DOUBLE-EXPONENTIAL COMPLEXITY OF COMPUTING A COMPLETE SET OF AC-UNIFIERS

(Preliminary Report)

*Deepak Kapur**and Paliath Narendran*
Institute of Programming and Logics
Department of Computer Science
State University of New York at Albany
Albany, NY 12222

Abstract

A new algorithm for computing a complete set of unifiers for two terms involving associative-commutative function symbols is presented. The algorithm is based on a non-deterministic algorithm given by the authors in 1986 to show the NP-completeness of associative-commutative unifiability. The algorithm is easy to understand, its termination can be easily established. More importantly, its complexity can be easily analyzed and is shown to be doubly exponential in the size of the input terms. The analysis also shows that there is a double-exponential upper bound on the size of a complete set of unifiers of two input terms. Since there is a family of simple associative-commutative unification problems which have complete sets of unifiers whose size is doubly exponential, the algorithm is optimal in its order of complexity in this sense. This is the first associative-commutative unification algorithm whose complexity has been completely analyzed. The approach can also be used to show a single exponential complexity for computing a complete set of unifiers for terms involving associative-commutative function symbols which also have the identity. Furthermore, for unification in the presence of associative-commutative-idempotent operators we get a doubly exponential bound.

1 Introduction

Given two first-order terms s and t involving associative-commutative (abbreviated as ac) function symbols, the ac unifiability check is to determine whether there exists a substitution σ such that

$$\sigma(s) =_{ac} \sigma(t),$$

where $=_{ac}$ stands for the equivalence relation on terms generated by the associativity and commutativity of ac function symbols. (We will call $\sigma(s)$ and $\sigma(t)$ as ac-equivalent terms.) If there is a σ satisfying the above equation, then σ is called an ac-unifier of s and t.

A set S of substitutions is called a complete set of ac-unifiers for s and t if

- every substitution in S is an ac-unifier of s and t, and
- any ac-unifier θ of s and t can be obtained from some substitution, say σ, in S, i.e., there is a θ' such that $\theta =_{ac} \theta' \circ \sigma$,

where two substitutions are ac-equivalent if for every variable, the terms they substitute are ac-equivalent (restricted to the variables in s and t).

Associative-commutative unification plays an important role in several areas of computer science and artificial intelligence, including resolution-based theorem proving, term rewriting systems, logic programming, constraint-based programming, functional programming, type inference, data base query languages, etc. AC-unification has been used in theorem provers such as RRL to prove nontrivial mathematical problems, for instance, see (Kapur and Zhang, 1991).

AC-unification is also an interesting problem because of the long history in trying to understand the problem. To the best of our knowledge, the first algorithms for ac-unification were by Livesey and Siekmann (1976) and Stickel (1975, 1981). Of these Stickel's recursive algorithm is more general and was

*Partially supported by the National Science Foundation grant no. CCR-8906678.

also shown to be complete. However, due to its complicated nature, termination of this algorithm could be shown only for a simple (restricted) subcase in (Stickel, 1981). Livesey and Siekmann's algorithm dealt only with terms involving a single ac function symbol having variables and constants as arguments. This algorithm was subsequently extended to general terms by Herold and Siekmann (1985). The question of termination of Stickel's ac unification algorithm remained an open problem for nearly a decade primarily because of the top down nature of the algorithm which generated subproblems in which the sizes of terms are bigger than the size of input terms. Its termination was proved by Fages (1984) using an ingenious complexity measure.

Each of the above algorithms checks for ac-unifiability and, if the given terms are ac-unifiable, computes a complete set of ac-unifiers. Recently, variations of these algorithms patterned after Martelli and Montanari's presentation of Robinson's unification algorithm have been proposed (Kirchner, 1989; Boudet et al, 1990). These presentations are simpler and more elegant in the sense that they emphasize the key steps in the algorithm and separate the control structure of the algorithm from other details. The algorithm by Boudet et al (1990), in particular, uses an efficient algorithm for computing a basis of non-negative solutions for a system of linear diophantine equations.

Nothing is known about the complexity of any of these algorithms. (Some partial analysis is done in Boudet et al (1990).) It is however likely to be worse than exponential in the size of the input since there are simple terms which can have hundreds of thousands most general unifiers (Bürckert et al, 1988). It was generally believed that the complexity of checking ac-unifiability is quite high – perhaps even non-primitive-recursive or super-exponential. In our attempts to study the ac-unification problem, we proved in 1986 that the ac-unifiability check can be done in non-deterministic polynomial time, thus showing the ac-unifiability check to be NP-complete (Benanav, Kapur and Narendran, 1985; Kapur and Narendran, 1986b[1]; Kapur and Narendran, 1987).

