phase of a previous iteration.

An efficient implementation of the consistency check when two multiequations are merged requires a suitable representation for right-hand sides of multiequations. Thus, instead of choosing the obvious solution of representing every right-hand side as a list of terms, we represent it as a multiterm, defined as follows.

A *multiterm* can be either empty or of the form $f(P_1, \ldots, P_n)$ where $f \in A_n$ and P_i $(i = 1, \ldots, n)$ is a pair $\langle S_i, M_i \rangle$ consisting of a set of variables S_i and a multiterm M_i. Furthermore, S_i and M_i cannot both be empty.

For instance, the multiset of consistent terms

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

can be represented with the multiterm

$$f(\langle \{x\}, b \rangle, \langle \{x, y\}, g(\langle \emptyset, a \rangle, \langle \{y\}, \emptyset \rangle) \rangle).$$

By representing right-hand sides in this way we have no loss of information, since the only operations which we have to perform on them are the operation of merging two right-hand sides and the operation of computing the common part and the frontier, which can be described as follows:

MERGE (M', M'') =
  case M' **of**
    $\emptyset$: M'';
    $f'(\langle S'_1, M'_1 \rangle, \ldots, \langle S'_n, M'_n \rangle)$:
      case M'' **of**
        $\emptyset$: M';
        $f''(\langle S''_1, M''_1 \rangle, \ldots, \langle S''_n, M''_n \rangle)$:
          if $f' = f''$ **and** MERGE$(M'_i, M''_i) \neq$ failure $(i = 1, \ldots, n)$
          then $f'(\langle S'_1 \cup S''_1,$ MERGE$(M'_1, M''_1) \rangle, \ldots,$
                $\langle S'_n \cup S''_n,$ MERGE$(M'_n, M''_n) \rangle)$
          else failure
      endcase
  endcase

COMMONPART$(f(\langle S_1, M_1 \rangle, \ldots, \langle S_n, M_n \rangle)) = f(P_1, \ldots, P_n)$

where $P_i =$ **if** $S_i = \emptyset$ **then** COMMONPART(M_i)
else $ANYOF(S_i)$ $\quad (i = 1, \ldots, n)$

where function ANYOF(S_i) returns an element of set S_i.

```
UPart = record
          MultEqNumber: Integer;
          ZeroCounterMultEq, Equations: ListOfMultEq
        end;
System = ↑PSystem;
PSystem = record
            T: ListOfMultEq;
            U: UPart
          end;
MultiTerm = ↑PMultiTerm;
PMultiTerm = record
               Fsymb: FunName;
               Args: ListOfTempMultEq
             end;
MultiEquation = ↑PMultiEquation;
PMultiEquation = record
                   Counter, VarNumber: Integer;
                   S: ListOfVariables;
                   M: MultiTerm
                 end;
TempMultEq = ↑PTempMultEq;
PTempMultEq = record
                S: QueueOfVariables;
                M: MultiTerm
              end;
Variable = ↑PVariable;
PVariable = record
              Name: VarName;
              M: MultiEquation
            end;
```

Figure 1

$$\text{FRONTIER}(f(\langle S_1, M_1\rangle, \ldots, \langle S_n, M_n\rangle)) = F_1 \cup \cdots \cup F_n$$

$$\text{where } F_i = \textbf{if } S_i = \varnothing \textbf{ then } \text{FRONTIER}(M_i) \textbf{ else } \{S_i = M_i\} \qquad (i = 1, \ldots, n).$$

Note that the common part and the frontier are defined only for nonempty multiterms and that they always exist.

6. IMPLEMENTATION

In order to describe the last details of our algorithm, we present here a PASCAL implementation. In Figure 1 we have the definitions of data types. All data structures used by the algorithm are dynamically created data structures connected through pointers. The *UPart* of a system has two lists of multiequations: *Equations*, which contains all initial multiequations, and *ZeroCounterMultEq*, which contains all multiequations with zero counter. Furthermore, the field *MultEqNumber* contains the number of multiequations in the *UPart*. A multiequation, besides having the fields *Counter*, *S*, and *M*, has a field *VarNumber*, which contains the number of variables in *S* and is used during compactification. The pairs $P_i = \langle S_i, M_i\rangle$, which are the arguments of a multiterm, have type

```
procedure Unify(R: System);
var Mult: MultiEquation;
    Frontier: ListOfTempMultEq;
begin
  repeat
    SelectMultiEquation(R↑.U, Mult);
    if not(Mult↑.M=Nil) then
    begin
      Frontier := Nil;
      Reduce(Mult↑.M, Frontier);
      Compact(Frontier, R↑.U)
    end;
    R↑.T := NewListOfMultEq(Mult, R↑.T)
  until R↑.U.MultEqNumber = 0
end (*Unify*);
```

