Appeared in JSC 1989, vol 8

Adventures in Associative-Commutative Unification

PATRICK LINCOLN and JIM CHRISTIAN *

Department of Computer Science, Stanford University, Stanford, CA, 94305, USA

lincoln@polya.stanford.edu

and

Department of Computer Science, University of Texas at Austin, Austin, TX, 78712, USA

jimc@rascal.ics.utexas.edu

(Received March 24, 1993)

Abstract

We have discovered an efficient algorithm for matching and unification in associative-commutative (AC) equational theories. In most cases of AC unification our method obviates the need for solving diophantine equations, and thus avoids one of the bottlenecks of other associative-commutative unification techniques. The algorithm efficiently utilizes powerful constraints to eliminate much of the search involved in generating valid substitutions. Moreover, it is able to generate solutions lazily, enabling its use in an SLD-resolution-based environment like Prolog. We have found the method to run much faster and use less space than other associative-commutative unification procedures on many commonly encountered AC problems.

1 Introduction

Associative-commutative (AC) equational theories surface in a number of computer science applications, including term rewriting, automatic theorem proving, software verification, and database retrieval. As a simple example, consider trying to find a substitution for variables in two sets – say $\{a, x, c\}$ and $\{c, y, b\}$ – which makes them identical up to reordering of elements. We can represent each set by using the constructor *cons*. This gives us the two terms $cons(a, cons(x, c))$ and $cons(c, (cons(y, b))$. Now, if we declare *cons* to be associative and commutative, then two axioms can be applied to determine equality of sets:

$$cons(cons(x, y), z) = cons(x, cons(y, z))$$

*Order irrelevant. Work completed at MCC, 3500 W. Balcones Cntr. Dr, Austin, TX, 78759.

$$cons(x, y) = cons(y, x)$$

From this we can decide that substituting b for x and a for y makes both sets equal modulo the AC axioms.

Unfortunately, the introduction of equality axioms into a system like a theorem prover brings with it unreasonably large search spaces. The commutativity axiom above, for instance, can be applied in an infinite sequence if it can be applied at all. Termination of systems incorporating equality axioms can often be had only by compromising completeness.

Happily, many of the problems caused by an equational theory can be circumvented if there exists a complete unification procedure for the theory. By relying on unification instead of search, troublesome axioms can be removed from a system.

A complete unification algorithm for AC theories was developed several years ago by Mark Stickel [18]. Independently, Livesey and Siekmann published a similar algorithm for AC and ACI unification. Their procedures center around generating solutions to a linear diophantine equation, each coefficient of which represents the multiplicity of some subterm in one of the unificands. There are two nagging properties of this method. First, it requires generating a basis for the solution space of the diophantine equation. Second, there can be a large amount of search involved in actually generating solutions once a basis is discovered.

We have found an algorithm for dealing with associative-commutative theories without resorting to the solution of diophantine equations. By weakening the variable abstraction introduced by Stickel, most cases of AC can be solved by working only with equations in which all coefficients are unity. The basis of solutions of such an equation possesses a highly regular structure; this allows us to optimize the representation of the problem and avoid spending time finding a basis. We are able instead to begin generating unifiers almost immediately. In addition, our representation allows the incorporation of several simple but powerful constraints in a way that is much more natural and efficient than previous methods have allowed.

Our algorithm can solve AC matching problems (so-called "one-way unification"), and most cases of AC unification very efficiently – in most cases, several times faster than Stickel's algorithm. However, if repeated variables occur in one unificand, our algorithm may return redundant unifiers. If repeated variables occur in both unificands, our algorithm may not terminate. In these easily detected cases, it suffices to dispatch to some complete algorithm, like Stickel's; the overhead in making the decision to dispatch is negligible. Fortunately, these cases occur in only a few percent of many applications of AC unification like Knuth-Bendix completion [12]. In some applications like database query compilation, these cases never occur. Thus our procedure can achieve a significant improvement in average execution speed of AC unification. Furthermore, our procedure requires nominal space overhead for generating solutions, and is amenable to the lazy generation of solutions required in an SLD-resolution environment like that of Prolog.

2 What is AC Unification?

Before delving into the intricacies of our unification procedure, it is reasonable at this point to clear any misunderstanding or confusion the reader might have with respect to AC unification and matching. In the introduction, we presented a simple example; now

we shall try to evoke a more complete appreciation for the problem. We begin with some requisite definitions.

We suppose the existence of three sets: *constants*, *variables*, and *function symbols*. A *term* is inductively defined:

- A constant is a term.
- A variable is a term.
- Let f be a function symbol; then $f(t_1, \ldots, t_n)$ is a term, where $t_1, \ldots, t_n$ are terms.

A *substitution* is a function from variables to terms. We shall represent a substitution as a set of *assignments*, where an assignment $v \leftarrow t$ maps variable v to term t. In order to justify applying substitutions to arbitrary terms, homomorphic extensions of substitutions are used; a substitution maps all constants and function symbols to themselves.

A term u is an *instantiation* of a term v if there is a substitution θ such that $u = v\theta$. Two terms *unify modulo a theory* T if there is a substitution which, when applied to both terms, yields two terms which can be proved equal in the theory. Ordinary unification is the special case wherein T is empty.

An *associative-commutative* theory is one comprising the following axioms:

- $f(f(x, y), z) = f(x, f(y, z))$ (Associativity)
- $f(x, y) = f(y, x)$ (Commutativity)

We will sometimes write $t_1 =_{AC} t_2$ to indicate equality of t_1 and t_2 modulo associativity and commutativity.

AC unification is unification modulo the theory of associativity and commutativity. *AC matching* is a restricted form of unification wherein assignments are allowed to variables in only one of the unificands. That is, the resulting substitution must not assign terms to variables from one of the unificands.

Obviously, AC-ness implies the ability to permute subterms rather arbitrarily. As an illustration, the terms $f(a, f(b, c))$, $f(f(a, c), b)$, and $f(b, f(a, c))$ are equal by the AC axioms. With the introduction of variables, terms may unify in ways which might at first seem contrived. The terms $f(x, y)$ and $f(a, z)$, for instance, share four unifiers. One of them is the substitution yielded by "ordinary" unification, namely, $\{x \leftarrow a, y \leftarrow z\}$. By applying the commutativity axiom, we find the unifier $\{y \leftarrow a, x \leftarrow z\}$. This is not surprising. But notice that application of the associativity axiom can cause variables to be distributed among subterms, as in the substitution $\{x \leftarrow f(a, v), z \leftarrow f(v, y)\}$, where v is a new variable. Here, a most-general common instance is $f(f(a, v), y))$; notice how components of z can be found both in the subterm $f(a, v)$ and in the subterm y. The last unifier is $\{y \leftarrow f(a, v), z \leftarrow f(v, x)\}$.

In general, the number of unifiers for any two terms can be exponential in the size of the terms; AC unification is NP-complete [1]. When there are only a few variables in

the unificands, however, the number of unifiers can be markedly reduced. In [18] Stickel shows that the two terms $f(x, f(x, f(y, z)))$ and $f(u, f(u, v))$ yield 69 unifiers, while their instantiations $f(x, f(x, f(y, a)))$ and $f(b, f(b, v))$ produce only four.

3 History Of AC Unification

Mark Stickel was the first to develop a complete, terminating algorithm for AC unification; the algorithm was initially presented in 1975 [17]. Livesey and Siekmann published a similar algorithm in 1976 [13]. Most AC unification procedures in use today are essentially modifications of that of Stickel or of Livesey and Siekmann, but a few novel approaches have been proposed. Within the loose framework of Stickel's method there are two hard problems: generating a basis of solutions to linear homogeneous diophantine equations, and searching through all combinations of this basis for a solution to the given AC unification problem.

Since Gordan's study of diophantine equations in 1873 [5], only in the last few years has there been any significant progress made regarding the generation of their bases. Fortenbacher, Huet, and Lankford have separately proposed a number of refinements to Gordan's basic method [4, 7, 12]. Recently, Zhang has discovered a class of diophantine equations which can be quickly solved [20]. However, no published algorithm has proven to be superior in all cases.