In this paper, we design a deterministic algorithm for computing a complete set of ac-unifiers; the algorithm is based on the non-deterministic algorithm presented in our earlier paper (Kapur and Narendran, 1986b). The key ideas of the algorithm are:

1. A decision tree is built by
   - considering whether non-variable subterms with the same outermost ac function symbol can be unified or not. This results in an equivalence relation on subterms.
   - by choosing the outermost function symbol of non-variable term substitutions for variables.

2. Solving diophantine equations is delayed all the way to the end, thus attempting to exploit the structure of the terms as much as possible. A leaf in the decision tree is a set of disjoint systems of simultaneous linear diophantine equations corresponding to each ac function symbol with constraints of the form
   - $c_1 \neq c_2$, denoting that the equivalence class of subterm c_1 is not the same as the equivalence class of subterm c_2, and
   - a constant c does not appear in the substitution for a variable x.

The decision tree is so constructed that a complete set of ac-unifiers of s and t is the union of complete sets of ac-unifiers of the unification problem corresponding to each leaf node. It is shown that each leaf node can be solved using efficient algorithms for diophantine equations and ac-unifiers can be constructed after performing an occur check.

The algorithm is simple to understand. Its termination is obvious. Further, there is considerable flexibility and possibility of using heuristics to speed up the computation as well as to discard paths which are likely to lead to leaf nodes which do not produce any ac-unifiers. Most importantly, the complexity of the algorithm can be easily analyzed. Computing a complete basis of non-negative solutions of simultaneous linear diophantine equations can be done in exponentially many steps. Using the basis, ac-unifiers can be constructed by considering all subsets, thus, in the worst case, needing doubly exponentially many steps. Since there are only exponentially many leaf nodes in a decision tree, the complexity of our algorithm has an upper-bound of double-exponentially many steps (i.e., there is a polynomial $p(n)$, where n is the input size, such that the number of steps is $O(2^{2^{p(n)}})$). This also gives a double-exponential bound on the size of a complete set of ac-unifiers. To our knowledge, this is the first ac-unification algorithm whose complexity has been completely analyzed.

Using the same approach, we can easily show better bounds for associative-commutative-identity unification problem for terms in which the ac function

[1] A final version of this paper is to appear in the *Journal of Automated Reasoning* sometime in 1992.

symbols also have an associated unit element or identity. A basis of nonnegative solutions of a system of simultaneous linear diophantine equations still needs to be computed, and requires exponentially many steps. However, a complete set of most general unifiers can be computed from the basis in steps proportional to the basis size. The complexity of the acu-unification algorithm is thus single-exponential, and there is also a single-exponential bound on the size of a complete set of unifiers.

The problem of partitioning a number of objects into different bins can be easily encoded as an ac unification problem (i.e. solving a diophantine equation) such that every distinct (ordered) partition corresponds to a distinct non-negative minimal solution of the diophantine equation. Since in the worst case, there can be exponentially many ordered partitions of a given number n, a complete basis of non-negative solutions has exponentially many minimal solutions. From this, one gets doubly exponentially many combinations of minimal solutions each of which corresponds to a minimal ac unifier. Domenjoud (1989) discussed a family of such ac unification problems with one ac function symbol:

$$x + \cdots + x = (y_1 + \cdots + y_1) + \cdots + (y_k + \cdots + y_k).$$

It is possible to get the impression that for more complex problems involving more than one ac function symbol, the number of steps is likely to be more than doubly exponential and number of unifiers is also likely to be more than doubly exponential. Our results show that this is not so; in fact, partitioning problems exhibit the worst case complexity of the problem. Having more ac function symbols or complex nesting in the input terms do not really add to the asymptotic complexity of the problem. In fact, one may get additional constraints that could drastically reduce the number of unifiers.

2 Algorithm

We first present the major steps of the nondeterministic algorithm for ac-unifiability presented in (Kapur and Narendran, 1986b). We then discuss steps which should be changed to get a deterministic algorithm to generate a complete set of ac-unifiers. Finally, we give a detailed description of the deterministic algorithm.

Throughout the rest of the paper, we assume that terms are *flattened*, i.e., a binary ac function symbol can be viewed to be of arbitrary arity, so a term $+(+(t_1, t_2, ..., t_k), t_{k+1})$ is replaced by $+(t_1, t_2, ..., t_k, t_{k+1})$.