Figure 2

```
procedure SelectMultiEquation(var U: UPart; var Mult: MultiEquation);
begin
  if U.ZeroCounterMultEq = Nil then fail('cycle');
  Mult := U.ZeroCounterMultEq↑.Value;
  U.ZeroCounterMultEq := U.ZeroCounterMultEq↑.Next;
  U.MultEqNumber := U.MultEqNumber - 1
end (*SelectMultiEquation*);
```

Figure 3

TempMultEq. Finally, all occurrences of a variable point to the same *Variable* object, which points to the multiequation containing it in its left-hand side.

The types "*ListOf...*," not given in Figure 1, are all implemented as records with two fields: *Value* and *Next*. Finally, *QueueOfVariables* is an abstract type with operations *CreateListOfVars*, *IsEmpty*, *HeadOf*, *RemoveHead*, and *Append*, which have a constant execution time.

In Figure 2 we rephrase Algorithm UNIFY as a PASCAL procedure. Procedure *SelectMultiEquation* selects from the *UPart* of the system a multiequation which is "on top" of the partial ordering, by taking it from the *ZeroCounterMultEq* list. Its implementation is given in Figure 3.

Procedure *Reduce*, given in Figure 4, computes the common part and the frontier of the selected multiequation. This procedure modifies the right-hand side of this multiequation so that it contains directly the common part. Note that the frontier is represented as a list of *TempMultEq* instead of as a list of multiequations.

Finally, in Figure 5 we give procedure *Compact*, which performs compactification by repeatedly merging multiequations. When two multiequations are merged, one of them is erased, and thus all pointers to it from its variables must be moved to the other multiequation. To minimize the computing cost, we always erase the multiequation with the smallest number of variables in its left-hand side. Procedure *MergeMultiTerms* is given in Figure 6.

A detailed complexity analysis of a similar implementation is given in [13]. There it is proved that an upper bound to execution time is the sum of two terms, one linear with the total number of symbols in the initial system and another one $n \log n$ with the number of *distinct* variables.

```
procedure Reduce(M: MultiTerm; var Frontier: ListOfTempMultEq);
var Arg: ListOfTempMultEq;
begin
  Arg := M↑.Args;
  while not(Arg = Nil) do
  begin
    if IsEmpty(Arg↑.Value↑.S) then Reduce(Arg↑.Value↑.M, Frontier)
    else
    begin
      Frontier := NewListOfTempMultEq(Arg↑.Value, Frontier);
      Arg↑.Value := NewTempMultEq(CreateQueueOfVars(HeadOf(Arg↑.Value↑.S)), Nil)
    end;
    Arg := Arg↑.Next
  end
end (*Reduce*);
```

Figure 4

Here we want only to point out that the nonlinear behavior stems from the operation described above of moving all pointers directed from variables to multiequations, whenever two multiequations are merged. To see how this can happen, let us consider the problem of unifying the two terms

$$f(x_1, x_3, x_5, x_7, x_1, x_5, x_1)$$

and

$$f(x_2, x_4, x_6, x_8, x_3, x_7, x_5).$$

During the first iteration of *Unify* we get a frontier whose multiequations are the pairs (x_1, x_2), (x_3, x_4), (x_5, x_6), (x_7, x_8), (x_1, x_3), (x_5, x_7), and (x_1, x_5). By executing *Compact* with this frontier, we see that it moves one pointer for each of the first four elements of the frontier, two pointers for each of the next two elements, and four pointers for the last element. Thus, it has an $n \log n$ complexity.