The extraction of solutions to AC problems given a basis of solutions to the diophantine equation is also an area of concern. In the past few years Fortenbacher [4] has proposed a method of reducing the search space by eliminating certain basis elements. Claude Kirchner has recently developed an AC unification algorithm within the framework of the Martelli-Montanari unification procedure, but, like Stickel's, his method requires solving diophantine equations [11, 14]. Also, Hullot invented an algorithm for AC unification which involves ordered partitions of multisets [9]. While his algorithm is faster than Stickel's, it does not seem to offer nearly the dramatic speed increases we have obtained with our procedure. We have not implemented Hullot's algorithm, but base our judgement on timing comparisons listed in his paper. In Germany, Büttner has developed a parallel algorithm for AC unification [2]. The method involves linear algebraic operations in multi-dimensional vector spaces, but he fails to provide the details necessary for a realistic comparison. Recently, Kapur [10] has developed an algorithm based on Stickel's method that uses Zhang's equation solving technique. A paper by Herold and Siekmann [6] pursues some issues of unification further, and the surveys by Huet and Oppen [8] and by Siekmann [16] summarize results in related areas.

4 Stickel's Method

In a twisted sort of way, our procedure derived from Stickel's; so we will briefly review his algorithm before presenting our own. In case we fail to do justice to the algorithm, the unfamiliar reader is encouraged to look up Stickel's paper, which is straightforward

and lucid [18].

In order to unify two terms modulo the AC axioms, three preparation steps are necessary.

First, they are both put through a "flattening" operation which removes nested AC function symbols, thus transforming unification in the free algebra into unification in an Abelian Semigroup. This operation can be viewed more precisely as term reduction by root application of the rewrite rule $f(t_1, \ldots, f(s_1, \ldots, s_m), \ldots, t_n) \longrightarrow f(t_1, \ldots, s_1, \ldots, s_m, \ldots, t_n)$. Hence, the term $f(f(a,a),a,f(g(u),y,x))$ will be flattened to $f(a,a,a,g(u),y,x)$, while $f(a,f(b,g(c)),f(y,y),z)$ will be changed to $f(a,b,g(c),y,y,z)$. The validity of the flattening operation is guaranteed by the associativity axiom, which implies that nesting of AC function symbols is largely irrelevant.

After flattening terms, the next step is to remove subterms which occur in both unificands. For instance, after deleting duplicate subterms from $f(a,a,a,g(u),y,x)$ and $f(a,b,g(c),y,y,z)$ we obtain the terms $f(a,a,g(u),x)$ and $f(b,g(c),y,z)$, specifically by removing one occurrence of a and one occurrence of y from each.

The final preparation step is to generalize each term, replacing each distinct argument with a variable. So $f(a,a,g(u),x)$ will be generalized to $f(x_1,x_1,x_2,x_3)$, and $f(b,g(c),y,z)$ is generalized to $f(y_1,y_2,y_3,y_4)$.

Now we get to the crux of the procedure. The goal is to construct a linear diophantine equation – that is, a linear equation with non-negative integer coefficients – where a coefficient corresponds to the multiplicity of one of the variables in the generalized terms. More accurately, given generalized terms t_1 with variables $x_1, \ldots, x_m$ and t_2 with variables $y_1, \ldots, y_n$, we set up an equation $c_1x_1 + \ldots + c_mx_m = d_1y_1 + \ldots + d_ny_n$. Each c_i is an integer equal to the number of times x_i occurs in t_1, while d_j represents the multiplicity of y_j in t_2. So the equation associated with the generalized terms $f(x_1,x_1,x_2,x_3)$ and $f(y_1,y_2,y_3,y_4)$ is $2x_1 + x_2 + x_3 = y_1 + y_2 + y_3 + y_4$.

The motivation for constructing a diophantine equation is not hard to grasp. For each assignment made by the unification algorithm of a term to a variable, the term must be present in both unificands an equal number of times; thus, unifying substitutions can be represented as solutions to the corresponding diophantine equation. The next step toward producing solutions is to generate a basis for the diophantine equation, which enables the systematic construction of solutions. We will not be concerned here with the procedure for finding a basis, except to say that we prefer Lankford's algorithm for most common equations (that is, those with small coefficients) [12]. After generating the basis, a new variable z_i is associated with each vector v_i in the basis. So, from the equation $2x_1 + x_2 + x_3 = y_1 + y_2 + y_3 + y_4$, we produce the variable-labeled basis in Table 1.

Now, unifying substitutions are generated by picking certain subsets of the basis vectors and summing them. If we choose basis vectors 2, 4, 7, and 9 from Table 1, we remove all other basis vectors, and consider only the chosen vectors, as in Table 2.

So for each x and y, we construct a substitution by reading down that variable's column. For instance, looking at x_1, we see that its substitution involves only z_9, $\{x_1 \leftarrow f(z_9)\}$. If a term's substitution only contains one element, as does x_1, we simply write

	a	$g(u)$	x	b	$g(c)$	y	z	
Number	$2x_1$	x_2	x_3	y_1	y_2	y_3	y_4	Label
1	0	0	1	0	0	0	1	z_1
2	0	0	1	0	0	1	0	z_2
3	0	0	1	0	1	0	0	z_3
4	0	0	1	1	0	0	0	z_4
5	0	1	0	0	0	0	1	z_5
6	0	1	0	0	0	1	0	z_6
7	0	1	0	0	1	0	0	z_7
8	0	1	0	1	0	0	0	z_8
9	1	0	0	0	0	0	2	z_9
10	1	0	0	0	0	1	1	z_{10}
11	1	0	0	0	0	2	0	z_{11}
12	1	0	0	0	1	0	1	z_{12}
13	1	0	0	0	1	1	0	z_{13}
14	1	0	0	0	2	0	0	z_{14}
15	1	0	0	1	0	0	1	z_{15}
16	1	0	0	1	0	1	0	z_{16}
17	1	0	0	1	1	0	0	z_{17}
18	1	0	0	2	0	0	0	z_{18}

Table 1: Basis for $2x_1 + x_2 + x_3 = y_1 + y_2 + y_3 + y_4$

	a	$g(u)$	x	b	$g(c)$	y	z	
Number	$2x_1$	x_2	x_3	y_1	y_2	y_3	y_4	Label
2	0	0	1	0	0	1	0	z_2
4	0	0	1	1	0	0	0	z_4
7	0	1	0	0	1	0	0	z_7
9	1	0	0	0	0	0	2	z_9

Table 2: Selected Basis Vectors for $2x_1 + x_2 + x_3 = y_1 + y_2 + y_3 + y_4$

$\{x_1 \leftarrow z_9\}$. More exactly, for each coefficient c_j in a basis vector v_i, the corresponding variable z_i is assigned c_j times to the corresponding variable in the generalized terms, for each c_j in the chosen solution. To continue the example, the full substitution is $\{x_1 \leftarrow z_9, x_2 \leftarrow z_7, x_3 \leftarrow f(z_2, z_4), y_1 \leftarrow z_4, y_2 \leftarrow z_7, y_3 \leftarrow z_2, y_4 \leftarrow f(z_9, z_9)\}$. Note that since there is a 2 in y_4's position of vector v_9, z_9 we assign z_9 twice to y_4, $\{y_4 \leftarrow f(z_9, z_9)\}$.

The final step of the algorithm is to unify variables in the generalized terms with their counterparts in the original terms. So, supposing we started with the terms $f(a, a, g(u), x)$ and $f(b, g(c), y, z)$, the unifying substitution resulting from choosing basis vectors 2, 4, 7, and 9 would be (after simplification) $\{x \leftarrow f(b, y), g(u) \leftarrow g(c), z \leftarrow f(a, a)\}$. Note that the substitution $g(u) \leftarrow g(c)$ would require recursive unification of terms, which would produce the substitution $u \leftarrow c$. This happens to be a valid substitution, which would be returned as one of the solutions to the original AC unification problem.

Stickel briefly mentioned some obvious optimizations to the above algorithm in his original paper, and Fortenbacher [4] has described some of these in detail, with formal justification. Below we give a quick review of some of the more important constraints used in actual implementations.

Notice that, given n basis vectors, there are potentially 2^n solutions; however, there are ordinarily a number of constraints on the solution space which reduce this number considerably.