By an admissible subterm of t we mean any subterm t' which correspond to a subtree in the tree representation of t (in flattened form). For instance, in $+(x, f(g(x)), y)$, where $+$ is the only ac function symbol, $+(x, f(g(x)), y)$, $f(g(x))$, x, $g(x)$, and y are admissible subterms, whereas $+(x, y)$ is not an admissible subterm. A variable x in a term t is said to *occur directly under a function symbol* f if and only if there are occurrences of x and f in t such that x is an argument of f. In the example, x and y occur directly under $+$ and x also occurs directly under g.

The key steps in the nondeterministic algorithm presented in (Kapur and Narendran, 1986b) were the following:

(a) Choice of equivalence classes of admissible subterms that get unified. (See the footnote on the next page.)

(b) Choice of the outermost function symbol in the substitution for a variable.

(c) Decomposition into systems of word equations, solving the word equations, performing occur check, and generating a (sequential) unifier corresponding to a solution.

The nondeterministic algorithm can be turned into a deterministic algorithm for *ac-unifiability,* by constructing a decision tree where every leaf node corresponds to a set of word equations and constraints, and the path to that leaf corresponds to the choices made on the way. This results in an algorithm that takes exponential time and space.

Since our goal here is to find a *complete set of unifiers,* and not merely a single unifier, it is necessary to change the last step, where we try to solve the word equations. Instead of checking for the existence of a solution, we must generate a *complete* set of solutions. This can be done using a modified version of algorithms for solving simultaneous linear diophantine equations discussed in the literature or particularly, a modified version of the algorithm given by (Boudet et al, 1990) with a crucial difference: there are constraints imposed because of the occur-check.

We now give the deterministic algorithm. The input is a system of equations of the form

$$s_1 = t_1, \quad s_2 = t_2, \quad \cdots, \quad s_k = t_k.$$

We give a distinct name to each of the non-variable and non-constant admissible subterms in $s_1, \cdots, s_k, t_1, \cdots, t_k$.

We first discuss how a decision tree is built and later discuss how to solve each of the leaf nodes in the decision tree. For every solution in the complete set of solutions we find at the leaf node, we can construct a sequential unifier for the original terms.

2.1 Computation at a Node

At every node (both leaf and nonleaf nodes) of the decision tree, the following steps are performed.

Step 1: Decomposition. Repeatedly decompose an equation into many equations when the outermost function symbol of the terms in an equation is not ac. For instance an equation

$$g(s_{11}\ ,\ s_{12}\ ,\ \cdots, s_{1i})\ =\ g(t_{11}\ ,\ t_{12}\ ,\ \cdots, t_{1i}),$$

where g is a non-ac function, decomposes into

$$\{s_{11}\ =\ t_{11}, \qquad s_{12}\ =\ t_{12}, \qquad \cdots, \qquad s_{1i}\ =\ t_{1i}\}.$$

Step 2: Processing. There is no need to introduce a trivial equation with the same left side and right side. If there are two contradictory constraints, the node is declared a *dead-end* as it will not generate any unifier. If an equation of the form $x = y$ is generated, then we replace x by y in all the other constraints at the node, and extend the partial substitution (also called the *solved form* of the constraints) built to include the substitution $\{x \leftarrow y\}$.

An equation $f(s_1, s_2, \cdots, s_{k_1}) = f(t_1, t_2, \cdots, t_{k_2})$, where f is ac, is simplified to $f(s_2, \cdots, s_{k_1}) = f(t_2, \cdots, t_{k_2})$ if s_1 and t_1 are the same terms or $s_1 = t_1$ is an equation at the node.

If an equation of the form $x = f(\cdots, x, \cdots)$ (violation of occur check) or $f(\cdots) = g(\cdots)$ (function clash) is encountered, that node is declared to be a dead-end as there is no unifier.

2.2 Extending Decision Tree

We first make decisions based on identifying admissible subterms in equations.[2] Then we decide about the outermost ac function symbols of substitutions for variables occurring directly under different ac function symbols in equations. In both cases, the number of decisions made is bounded by the size of the input terms.

When a decision tree is extended, the descendants inherit the constraints from their ancestors. So, we will mention only new constraints imposed at nodes.

Step 3: Choosing two subterms to be made equivalent. Choose two admissible subterms, say g_i and g_j, which can potentially unify. Extend the decision tree by generating two nodes from this node: (i) the node has an additional constraint $g_i = g_j$, and (ii) the node which has the additional constraint $g_i \neq g_j$.

After repeated application of these steps, we have leaf nodes some of which are dead-ends because they will not generate any ac-unifier, while others can potentially generate an ac-unifier. The set of subterms is thus partitioned into several equivalence classes, each of which is given a distinct name.

There are three kinds of constraints: (i) equations relating variables to nonvariable subterms, (ii) equations relating nonvariable subterms such that the outermost symbol of the two sides of each equation is an ac function symbol, and (iii) inequations of the form $s_1 \neq s_2$, stating that s_1 and s_2 are not unifiable (by the unifier(s) we have targetted).