As a final remark, we point out that we might modify the worst-case behavior of our algorithm with a different implementation of the operation of multiequation merging. In fact, we might represent sets of variables as trees instead of as lists, and we might use the well-known UNION–FIND algorithms [1] to add elements and to access them. In this case the complexity would be of the order of $m \cdot G(m)$, where G is an extremely slowly growing function (the inverse of the Ackermann function). However, m would be, in this case, the number of variable *occurrences.*

7. COMPARISONS WITH OTHER ALGORITHMS

In this section we compare the performance of our algorithm with that of two well-known algorithms: Huet's algorithm [7], which has an almost linear time complexity, and Paterson and Wegman's algorithm [15], which is theoretically the best having a linear complexity.

As an example of the assertion made at the end of Section 2, let us give a sketchy description of the two algorithms using the terminology of this paper. Both algorithms deal with sets of multiequations whose left-hand sides are disjoint and whose right-hand sides consist of only one term of depth one, that is,

```
procedure Compact(Frontier: ListOfTempMultEq; var U: UPart);
var Vars: QueueOfVariables;
    V: Variable;
    Mult, Mult1: MultiEquation;
procedure MergeMultEq(var Mult: MultiEquation; Mult1: MultiEquation);
var Multt: MultiEquation;
    V: Variable;
    Vars: ListOfVariables;
begin
  if not(Mult = Mult1) then
  begin
    if Mult↑.VarNumber < Mult1↑.VarNumber then
    begin
      Multt := Mult;
      Mult := Mult1;
      Mult1 := Multt
    end;
    Mult↑.Counter := Mult↑.Counter + Mult1↑.Counter;
    Mult↑.VarNumber := Mult↑.VarNumber + Mult1↑.VarNumber;
    Vars := Mult1↑.S;
    repeat
      V := Vars↑.Value;
      Vars := Vars↑.Next;
      V↑.M := Mult;
      Mult↑.S := NewListOfVariables(V, Mult↑.S)
    until Vars = Nil;
    MergeMultiTerms(Mult↑.M, Mult1↑.M);
    U.MultEqNumber := U.MultEqNumber − 1
  end
end (*MergeMultEq*);
begin
  while not(Frontier = Nil) do
  begin
    Vars := Frontier↑.Value↑.S;
    V := HeadOf(Vars);
    RemoveHead(Vars);
    Mult := V↑.M;
    Mult↑.Counter := Mult↑.Counter − 1;
    while not IsEmpty(Vars) do
    begin
      V := HeadOf(Vars);
      RemoveHead(Vars);
      Mult1 := V↑.M;
      Mult1↑.Counter := Mult1↑.Counter − 1;
      MergeMultEq(Mult, Mult1)
    end;
    MergeMultiTerms(Mult↑.M, Frontier↑.Value↑.M);
    if Mult↑.Counter = 0 then
      U.ZeroCounterMultEq := NewListOfMultEq(Mult, U.ZeroCounterMultEq);
    Frontier := Frontier↑.Next
  end
end (*Compact*);
```

Figure 5

```
procedure MergeMultiTerms(var M: MultiTerm; M1: MultiTerm);
var Arg, Arg1: ListOfTempMultEq;
begin
  if M = Nil then M := M1
  else if not(M1 = Nil) then
  begin
    if not (M↑.Fsymb = M1↑.Fsymb) then fail('clash')
    else
    begin
      Arg := M↑.Args;
      Arg1 := M1↑.Args;
      while not(Arg = Nil) do
      begin
        Append(Arg↑.Value↑.S, Arg1↑.Value↑.S);
        MergeMultiTerms(Arg↑.Value↑.M, Arg1↑.Value↑.M);
        Arg := Arg↑.Next;
        Arg1 := Arg1↑.Next
      end
    end
  end
end (*MergeMultiTerms*);
```

Figure 6

of the form $f(x_1, \ldots, x_n)$ where $x_1, \ldots, x_n$ are variables. For instance,

$$\begin{aligned} \{x_1\} &= f(x_2, x_3, x_4); \\ \{x_2\} &= a; \\ \{x_3\} &= g(x_2); \\ \{x_4\} &= a; \\ \{x_5\} &= f(x_6, x_7, x_8); \\ \{x_6\} &= a; \\ \{x_7\} &= g(x_8); \\ \{x_8\} &= \varnothing. \end{aligned} \tag{4}$$