First, any basis vector which would force the unification of distinct constants can be eliminated from the basis. In Table 1, the basis vectors 15, 16, 17, and 18 can never be used in valid substitutions, since they each will require some unification of one constant a with another, b. In this case eliminating these vectors reduces the search space by a factor of 16. Also, vectors 8, 12, 13, and 14 would force unification of a constant with a functional term headed by g, which must fail. Removing these and the above vectors reduces the search space from over 262,144 to 1,024. Any basis vectors which has non unit coefficients for variables representing constants or functional terms can be eliminated since the use of that basis element would force the unification of, say, a constant with something of the form $f(z_1, z_2)$. Basis vectors 14 and 18 from the above example could thus be removed, if they were not already invalidated by other constraints. Finally, any terms headed by distinct ununifiable function symbols can never be unified, and so any basis which would require this kind of unification can be eliminated from the set of basis elements.

During the construction of unifiers from subsets of basis elements, similar constraints apply. Consider that each variable in the generalized term must be assigned to something. Thus combinations of vectors which sum to zero in any column will not yield valid unifiers. Also consider that any combination of basis vectors which assigns more than one z_i to a variable which is associated with a constant in the original term is invalid, since a constant can not be unified with a term of the form, say, $f(z_i, z_j)$. Furthermore, as soon as we notice that we have chosen a combination of vectors which would cause a clash of any type, we can abort that combination and try another, instead of waiting until an entire substitution is complete before checking its consistency. Also, Stickel implies that all the unifications be completed as the last step in his procedure. It turns out that

much of this unification work can be performed early, resulting in less total unifications, although this is of marginal utility if only one unifier at a time is desired, due to the space overhead of retaining early unification results. Fortenbacher also tries to take advantage of early failures in a similar manner. In any realistic implementation of AC unification, these types of constraints must be enforced.

5 Our Method

As previously mentioned, there are two basic difficulties with algorithms based on Stickel's approach. First, generation of a basis for a diophantine equation is an expensive operation. Second, given a basis, the search which must be performed to produce solutions can be very expensive. It is thus necessary to enforce many non-trivial constraints [4, 19]. Stickel, Kapur, and others have implemented quite impressive constraints on the generation of solutions which tame this search problem. However, these efficient implementations "require a less pure conceptual abstraction" [19] than other techniques might. For example, efficient implementation of Stickel's algorithm requires discovery of basis elements of Diophantine Equations "with the additional constraints that certain variables cannot have a value greater than one and that, for certain pairs of variables, only one of them can be nonzero" [19].

We believe that our novel representation facilitates a much cleaner conceptual abstraction of constraints on AC unifiers than do previous methods, and thus enables our algorithm to exploit powerful constraints in a very natural and efficient way. The main contribution of our representation is that it allows us to avoid solving Diophantine equations in most cases. In particular, whenever there are no repeated variables in either term, no diophantine equations need be solved. In other cases it is often, though not always, necessary to solve a diophantine equation.

5.1 Slaying the Diophantine Dragon

The observation that certain classes of Diophantine equations have very simple solutions is not new. Zhang's diophantine equation solving technique is based on a similar notion; if there are several unit coefficients in a diophantine equation, a complete basis of solutions to that equation can be generated from the basis of a much simpler equation [20]. We have gone further in our specialization, restricting our attention to equations which have only unit coefficients. It turns out that we are able to force many AC unification problems which do not appear to generate equations with unit coefficients into a form which can be solved using our technique. In fact, it is only those AC unification problems with repeated variables which we cannot completely solve in general.

The preparation phase of our algorithm is very similar to previous approaches. First, both terms are put through the flattening operation described above. Then duplicate subterms are removed pairwise from both unificands, and subterms are sorted so that atomic constants are grouped together, followed by function terms, followed by variables. For instance, $f(a, g(u), a, y, a, x)$ would be sorted to produce $f(a, a, a, g(u), y, x)$.

Number	x_1	x_2	x_3	x_4	y_1	y_2	y_3	y_4	Label
1	0	0	0	1	0	0	0	1	z_1
2	0	0	0	1	0	0	1	0	z_2
3	0	0	0	1	0	1	0	0	z_3
4	0	0	0	1	1	0	0	0	z_4
5	0	0	1	0	0	0	0	1	z_5
6	0	0	1	0	0	0	1	0	z_6
7	0	0	1	0	0	1	0	0	z_7
8	0	0	1	0	1	0	0	0	z_8
9	0	1	0	0	0	0	0	1	z_9
10	0	1	0	0	0	0	1	0	z_{10}
11	0	1	0	0	0	1	0	0	z_{11}
12	0	1	0	0	1	0	0	0	z_{12}
13	1	0	0	0	0	0	0	1	z_{13}
14	1	0	0	0	0	0	1	0	z_{14}
15	1	0	0	0	0	1	0	0	z_{15}
16	1	0	0	0	1	0	0	0	z_{16}

Table 3: Basis for $x_1 + x_2 + x_3 + x_4 = y_1 + y_2 + y_3 + y_4$

Now, our generalization step differs from others, in that we assign a distinct variable for *each* argument. Thus, while Stickel's algorithm would convert the $f(a, a, g, x)$ to $f(X_1, X_1, X_2, X_3)$, ours will produce $f(X_1, X_2, X_3, X_4)$. Effectively, we convert the problem of solving the unification problem $f(x_1, \ldots, x_m) = f(y_1, \ldots, y_n)$ into the equivalent conjunction of problems $f(X_1, \ldots, X_m) = f(Y_1, \ldots, Y_n) \wedge x_1 = X_1 \wedge \ldots \wedge x_m = X_m \wedge y_1 = Y_1 \wedge \ldots \wedge y_n = Y_n$, where the X_i and Y_j are distinct variables, and the x_i and and y_j are subterms of the original equation.

Notice that the diophantine equation corresponding to any pair of such generalized terms will have only unit coefficients. Such an equation has a nice property, as stated in theorem 1. To illustrate, the equation $x_1 + x_2 + x_3 + x_4 = y_1 + y_2 + y_3 + y_4$ has the solution basis shown in Table 3.

Theorem 1 *Given a diophantine equation of the form $x_1 + \ldots + x_m = y_1 + \ldots + y_n$, the minimal solution basis is that set of solutions such that, for each solution, exactly one x_i has value one, exactly one y_j has value one, and all other variables have value zero. Also, the number of basis solutions is nm.*

Proof: See Section 6. □

Knowing that the basis has such a nice, regular structure, we need not explicitly generate it; for, given only the respective arities of the generalized unificands, we can immediately construct a two dimensional matrix, where each column is labeled with an x_i, and each row is labeled with one of the y_j. Each entry i, j in the matrix is a boolean value, that corresponds to a new variable, $z_{i,j}$, which represents the solution vector which assigns a one to x_j and y_i. Thus every true boolean value $z_{i,j}$ in a solution matrix corresponds to one basis element of the solution of the diophantine equation. Any assignment of true and false to all the elements of a matrix represents a potential solution

	x_1	x_2	x_3	x_4
y_1	$z_{1,1}$	$z_{1,2}$	$z_{1,3}$	$z_{1,4}$
y_2	$z_{2,1}$	$z_{2,2}$	$z_{2,3}$	$z_{2,4}$
y_3	$z_{3,1}$	$z_{3,2}$	$z_{3,3}$	$z_{3,4}$
y_4	$z_{4,1}$	$z_{4,2}$	$z_{4,3}$	$z_{4,4}$

Table 4: Matrix representation of basis for $x_1 + x_2 + x_3 + x_4 = y_1 + y_2 + y_3 + y_4$

		a	a	$g(u)$	x
		x_1	x_2	x_3	x_4
b	y_1	$z_{1,1}$	$z_{1,2}$	$z_{1,3}$	$z_{1,4}$
$g(c)$	y_2	$z_{2,1}$	$z_{2,2}$	$z_{2,3}$	$z_{2,4}$
y	y_3	$z_{3,1}$	$z_{3,2}$	$z_{3,3}$	$z_{3,4}$
z	y_4	$z_{4,1}$	$z_{4,2}$	$z_{4,3}$	$z_{4,4}$

	C	T	V
C	0	0	$\leftrightarrow$
T	0	$\updownarrow$	$\leftrightarrow$
V	$\updownarrow$	$\updownarrow$	any

Table 5: Matrix for a simple problem and some constraints

to the AC unification problem in the same way that any subset of the basis elements of the diophantine equation represents a potential solution to the same AC problem.