Furthermore, we have a set S of equations whose left- and right-hand sides are variables; for instance,

$$S\colon \{x_1 = x_5\}.$$

A step of both algorithms consists of choosing an equation from S, merging the two corresponding multiequations, and adding to S the new equations obtained as the outcome of the merging. For instance, after the first step we have

$$\begin{aligned} \{x_1, x_5\} &= f(x_2, x_3, x_4); \\ \{x_2\} &= a; \\ \{x_3\} &= g(x_2); \\ \{x_4\} &= a; \\ \{x_6\} &= a; \\ \{x_7\} &= g(x_8); \\ \{x_8\} &= \varnothing; \end{aligned} \tag{5}$$

$$S\colon \{x_2 = x_6, x_3 = x_7, x_4 = x_8\}.$$

The two algorithms differ in the way they select the equation from S. In Huet's algorithm S is a list; at every step, the first element of it is selected, and the new equations are added at the end of the list. The algorithm stops when S is empty, and up to this point it has not yet checked the absence of cycles. Thus, there is a last step which checks whether the final multiequations are partially ordered.

The source of the nonlinear behavior of this algorithm is the same as for our algorithm, that is, the access to multiequations after they have been merged. To avoid this, Paterson and Wegman choose to merge two multiequations only when their variables are no longer accessible. For instance, from (5) their algorithm selects $x_3 = x_7$ because x_2 and x_8 are still accessible from the third and sixth multiequation, respectively, getting

$$\begin{aligned} \{x_1, x_5\} &= f(x_2, x_3, x_4); \\ \{x_2\} &= a; \\ \{x_3, x_7\} &= g(x_2); \\ \{x_4\} &= a; \\ \{x_6\} &= a; \\ \{x_8\} &= \emptyset; \end{aligned}$$

$$S\colon \{x_2 = x_8, x_2 = x_6, x_4 = x_8\}.$$

To select the multiequations to be merged, this algorithm "climbs" the partial ordering among multiequations until it finds a multiequation which is "on top"; thus the detection of cycles is intrinsic in this algorithm.

Let us now see how our algorithm works with the above example. The initial system of multiequations is

$$\begin{aligned} U\colon \{&[0]\ \{x_1, x_5\} = f(\langle\{x_2, x_6\}, \emptyset\rangle, \langle\{x_3, x_7\}, \emptyset\rangle, \langle\{x_4, x_8\}, \emptyset\rangle), \\ &[2]\ \{x_2\} = a, \\ &[1]\ \{x_3\} = g(\langle\{x_2\}, \emptyset\rangle), \\ &[1]\ \{x_4\} = a, \\ &[1]\ \{x_6\} = a, \\ &[1]\ \{x_7\} = g(\langle\{x_8\}, \emptyset\rangle), \\ &[2]\ \{x_8\} = \emptyset\}; \end{aligned}$$

$$T\colon ().$$

The next step is

$$\begin{aligned} U\colon \{&[1]\ \{x_2, x_6\} = a, \\ &[0]\ \{x_3, x_7\} = g(\langle\{x_2, x_8\}, \emptyset\rangle), \\ &[1]\ \{x_4, x_8\} = a\}; \end{aligned}$$

$$T\colon (\{x_1, x_5\} = f(x_2, x_3, x_4)),$$

and so on.

In this algorithm the equations containing the pairs of variables to be unified are kept in the multiterms, and the mergings are delayed until the corresponding multiequation is eliminated.

An important difference between our algorithm and the others is that our algorithm may use terms of any depth. This fact entails a gain in efficiency, because it is certainly simpler to compute the common part and the frontier of deep terms than to merge multiequations step by step. Note, however, that this feature might also be added to the other algorithms. For instance, by adding the capability of dealing with deep terms to Paterson and Wegman's algorithm, we essentially obtain a linear algorithm which was independently discovered by the authors [13].

In order to compare the essential features of the three algorithms, we notice that they can stop either with success or with failure for the detection of a cycle or with failure for the detection of a clash. Let P_m, P_c, and P_t be the probabilities of stopping with one of these three events, respectively. We consider three extreme cases:

(1) $P_m \gg P_c$, P_t (*very high probability of stopping with success*). Paterson and Wegman's algorithm is asymptotically the best, because it has a linear complexity whereas the other two algorithms have a comparable nonlinear complexity.

However, in a typical application, such as, for example, a theorem prover, the unification algorithm is not used for unifying very large terms, but instead it is used a great number of times for unifying rather small terms each time. In this case we cannot exploit the asymptotically growing difference between linear and nonlinear algorithms, and the computing times of the three algorithms will be comparable, depending on the efficiency of the implementation.

An experimental comparison of these algorithms, together with others, was carried out by Trum and Winterstein [21]. The algorithms were implemented in the same language, PASCAL, with similar data structures, and tried on five different classes of unifying test data. Our algorithm had the lowest running time for all test data. In fact, our algorithm is more efficient than Huet's because it does not need a final acyclicity test, and it is more efficient than Paterson and Wegman's because it needs simpler data structures.

(2) $P_c \gg P_t \gg P_m$ (*very high probability of detecting a cycle*). Paterson and Wegman's algorithm is the best because it starts merging two multiequations only when it is sure that there are no cycles above them. Our algorithm is also good because cycle detection is embedded in it. In contrast, Huet's algorithm must complete all mergings before being able to detect a cycle, and thus it has a very poor performance.

(3) $P_t \gg P_c \gg P_m$ (*very high probability of detecting a clash*). Huet's algorithm is the best because, if it stops with a clash, it has not paid any overhead for cycle detection. Our algorithm is better than Paterson and Wegman's because clashes are detected during multiequation merging and because our algorithm may merge some multiequations earlier, like $\{x_2, x_6\}$ and $\{x_4, x_8\}$ in the above example. On the other hand, mergings which are delayed by our algorithm, by putting them in multiterms, cannot be done earlier by the other algorithm because they refer to multiequations which are still accessible. The difference in the performance of the two algorithms may become quite large if terms of any depth are allowed.

8. CONCLUSION

A new unification algorithm has been presented. Its performance has been compared with that of other well-known algorithms in three extreme cases: high probability of stopping with success, high probability of detecting a cycle, and high probability of detecting a clash. Our algorithm was shown to have a good performance in all the cases, and thus presumably in all the intermediate cases, whereas the other algorithms had a poor performance in some cases.

Most applications of the unification algorithm, such as, for instance, a resolution theorem prover or the interpreter of an equation language, require repeated use of the unification algorithm. The algorithm described in this paper can be very efficient even in this case, as the authors have shown in [12]. There they have proposed to merge this unification algorithm with Boyer and Moore's technique for storing shared structures in resolution-based theorem provers [3] and have shown that, by using the unification algorithm of this paper instead of the standard one, an exponential saving of computing time can be achieved. Furthermore, the time spent for initializations, which might be heavy for a single execution of the unification algorithm, is there reduced through a close integration of the unification algorithm into the whole theorem prover.