For instance, suppose we are given the (already flattened) unificands $f(a, a, g(u), x)$ and $f(b, g(c), y, z)$. Substituting new variables for each argument, we obtain $f(x_1, x_2, x_3, x_4)$ and $f(y_1, y_2, y_3, y_4)$. The associated solution matrix is displayed in Table 5.

In our implementation, we do not create the entire n by m matrix; rather, we will utilize a more convenient and compact data structure. But for now, let us pretend that the matrix is represented as a simple 2-dimensional array. As we will demonstrate below, the matrix representation is inherently amenable to constraining the search for unifiers.

5.2 Constraining Search

Remember that unificands are sorted in the preparation step of our algorithm. Hence, a given solution matrix comprises nine regions, illustrated in Table 5. In the table, C, T, and V stand, respectively, for atomic constants, functional terms, and variables. An entry in the lower left region of the matrix, for instance, corresponds to an assignment in the (unprepared) unificands of a constant in one and a variable in the other.

As Table 5 indicates, there are several constraints on the distribution of ones and zeros within a solution matrix. First, notice that there must be at least one non-zero entry in each row and column of a solution matrix, so that all variables in the generalized terms receive an assignment. The upper left corner involves assignments to incompatible constants (since we have removed duplicate arguments from the unificands, no constants from one term can possibly unify with any constant from the other term). This part of any solution matrix, then, must consist only of zeros. Similarly, the C/T and T/C regions of a solution matrix must contain all zeros. The C/V region is constrained to

		a	a	$g(u)$	x
		x_1	x_2	x_3	x_4
b	y_1	0	0	0	1
$g(c)$	y_2	0	0	1	0
y	y_3	0	0	0	1
z	y_4	1	1	0	0

Unifying substitution:

$$x \leftarrow f(b, y)$$
$$z \leftarrow f(a, a)$$
$$u \leftarrow c$$

Table 6: A solution to the matrix

have exactly a single one in each column, since any additional ones would cause the attempted unification of a functional term, say $f(z_{1,1}, z_{1,2})$, with a constant. Similarly, any T row or T column must contain exactly one one. Finally, the V/V region of a matrix can have any combination of ones and zeros which does not leave a whole row or column filled only with zeros.

5.3 Generating Solutions

Once a unification problem has been cast into our matrix representation, it is not a difficult matter to find unifying substitutions. The approach is to determine a valid configuration of ones and zeros within the matrix, perform the indicated assignments to the variables in the generalized terms, and finally unify the arguments of the original unificands with their variable generalizations.

Consider the matrix in Table 5. We know that location $(1, 1)$ must be zero, since it falls within the C/C region of the matrix. Likewise, $(1, 2)$, $(1, 3)$, $(2, 1)$, and $(2, 2)$ must always be zero. In fact, the only possible position for the required one in the y_1 column is at $(1, 4)$. Filling out the rest of the matrix, we arrive at the solution shown in Table 6 after assigning the nonzero $z_{i,j}$'s to the x and y variables, and then unifying the variables with the original unificand arguments, we obtain the substitution shown beside Table 6 $g(u) \leftarrow g(c)$ produces $\{u \leftarrow c\}$. In general, such recursive unifications can involve full AC unification.

5.4 Lazy generation of solutions

In other AC unification algorithms, the overhead in storing and restoring state in the midst of discovering AC unifiers is prohibitive. Thus most other algorithms generate all solutions to an AC unification problem at once. But in some contexts it is desirable to return only one solution, and to delay discovery of additional unifiers until they are demanded. Using our technique, as each unifier is generated, the matrix configuration can be stored, and search suspended. If an additional unifier is demanded, it can be generated directly from the stored matrix representation.

Since we can implement certain parts of the matrix as a binary representation of a number, simple counting in binary is all that is required for complete enumeration. Thus assuming that we are in an SLD-resolution framework, the information necessary

to generate the next solution can be represented as a sequence of binary numbers. Upon backtracking, the binary representation is incremented, and checked for a few simple constraints. If the constraints are satisfied, the binary numbers are stored and the unifier is constructed and passed along.

The matrix can be represented very compactly, since most regions within it are sparse. With each column or row, associate a counter; the value of the counter represents which entry within a row or column contains a one. The variable-variable region cannot be represented so compactly, however, since nearly any assignment of ones and zeros is possible. Additional information can be maintained along with the matrix data structure, such as a tally of variable assignments up to the current point of the assignment procedure, and auxiliary data structures to keep track of repeated arguments.

An important advantage of this approach to generating solutions is that it allows early detection of failure; as soon as an inconsistent state is discovered, the procedure can abort the state, effectively pruning entire branches of the search tree.

5.5 Repeated terms

Until now, we have assumed that all arguments within a unificand are distinct. However, this is not necessarily the case for AC unification. In practice, repeated terms occur infrequently; Lankford, for instance, has found that more than 90 percent of the unification problems encountered in some completion applications involve unificands with distinct arguments. Nevertheless, the ability to handle repeated arguments is certainly desirable.

Our algorithm can easily be adapted to handle repetitions in constants and functional terms in either or both unificands, but repeated variables are more difficult to manage. If they occur in a single unificand, our algorithm is complete and terminating, but may return redundant unifiers. Although the set of unifiers returned by Stickel's algorithm is similarly not guaranteed to be minimal, in many cases our algorithm generates many more redundant unifiers than would Stickel's. If a minimal set of unifiers is required, it suffices to simply remove elements of the non-minimal set which are subsumed by other unifiers in the set.

If repeated variables occur in both unificands, our algorithm might generate subproblems at least as hard as the original, and thus may not terminate. Stickel's algorithm can be employed whenever repeated variables are detected; the overhead involved in making this decision is negligible. Thus in the worst cases we do simple argument checking, and dispatch to Stickel's algorithm. We have several methods of minimizing the use of Stickel's algorithm but we have not yet discovered a straightforward, general method. In section 6 we prove that our procedure does indeed terminate with a complete set of unifiers whenever variables are repeated in at most one of the unificands.

Assuming no repeated variables in one term, our algorithm can handle arbitrary repetitions of constants and functional terms. But before disclosing the modification to our algorithm which facilitates handling of repeated arguments, we show with a simple example why the modification is needed. Suppose we wish to unify $f(a, a)$ with $f(x, y)$, which is a subproblem of the earlier example. Without alteration, our algorithm as

	a	a
x	1	0
y	0	1

	a	a
x	0	1
y	1	0

Table 7: Redundant matrix configurations for $f(a,a) = f(x,y)$

so far stated will generate the two configurations shown in Table 7. While the matrix configurations are distinct, they represent identical unifying substitutions – namely $\{x \leftarrow a, y \leftarrow a\}$.

The solution to this problem is surprisingly simple. In short, whenever adjacent rows represent the same term, we require that the contents of the upper row, interpreted as a binary number, be greater than or equal to the contents of the lower row. A symmetric restriction is imposed on columns. Obviously, the information that a variable x_i corresponds to a repeated constant or term must be recorded in some auxiliary data structure in a realistic implementation.

5.6 An Algorithm for Associative-Commutative Unification

Until now, we have concentrated almost exclusively on the matrix solution technique which lies at the heart of our AC unification algorithm. Following is a statement of the unification algorithm proper. This will serve, in the next section, as a basis for results involving the completeness and termination of our method. The algorithm is presented as four procedures: `AC-Unify, Unify-With-Set, Unify-Conjunction, and Matrix-Solve`.

`Procedure AC-Unify`: Given two terms x and y, return a complete set of unifiers for the equation $x =_{AC} y$.

Step 1 If x is a variable, then see if y is a functional term and x occurs in y. If both are true, return `fail`. Otherwise, return $\{\{x \leftarrow y\}\}$, unless $x = y$ — in that case, return the null substitution set $\{\{\}\}$.

Step 2 If y is a variable, then see if y occurs in x. If it does, return `fail`. Otherwise, return $\{\{y \leftarrow x\}\}$.

Step 3 If x and y are distinct constants, return `fail`.

Step 4 If x and y are the same constant, return $\{\{\}\}$.

Step 5 At this point, x and y are terms of the form $f(x_1, \ldots, x_m)$ and $g(y_1, \ldots, y_n)$. If $f \neq g$, return `fail`.

Step 6 If f is not an AC function symbol, and $m = n$, then call procedure `Unify-With-Set` with the substitution set $\{\{\}\}$ and the conjunction of equations $x_1 =_{AC} y_1 \wedge \ldots \wedge x_n =_{AC} y_n$, and return the result. If $m \neq n$, return `fail`.

Step 7 Flatten and sort x and y, if they are not already flattened and sorted, and remove arguments common to both terms. Call the resulting terms $\hat{x}$ and $\hat{y}$, respectively.
Assume $\hat{x} = f(x_1, \ldots, x_j)$ and $\hat{y} = f(y_1, \ldots, y_k)$. Set up the conjunction of equations $f(X_1, \ldots, X_j) =_{AC} f(Y_1, \ldots, Y_k) \wedge X_1 =_{AC} x_1 \wedge \ldots \wedge X_j =_{AC} x_j \wedge Y_1 =_{AC} y_1 \wedge \ldots \wedge Y_k =_{AC} y_k$, where the X_i and Y_i are new, distinct variables. Call this conjunction E.

Step 8 Let T be the result of applying `Matrix-Solve` to the conjunction E. If $T =$ `fail`, return `fail`.

Step 9 Call procedure `Unify-With-Set` with the set of substitutions T and the conjunction of equations $X_1 =_{AC} x_1 \wedge \ldots \wedge X_j =_{AC} x_j \wedge Y_1 =_{AC} y_1 \wedge \ldots \wedge Y_k =_{AC} y_k$, and return the result.

`Procedure Unify-With-Set`: Given a set of substitutions T and a conjunction of equations E, return $\bigcup_{\theta \in T} CSU(\theta E)$, where $CSU(X)$ is a complete set of unifiers for X.

Step 1 Let $S = \{\}$.

Step 2 For each $\theta \in T$, set S to $S \cup \{\bigcup_{\sigma_j \in Z} \{\theta \cup \sigma_j\}\}$, where Z is the result of applying procedure `Unify-Conjunction` to $E\theta$.

Step 3 Return S.

`Procedure Unify-Conjunction` Given a conjunction of equations $E = e_1 \wedge \ldots \wedge e_n$, return a complete set of unifiers for E.

Step 1 . Let V be the result of calling procedure `AC-Unify` with e_1. If $n = 1$, return V. If $V =$ `fail`, return `fail`.

Step 2 . Call procedure `Unify-With-Set` with the set of substitutions V and the conjunction $e_2 \wedge \ldots \wedge e_n$, and return the result.

`Procedure Matrix-Solve` Given a conjunction of equations $f(X_1, \ldots, X_m) =_{AC} f(Y_1, \ldots, Y_n) \wedge X_1 =_{AC} x_1 \wedge \ldots \wedge X_m =_{AC} x_m \wedge Y_1 =_{AC} y_1 \wedge \ldots \wedge Y_n =_{AC} y_n$, where the X_i and Y_i are distinct variables, determine a set of substitutions which will unify $f(X_1, \ldots, X_m)$ with $f(Y_1, \ldots, Y_n)$.

Step 1 Establish an m-by-n matrix M where row i (respectively column j) is headed by X_i (Y_j).

Step 2 Generate an assignment of 1s and 0s to the matrix, subject to the following constraints. If x_i (y_j) is a constant or functional term, then exactly a single 1 must occur in row i (column j). If x_i and y_j are both constants, or if one is a constant and the other is a functional term, then $M[i, j] = 0$. Also, there must be at least a single 1 in each row and column. Finally, if $x_i = x_{i+1}$ for some i, then row i interpreted as a binary number must be less than or equal to row $i + 1$ viewed as a binary number. (Symmetrically for y_j and y_{j+1}.)

Step 3 With each entry $M[i, j]$, associate a new variable $z_{i,j}$. For each row i (column j) construct the substitution $X_i \leftarrow f(z_{i,j_1}, \ldots, z_{i,j_k})$ where $M[i, j_l] = 1$, or $X_i \leftarrow z_{i,j_k}$ if $k = 1$. (symmetrically for Y_j).

Step 4 Repeat Step 2 and Step 3 until all possible assignments have been generated, recording each new substitution. If there is no valid assignment, return `fail`.

Step 5 Return the accumulated set of substitutions.

When there are repeated variables in both unificands, it is possible that our algorithm will not terminate. For example, in the unification of $f(x, x)$ with $f(y, y)$ one of the recursive subproblems generated is identical (up to variable renaming) to the original problem. However, as we prove in the next section our algorithm is totally correct in other cases.

6 Theorems and Such

The intent of this section is to convince even skeptical readers of the viability of our method. Thus we will attempt to establish somewhat carefully the correctness, completeness, and termination of our algorithm for AC matching and unification.

6.1 Partial Correctness

We demonstrate here that, whenever our algorithm terminates, it returns a complete set of unifiers if one exists. We begin by establishing the soundness of certain steps of the algorithm, and then show that each step of the algorithm preserves completeness.

Stickel showed in his paper that like arguments in unificands can be removed without affecting correctness or completeness. We state his theorem here without proof.

Theorem 2 (Stickel) *Let $s_1, \ldots, s_m, t_1, \ldots, t_n$ be terms with $s_i = t_j$ for some i,j. Let θ be a unifier of $f(s_1, \ldots, s_m)$ and $f(t_1, \ldots, t_n)$ and σ be a unifier of $f(s_1, \ldots, s_{i-1}, s_{i+1}, \ldots, s_m)$ and $f(t_1, \ldots, t_{j-1}, t_{j+1}, \ldots, t_n)$. Then θ is a unifier of $f(s_1, \ldots, s_{i-1}, s_{i+1}, \ldots, s_m)$ and $f(t_1, \ldots, t_{j-1}, t_{j+1}, \ldots, t_n)$, and σ is a unifier of $f(s_1, \ldots, s_m)$ and $f(t_1, \ldots, t_n)$.*

The following lemma justifies the variable generalization step of our algorithm.

Lemma 1 *Let $t_1 = f(x_1, \ldots, x_m)$ and $t_2 = f(y_1, \ldots, y_n)$, and let S be the conjunction of equations $f(X_1, \ldots, X_m) =_{AC} f(Y_1, \ldots, Y_n) \wedge x_1 =_{AC} X_1 \wedge \ldots \wedge x_m =_{AC} X_m \wedge y_1 =_{AC} Y_1 \wedge \ldots \wedge y_n =_{AC} Y_n$, where the X_i and Y_j are distinct variables. Let $\sigma = X_1 \leftarrow x_1, \ldots, X_m \leftarrow x_m, Y_1 \leftarrow y_1, \ldots, Y_n \leftarrow y_n$. (1) If θ is a unifier for the equation $t_1 =_{AC} t_2$, $\theta\sigma$ is a unifier for S. (2) Let ϕ be a unifier for S. Then ϕ is a unifier of t_1 and t_2.*

> **Proof:** (1) Obviously σ is a valid substitution, since the X_i and Y_j are variables. Applying σ to t_1 and t_2, we obtain the equation for which θ is a unifier. (2) ϕ applied to S must make both terms in any pair (X_i, x_i) or (Y_i, y_i) equal, since ϕ is a unifying substitution. Hence we may substitute equals for equals and produce the equation $\phi t_1 =_{AC} \phi t_2$, which is no more general than the equation $t_1 =_{AC} t_2$. So ϕ must be a unifier of t_1 and t_2. □

Certainly, if we have a complete unification procedure for S, then we can generate a complete set of unifiers for t_1 and t_2. Lemma 1 tells us that if θ is a unifier of t_1 and t_2, then there exists an equivalent unifier $\theta\sigma$ for S. Assuming the unification procedure is complete, $\theta\sigma$ must be an instance of a unifier returned by the procedure. So we can find all most general unifiers of t_1 and t_2 by determining those for S.

The next lemma is due to Huet; it's proof can be found in his paper [7].