REFERENCES

1. AHO, A.V., HOPCROFT, J.E., AND ULLMAN, J.D. *The Design and Analysis of Computer Algorithms.* Addison-Wesley, Reading, Mass., 1974.
2. BAXTER, L.D. A practically linear unification algorithm. Res. Rep. CS-76-13, Dep. of Applied Analysis and Computer Science, Univ. of Waterloo, Waterloo, Ontario, Canada.
3. BOYER, R.S., AND MOORE, J.S. The sharing of structure in theorem-proving programs. In *Machine Intelligence*, vol. 7, B. Meltzer and D. Michie (Eds.). Edinburgh Univ. Press, Edinburgh, Scotland, 1972, pp. 101–116.
4. BURSTALL, R.M., AND DARLINGTON, J. A transformation system for developing recursive programs. *J. ACM 24*, 1 (Jan. 1977), 44–67.
5. CHANG, C.L., AND LEE, R.C. *Symbolic Logic and Mechanical Theorem Proving.* Academic Press, New York, 1973.
6. HEWITT, C. Description and Theoretical Analysis (Using Schemata) of PLANNER: A Language for Proving Theorems and Manipulating Models in a Robot. Ph.D. dissertation, Dep. of Mathematics, Massachusetts Institute of Technology, Cambridge, Mass., 1972.
7. HUET, G. Résolution d'équations dans les langages d'ordre 1, 2, . . . , ω. Thèse d'état, Spécialité Mathèmatiques, Université Paris VII, 1976.
8. HUET, G.P. A unification algorithm for typed λ-calculus. *Theor. Comput. Sci. 1*, 1 (June 1975), 27–57.
9. KNUTH, D.E., AND BENDIX, P.B. Simple word problems in universal algebras. In *Computational Problems in Abstract Algebra*, J. Leech (Ed.). Pergamon Press, Elmsford, N.Y., 1970, pp. 263–297.
10. KOWALSKI, R. Predicate logic as a programming language. In *Information Processing 74*, Elsevier North-Holland, New York, 1974, pp. 569–574.
11. LEVI, G., AND SIROVICH, F. Proving program properties, symbolic evaluation and logical procedural semantics. In *Lecture Notes in Computer Science*, vol. 32: *Mathematical Foundations of Computer Science 1975.* Springer-Verlag, New York, 1975, pp. 294–301.
12. MARTELLI, A., AND MONTANARI, U. Theorem proving with structure sharing and efficient unification. Internal Rep. S-77-7, Ist. di Scienze della Informazione, University of Pisa, Pisa, Italy; also in Proceedings of the 5th International Joint Conference on Artificial Intelligence, Boston, 1977, p. 543.
13. MARTELLI, A., AND MONTANARI, U. Unification in linear time and space: A structured presen-

tation. Internal Rep. B76-16, Ist. di Elaborazione delle Informazione, Consiglio Nazionale delle Ricerche, Pisa, Italy, July 1976.
14. MILNER, R. A theory of type polymorphism in programming. *J. Comput. Syst. Sci. 17*, 3 (Dec. 1978), 348–375.
15. PATERSON, M.S., AND WEGMAN, M.N. Linear unification. *J. Comput. Syst. Sci. 16*, 2 (April 1978), 158–167.
16. ROBINSON, J.A. Fast unification. In Theorem Proving Workshop, Oberwolfach, W. Germany, Jan. 1976.
17. ROBINSON, J.A. Computational logic: The unification computation. In *Machine Intelligence*, vol. 6, B. Meltzer and D. Michie (Eds.). Edinburgh Univ. Press, Edinburgh, Scotland, 1971, pp. 63–72.
18. ROBINSON, J.A. A machine-oriented logic based on the resolution principle. *J. ACM 12*, 1 (Jan. 1965), 23–41.
19. SHORTLIFFE, E.H. *Computer-Based Medical Consultation: MYCIN*. Elsevier North-Holland, New York, 1976.
20. STICKEL, M.E. A complete unification algorithm for associative-commutative functions. In Proceedings of the 4th International Joint Conference on Artificial Intelligence, Tbilisi, U.S.S.R., 1975, pp. 71–76.
21. TRUM, P., AND WINTERSTEIN, G. Description, implementation, and practical comparison of unification algorithms. Internal Rep. 6/78, Fachbereich Informatik, Univ. of Kaiserlautern, W. Germany.
22. VENTURINI ZILLI, M. Complexity of the unification algorithm for first-order expressions. *Calcolo 12*, 4 (Oct.–Dec. 1975), 361–372.
23. VON HENKE, F.W., AND LUCKHAM, D.C. Automatic program verification III: A methodology for verifying programs. Stanford Artificial Intelligence Laboratory Memo AIM-256, Stanford Univ., Stanford, Calif., Dec. 1974.
24. WALDINGER, R.J., AND LEVITT, K.N. Reasoning about programs. *Artif. Intell. 5*, 3 (Fall 1974), 235–316.
25. WARREN, D.H.D., PEREIRA, L.M., AND PEREIRA, F. PROLOG—The language and its implementation compared with LISP. In Proceedings of Symposium on Artificial Intelligence and Programming Languages, Univ. of Rochester, Rochester, N.Y., Aug. 15–17, 1977. Appeared as joint issue: *SIGPLAN Notices* (ACM) *12*, 8 (Aug. 1977), and *SIGART Newsl.* 64 (Aug. 1977), 109–115.

Received September 1979; revised July 1980 and September 1981; accepted October 1981