Lemma 2 (Huet) *Let $a_1x_1, \ldots, a_mx_m = b_1y_1, \ldots, b_ny_n$, and let $(x, y) \in \mathcal{N}^m \times \mathcal{N}^n$ be a minimal solution. Then, for any x_i in x, $x_i \leq max(b_1, \ldots, b_n)$ and, for any y_j in y, $y_i \leq max(a_1, \ldots, a_m)$.*

As noted before, the diophantine equation associated with the equation $f(X_1, \ldots, X_m) = f(Y_1, \ldots, Y_n)$, where each X_i and Y_j is a distinct variable, is simply $X_1 + \ldots + X_m = Y_1 + \ldots + Y_n$. Since all coefficients have unit value, we know by Huet's lemma that all components of any basis vector can have a value of either zero or one. Naturally, any basis vector must assign the value 1 to the same number of X_i as Y_i. Also, it is clear that *any* vector which assigns a 1 to exactly one X_i and one Y_j is a basis solution to the equation; call such a solution *special*. Now, any solution vector which assigns 1 to more than a single variable on each side of the equation is reducible by some special vector, since we can select some pair of ones within such a vector and produce a special vector. So only the special vectors are basis vectors, and there are mn of them. This establishes Theorem 1, stated earlier.

The point of all this is simply that our matrix representation is indeed a valid way in which to cast the problem of AC unification. An AC unification problem can be converted to one in which we need only worry about diophantine equations with unit coefficients. This yields a special case of Stickel's algorithm, in which the variable used to label a basis vector can be assigned only to a single variable in each of the generalized unificands; and this information can be conveniently represented in a two-dimensional matrix.

By the isomorphism of AC unification in the all-variables case to the solving of diophantine equations, and by the above facts, it is clear that the procedure `Matrix-Solve` is sound. However, we must also demonstrate that the assignments which it rejects cannot possibly contribute to a complete set of unifiers for a problem. While this seems fairly intuitive, we state it explicitly in the next lemma. We will use CSU to abbreviate "complete set of unifiers", and $CSU(X)$ to denote the complete set of unifiers for X.

Lemma 3 *Let E be the equation $f(X_1, \ldots, X_m) =_{AC} f(Y_1, \ldots, Y_n)$, where the X_i and Y_j are distinct variables, and let S be the conjunction of equations $x_1 =_{AC} X_1 \wedge \ldots \wedge x_m =_{AC} X_m \wedge y_1 =_{AC} Y_1 \wedge \ldots \wedge y_n =_{AC} Y_n$. Then (1) the set of substitutions T returned by the procedure* `Matrix-Solve` *applied to E is a subset of $CSU(E)$; and (2) $CSU(E \wedge S) = \bigcup_{\theta \in T} CSU(S\theta)$.*

> **Proof:** (1) follows by Huet's lemma and by the isomorphism of the solving of linear diophantine equations to the AC unification in the all-variables case. In the case of (2), it is a simple fact that $CSU(E \wedge S) = \bigcup_{\theta \in CSU(E)} CSU(S\theta)$; what remains is to show that the substitutions in $CSU(E) - T$ cannot yield valid unifiers. So let us examine the cases in which `Matrix-Solve` discards substitutions. First, if x_i is a constant, then the assignment of a sum of variables to X_i would make the equation $X_i =_{AC} x_i$ unsolvable. Likewise, if x_i is a functional term, headed by a non-AC function symbol, then the equation $X_i =_{AC} x_i$ is unsolvable when X_i is assigned a sum of variables. And finally, if x_i is a functional term headed by an AC function symbol, the equation $X_i =_{AC} x_i$ is unsolvable when X_i is assigned a sum of variables introduced by `Matrix-Solve`, since any such sum would be headed by

the root AC function symbol. Since all terms are flattened initially, any functional subterm can not be headed by the root AC function symbol. The analysis is similar when some y_j is a constant or functional term. Since constants present in both terms are removed during the preparation step, the assignment of the same variable to X_i and Y_j, when x_i and y_j are both constants, will result in the attempted solution of the equations $z =_{AC} x_i$ and $z =_{AC} y_j$ for some new variable z. But this would force an attempt to unify x_i with y_j; and would fail, since they are distinct constants. A similar argument applies to the case when one of x_i and y_j is a constant and the other is a functional term. Furthermore, each column and row in the matrix set up by `Matrix-Solve` must have at least a single 1 in it, since, otherwise, the effect would be to make some X_i or Y_j "disappear". While this might be appropriate for a theory with an identity element, it is not for associative-commutativity. Lastly, if x_i and x_{i+1} are identical, then interchanging rows i and $i+1$ yields an equivalent substitution. Thus the ordering restriction applied by `Matrix-Solve` preserves completeness. □

We are now in a position to state our main theorem regarding the partial correctness of our algorithm.

Theorem 3 *Given any two terms x and y, the procedure* `AC-Unify` *returns a complete set of unifiers for the equation $x =_{AC} y$, assuming that the algorithm terminates.*

Proof: To see that our procedure is complete, we show that each step either returns a complete set of unifiers for the given problem, or converts the problem to an equivalent one, the solution to which yields a complete set for the original problem.

We look first at the procedure `AC-Unify`. Steps 1 through 5 obviously return a complete set of unifiers for the appropriate input. Step 6 is justified by the fact that $CSU(f(s_1,\ldots,s_n) =_{AC} f(t_1,\ldots,t_n)) = CSU(s_1 =_{AC} t_1 \wedge \ldots \wedge s_n =_{AC} t_n)$, where f is a non-AC function symbol. Step 7 is justified by Lemma 1. Steps 8 and 9 and procedure `Unify-With-Set` together satisfy the conditions of Lemma 3.

Procedure `Unify-Conjunction` is justified by the fact that $CSU(e_1 \wedge \ldots \wedge e_n) = T$, where $T = \bigcup_{\theta \in CSU(e_1)} CSU(e_2\theta \wedge \ldots \wedge e_n\theta)$ Finally, the completeness of procedure `Matrix-Solve` was already established by Lemma 3. □

6.2 Termination

Fages' work [3] is witness to the difficulty of demonstrating termination in the general case of AC unification. We have discovered that some special mechanism is required in order to assure termination of our algorithm in the case that both terms contain repeated variables. We have come up with two alternatives: (1) we could dispatch to Stickel's algorithm, with its attendant proof of termination, in difficult cases; or (2) we could incorporate loop detection into our algorithm. Since we have been unable to prove completeness of the latter, we will be content for now with proving termination for those cases when at least one term does *not* contain repeated variables.

Our strategy is straightforward, if brutal. We will define a complexity measure on equations, which we show to be decreased upon every recursive call of the algorithm.

Let us say that a pair of unificands is *valid* if, in at least one of the unificands, no variable occurs more than once. A pair is *invalid* if both terms contain repeated variables.

First, we show that the algorithm terminates whenever both terms contain only variables, and the terms form a valid pair. Next, we demonstrate that `AC-Unify` will never produce recursive calls involving invalid pairs of terms, assuming that the original arguments were valid.

Theorem 4 *Let $X = f(x_1, \ldots, x_m)$ and $Y = f(y_1, \ldots, y_n)$ be flattened terms, where all of the x_i and y_j are variables, and the x_i are distinct. Then procedure* `AC-Unify` *applied to X and Y terminates.*

Proof:

Trivially, the algorithm terminates if both X and Y comprise distinct variables; this case is dispatched to the matrix operations, which clearly require only a finite number of steps.

Otherwise, `AC-Unify` will produce a list of recursive problems of the form $f(X_1, \ldots, X_m) = f(Y_1, \ldots, Y_n), X_1 = x_1, \ldots, X_m = x_m, Y_1 = y_1, \ldots, Y_n = y_n$.

The first recursive call will be on the equation $f(X_1, \ldots, X_m) = f(Y_1, \ldots, Y_n)$, which contains distinct new variables; hence this case will terminate. The solution substitution generated by the matrix operations will involve only assignments of the form $X_i \leftarrow z_k$ or $X_i \leftarrow f(z_1, \ldots, z_k)$ (similarly for each Y_j). Moreover, each X_i will be assigned a term whose variables are distinct from those in terms assigned to any other X_j, $j \neq i$. This is true for each Y_j, too.

So, after application of the resulting substitution to the remaining equations, the list will be of the form $s_1 = x_1, \ldots, s_m = x_m, t_1 = y_1, \ldots, t_n = y_n$, where the variables in each s_i are distinct from s_j,$j \neq i$, and similarly for the ts (though the variables in each s_i will overlap with those of some t_j). Now, each x_i is a distinct variable, so the next m recursive calls terminate by one of the base cases of `AC-Unify`. In addition, none of the variables in any s_i will have been unified during these recursive invocations (because each call will involve a completely "new" set of variables, even when substitutions are accumulated), and so the variables in the ts remain distinct from one another.

During the last n recursive calls, the left-hand term of the equation being solved will contain distinct variables which have not been "bashed" by previous calls; and the right-hand side will be either a variable (some y variable) or a term composed of distinct variables (after substitution of a unifier for some repeated y variable). Also, the variables in the left-hand side will be disjoint from those in the right-hand side. Thus, the call will terminate by one of the base cases or after a call to the matrix routine.

□

The proof of Theorem 4 is nearly identical to that required for the next fact:

Theorem 5 *Assuming that at most one of X and Y contains repeated variables, then no recursive call generated during the execution of* `AC-Unify` *of X and Y will attempt to unify two terms containing repeated variables.*

Proof: The proof is like that for Theorem 4, except that both X and Y may contain non-variable arguments (though, in at least one of X and Y, no variable may occur more than once). The reader need only convince herself that the unification of two terms in which no variable occurs more than once will produce a most common instance in which no variable occurs more than once. The only difficulty is the case when a repeated variable from one unificand appears in a functional term of the opposite unificand. For example, in $f(X,Z,g(Y,b)) =_{AC} f(Y,Y,g(a,b))$, Y appears as a repeated variable in the right hand unificand, but also appears in a functional term on the left. In cases such as this, the validity of recursively generated problems is guaranteed by the fact that unifiers from `Matrix-Solve` will never assign a term containing repeated variables to a variable which was repeated in the original problem. With these facts, the previous proof can be applied almost directly. □

For concreteness in justifying the next theorem, we shall make use of a directed graph representation of terms. Let t be a term. If t is not yet represented in the graph, then create a new node N. If t is a variable or atomic constant, label N with t. Otherwise, let $t = f(t_1, \ldots, t_n)$. Label N with f, and for each t_i add an edge from N to the root node of the graph of t_i. By a slight abuse of language, we shall often refer to a node as a "variable node", "constant node", or anything else appropriate. We also say that a variable v is "repeated n times" if v textually appears n times in printed representation of a term, or equivalently, if the indegree of the node corresponding to v is n.

When a substitution is applied to a term, we modify its graph representation in the obvious way. Given an assignment $x \leftarrow t$ in the substitution, the representation of t is added to the graph, if necessary, and any edge directed to x is forwarded to the root of the subgraph representation of t.

We now show that `AC-Unify` terminates in the base case, and in general any recursive call to `AC-Unify` is made with a simpler equation.

The complexity metric we will use is a lexicographic extension $\langle R_N, R_{N-1}, \ldots, R_2, R_1, S\rangle$ of noetherian relations. Intuitively, the R_is represent the degree of variable repetition, and S is simply the number of nonvariable symbols present in an equation. Formally, let e be an equation. Let R_m be the number of variables repeated m times in e, and let S be the total number of nonvariable symbols in e. We define the lexicographic composition to be $\langle R_N, R_{N-1}, \ldots, R_1, S\rangle$. N, the maximum degree of repetition, can be determined from the initial equation. S is the total number of nonvariable nodes in the graph representation of both sides of the equation.

Theorem 6 *Let X and Y be terms such that in at least one of them, no variable occurs more than once. Then* `AC-Unify` *terminates.*

Proof: Without loss of generality, assume any repeated variables occur in Y.

Since none of steps 1 through 5 of `AC-Unify` cause recursive calls, they terminate immediately.

If the head of each term is a non-AC function symbol, Step 6 will call `Unify-With-Set` with a set of equations, each one of which is simpler than the original equation. Even if repeated variables exist, each recursive problem will be simpler

than the original. Certainly, if no variables in one new equation are shared by any other, then that equation is simpler than the original despite any substitutions generated by previous equations. If some variable is shared among two equations, then each equation will have less complexity than the original equation, since the original equation had at least one more variable repetition than either subproblem. After any substitution generated from previous equations is applied to one of these equations, it must still be simpler than the original since either (1) some variable is repeated less often, or (2) the degree of repetition is the same, but there are fewer variables, or (3) there are the same number of variables and repetitions, but fewer non variable symbols. Since $\langle R_N, R_{N-1}, \ldots, (R_n - 1), \ldots, (R_{n-m} + k), \ldots R'_1, S' \rangle$ is strictly less than $\langle R_N, R_{N-1}, \ldots, R_n, \ldots, R_{n-m}, \ldots R_1, S \rangle$ lexicographically, R definitely decreases. This is the case because at the very least, the root function symbol does not occur in the subproblem.

If the head of each term is an AC function symbol, termination is more difficult to demonstrate, but it is the case that Steps 7-9 will either terminate or generate recursive calls which are strictly simpler than the original problem.

If all immediate subterms of X and Y are variables, then by Theorem 4 `AC-Unify` terminates.

During Step 7, the conjunction of equations $E\ f(X_1, \ldots, X_m) =_{AC} f(Y_1, \ldots, Y_n)\ \wedge\ X_1 =_{AC} x_1\ \wedge \ldots \wedge\ X_m =_{AC} x_m\ \wedge\ Y_1 =_{AC} y_1\ \wedge \ldots \wedge\ Y_n =_{AC} y_n$, is created. This step certainly terminates.

Step 8 solves the first equation of E, which is made up of distinct variables, and thus by Theorem 4 Step 8 terminates.

Finally, Step 9 generates problems strictly simpler than the original. First, the equations of E involving Xs are solved. All the X equations have sums of distinct variables on the left, and terms from the original problem, the x_is, on the right. After all the equations in E involving Xs are solved, the substitution must have assigned to each variable x_i some sum of new variables, and must have assigned to some new variable each non-variable x_i. Since the non-variable x_i have no variables in common (no repeated variables), each t_j must have no variables in common with any t_k for which $j \neq k$. The remaining equations now look like:

$$t_1 =_{AC} y_1\ \wedge \ldots \wedge\ t_n =_{AC} y_n$$

The variables in each t_j are distinct from any other t_k, but may not be distinct from some y_o, if the original problem was similar to $f(X, Z, g(Y, b)) =_{AC} f(Y, Y, g(a, b))$, in having variables from one side appear as deeper subterms in the opposite unificand.

Now, let y be the variable which is the immediate subterm with the highest degree of repetition in the first subproblem, say r. Also, let s be the number of times y appears as an immediate subterm of the current equation. Obviously, y occurs $r - s$ times elsewhere as a deeper subterm of some functional terms.

Any substitution generated will assign some term, in general $f(z_1, \ldots z_k)$ for some k, or it will assign some subterm t of the original equation to y. In the latter case, let k be the number of variables embedded within t. t must not include repeated variables, since variables can only be repeated in one term originally, and as shown by Theorem 5 this property is preserved. If y is assigned $f(z_1, \ldots z_k)$, the zs are distinct, as shown above. Each equation is strictly simpler than the original since either (1) some variable is repeated less often, or (2) the degree of repetition is the same, but there are less variables, or (3) there are the same number of variables

and repetitions, but there are less non variable symbols. This is the case because at the very least, the root function symbol does not occur in the subproblem.

After each equation is solved, any remaining equation in the conjunction will still be less complex than the original problem. For each variable y with degree of repetition r that appears on the right of an equation, y will be assigned some term $f(z_1, \ldots, z_k)$ for some k. Any remaining equation must still be less complex than the original problem, since R_r will be decremented by at least 1 (y appears one less time), and R_{r-s} will be incremented by k if $(r - s) \geq 0$ (y is replaced by $f(z_1, \ldots, z_k)$ elsewhere). Since $\langle R_N, R_{N-1}, \ldots, (R_n - 1), \ldots, (R_{n-m} + k), \ldots R'_1, S' \rangle$ is strictly less than $\langle R_N, R_{N-1}, \ldots, R_n, \ldots, R_{n-m}, \ldots R_1, S \rangle$ lexicographically, R definitely decreases. If y is not repeated, then $R_i, i > 1$ is uneffected, and R_1 decreases. If there are no variables as immediated subterms (y was replaced by some substitution), $R_i, i \geq 1$ will be unchanged, and the number of nonvariable symbols will decrease, since the immediate nonvariable subterms no longer appear.

Since there are finitely many equations in the conjunction generated by Step 7, and each is less complex than the original problem, the Steps 7-9 terminate. Also, since each possibility of `AC-Unify` either terminates directly, or calls `AC-Unify` with a simpler problem, `AC-Unify` terminates. □

7 Benchmarks

Table 8 reflects the time in seconds necessary to prepare the unificands and to find and construct all AC unifiers. For each problem, timings were supplied by Kapur and Zhang (RRL), Stickel (SRI), and ourselves (MCC). All data were collected on a Symbolics 3600 with IFU. As shown in the table, our algorithm is consistently three to five times faster than Stickel's and Kapur's.

These benchmarks do not include any problems with repeated variables, since in such cases, our algorithm would either return non-minimal sets of unifiers, or it would dispatch to Stickel's procedure. This is not as serious a concession as it might appear, since the most common cases of AC Unification are the ones without repeated variables. In fact, Lankford has found that less than eight percent of uses of AC unification in applications like Knuth-Bendix completion have repetitions of anything, and less than three percent have repetitions on both sides [12].

8 Future Extensions

With simple modifications, our algorithm can apparently handle arbitrary combinations of associativity, commutativity, identity, and idempotence. We say "apparently" because we have not yet proven completeness or termination in all these cases, but preliminary findings have been encouraging. Also, our algorithm seems amenable to parallel implementation. The most efficient sequential implementation of our method makes use of binary numbers to represent the state of the matrix, and thus distributing the search for unifiers only requires communicating a starting and stopping point in the search, and

Problem	♯ solns	RRL	SRI	MCC
$xab = ucde$	2	0.020	0.018	0.005
$xab = uccd$	2	0.023	0.011	0.005
$xab = uccc$	2	0.018	0.008	0.004
$xab = uvcd$	12	0.045	0.047	0.013
$xab = uvcc$	12	0.055	0.032	0.014
$xab = uvwc$	30	0.113	0.096	0.034
$xab = uvwt$	56	0.202	0.171	0.079
$xaa = ucde$	2	0.028	0.013	0.005
$xaa = uccd$	2	0.023	0.009	0.004
$xaa = uccc$	2	0.021	0.006	0.005
$xaa = uvcd$	8	0.043	0.032	0.010
$xaa = uvcc$	8	0.035	0.020	0.011
$xaa = uvwc$	18	0.087	0.062	0.023
$xaa = uvwt$	32	0.192	0.114	0.051
$xya = ucde$	28	0.093	0.094	0.024
$xya = uccd$	20	0.068	0.050	0.018
$xya = uccc$	12	0.045	0.026	0.013
$xya = uvcd$	88	0.238	0.247	0.064
$xya = uvcc$	64	0.211	0.133	0.048
$xya = uvwc$	204	0.535	0.538	0.160
$xya = uvwt$	416	0.918	1.046	0.402
$xyz = ucde$	120	0.375	0.320	0.118
$xyz = uccd$	75	0.185	0.168	0.072
$xyz = uccc$	37	0.093	0.073	0.038
$xyz = uvcd$	336	0.832	0.840	0.269
$xyz = uvcc$	216	0.498	0.431	0.171
$xyz = uvwc$	870	2.050	2.102	0.729
$xyz = uvwt$	2161	5.183	5.030	1.994

Table 8: Benchmarks of AC Unification

the original problem. Other methods, such as Stickel's, probably require either communicating the entire basis of solutions of a diophantine equation or recomputing that basis at each node.

9 Conclusion

We have just described an algorithm which we believe to be the most efficient way of solving a large class of associative-commutative matching and unification problems. Our algorithm is based on a weakened form of Stickel's variable abstraction method, which obviates the need for solving diophantine equations. It utilizes a matrix representation which conveniently enforces powerful search constraints. Compared to Stickel's and Kapur's procedures, our method often yields a significant improvement in speed. Certainly, applications of AC unification stand to benefit from our research.

We would like to thank Dallas Lankford for introducing us to his diophantine basis generation algorithm, and for supplying us with pointers to some useful information. We would also like to thank Hassan Aït-Kaci, Mike Ballantyne, Woody Bledsoe, Bob Boyer, and Roger Nasr for their comments, criticisms, and *laissez-faire* supervision. Finally, we would like to thank Mark Stickel, Hantao Zhang, and Deepak Kapur, for their insightful criticisms and for supplying benchmark times.

References

[1] D.Bananev, D.Kapur, and P.Narendran. "Complexity of Matching Problems". Rewriting Techniques and Applications, Springer-Verlag. Lecture Notes in Computer Science Vol.202, Dijon, France, May, 1985, pp 417-429.

[2] W.Büttner. "Unification in Datastructure Multisets". *Journal of Automated Reasoning*, 2 1986 pp. 75-88.

[3] F.Fages. "Associative-Commutative Unification". *Journal of Symbolic Computation, Vol. 3, Number 3*, June 1987 pp 257-275.

[4] A.Fortenbacher. "An Algebraic Approach to Unification Under Associativity and Commutativity". Rewriting Techniques and Applications, Dijon, France, May 1985, ed Jean-Pierre Jouannaud. Springer-Verlag Lecture Notes in Computer Science Vol. 202, (1985) pp. 381-397

[5] P.Gordan. "Ueber die Auflösung linearer Gleichungen mit reelen Coefficienten". *Mathematische Annalen*, VI Band, 1 Heft (1873), 23-28.

[6] A.Herold and J.Siekmann. "Unification in Abelian Semigroups". *Journal of Automated Reasoning 3* Sept 1987 pp 247-283.

[7] G.Huet. "An Algorithm to Generate the Basis of Solutions to Homogeneous Linear Diophantine Equations". IRIA Research Report No. 274, January 1978.

[8] G.Huet and D.C.Oppen. "Equations and Rewrite Rules: a Survey". In Formal Languages: Perspectives and Open Problems, ed R. Book, Academic Press, 1980.

[9] J.M.Hullot. "Associative Commutative Pattern Matching". *Proc. International Joint Conference on Artificial Intelligence*, Volume One, pp406-412, Tokyo, August 1979.

[10] D.Kapur, G.Sivakumar, and H.Zhang. "RRL: A Rewrite Rule Laboratory". *Proc. of the Eighth Conference on Automated Deduction*, Oxford, England, 1986, Springer-Verlag Lecture Notes in Computer Science Vol.230 pp 691-692.

[11] C.Kirchner. "Methods and Tools for Equational Unification". in *Proc. Colloquium on the Resolution of Equations in Algebraic Structures*, May 1987, Austin, Texas.

[12] D.Lankford. "New Non-negative Integer Basis Algorithms for Linear Equations with Integer Coefficients". May 1987. Unpublished. Available from the author, 903 Sherwood Drive, Ruston, LA 71270.

[13] M.Livesey and J.Siekmann. "Unification of A + C-terms (bags) and A + C + I-terms (sets)". Intern. Ber. Nr. 5/76, Institut für Informatik I, Unifersität Karsruhe, 1976.

[14] A.Martelli and U.Montanari. "An Efficient Unification Algorithm". *ACM Transactions on Programming Languages and Systems*, 4(2):258-282, 1982.

[15] O.Shmueli, S.Tsur, and C.Zaniolo. "Rewriting of Rules Containing Set Terms in a Logic Data Language". *Proc. Principles of Database Systems*, Austin, TX, USA, March 1988.

[16] J.Siekmann. "Universal Unification". *Proc. of the Seventh Conference on Automated Deduction*, Napa, CA, USA, May 1984, Springer-Verlag Lecture Notes in Computer Science Vol.170 pp 1-42.

[17] M.Stickel. "A Complete Unification Algorithm for Associative-Commutative Functions". *Proc. 4th International Joint Conference on Artificial Intelligence*, Tbilisi (1975), pp.71-82.

[18] M.Stickel. "A Unification Algorithm for Associative-Commutative Functions". *Journal of the ACM*, Vol.28, No.3, July 1981, pp.423-434.

[19] M.Stickel. "A Comparison of the Variable-Abstraction and Constant-Abstraction methods for Associative-Commutative Unification". *Journal of Automated Reasoning 3*, Sept 1987, pp.285-289.

[20] H.Zhang. "An Efficient Algorithm for Simple Diophantine Equations". Tech. Rep. 87-26, Dept. of Computer Science, RPI, 1987.