Equations relating nonvariable subterms with an ac outermost symbol (or a variable with a nonvariable subterm) can be partitioned based on their outermost ac symbol into subsystems of equations involving the same ac symbol. However, different subsystems may share variables, i.e., the same variable may be occurring directly under different ac function symbols.

Step 4: Choose an outermost ac function symbol for a variable. For a node in which a variable occurs directly under different ac function symbols, the decision tree is further extended. If a variable x appears as an argument to different function symbols $f_1, \cdots, f_i$, extend the tree by adding $i+1$ nodes such that for each $1 \leq j \leq i$, the new node has $x = f_i(x_1, x_2)$ as an additional constraint, where x_1, x_2 are new variables, and the $i+1^{th}$ node has $x \neq f_1(x_1, x_2), \cdots, x \neq f_i(x_1, x_2)$ as additional constraints. For each of these nodes, except the last, we replace x by the corresponding $f_i(x_1, x_2)$ everywhere. In the $i+1^{th}$ node, x is treated as a constant.

2.3 Solving a Leaf Node

This is computationally the most expensive as well as the most interesting step in the algorithm. As stated earlier, it involves solving simultaneous linear diophantine equations for generating a complete basis

[2] It is often not necessary to consider subterms which appear on the same side of constraints. Also it is usually only necessary to consider subterms appearing under the same ac function symbol. There is a lot of scope for using clever heuristics in this step which would rule out paths not likely to result in generation of any unifiers. Details about such heuristics which play an important role in an implementation of this algorithm are omitted from the extended abstract.

of nonzero positive solutions as well as performing the occur check to check whether a non-zero positive solution leads to a unifier. Note that whatever optimal algorithms we have for solving simultaneous equations can be used here.

At every leaf node, we have partitioned all subterms of the flattened representations of the input terms into several equivalence classes which are given distinct names, say b_1, b_2, $\cdots$, b_m. Further, equations have the property that no top level variables appear under different ac function symbols. Because of step 4, we can partition equations with the same outermost ac function symbols into disjoint subsystems which can be solved independently. Let E_f stand for the subsystem of equations with f as the outermost function symbol.

Step 5: Solving Word Equations. Form word equations from the equations in each E_f as follows: For every equation, (a) replace each top-level nonvariable argument to f on the left side as well as the right side by the name of its equivalence class, and (b) drop f. For instance, if $f(x,\ x,\ g(y,z),\ u) = f(w,\ w,\ w)$ is an equation in E_f, then form the word equation $xxb_iu = www$ where b_i is the name of the class $g(y,z)$ belongs to.

A complete basis of nonzero positive solutions to these groups of equations involves two steps: (i) computing a complete basis of nonnegative solutions, and (ii) generating a basis of all nonzero positive solutions from the basis solutions by forming appropriate combinations of nonnegative solutions (Huet, 1978; Stickel, 1981). Such a complete basis can be obtained using an algorithm for solving a system of simultaneous linear diophantine equations such as in (Gathen and Sieveking, 1978; Schrijver, 1986) or in (Boudet et al, 1990).

Not every solution to these equations, however, would lead to a unifier for the original terms. There could be cyclic dependencies that are unsatisfiable. (This can be checked using the basis of nonnegative solutions before forming their combinations in order to generate a nonzero positive solution.) For instance, if b_i is an equivalence class containing a nonvariable term with the variable x, then no solution to x can involve b_i, i.e., x cannot get the substitution b_i or $f(...,b_i,...)$. Such a solution to x must be rejected. This problem is handled in the next step, *Occur Check.*

For example, consider $f(x,y,g(x)) = f(z,z)$; we name the equivalence class of $g(x)$ as b. We get a word equation: $xyb = zz$, which results in a diophantine equation:

$$x + y + b = 2z.$$

The basis set of nonnegative solutions include 6 solutions including the solution $x = 1, y = 0, b = 1, z = 1$. A nonzero positive solution to the diophantine equation that includes this solution will result in a substitution $x \leftarrow b$ and/or $x \leftarrow f(...,b,..)$, which fails because of the occur check since b stands for $g(x)$.

This step thus involves:

1. Creating new variables for names of equivalence classes (variable abstraction a la Stickel).

2. Generating solutions for the resulting set of equations over nonnegative integers (i.e, over the *monoid* in this case). These form the solution matrix.

3. Weeding out those solutions (i.e., delete the rows in the solution matrix) which violate the occur check discussed below (step 6).

4. Forming combinations making sure that the entries for the variables abstracting the names of equivalence classes remain 1.

5. Generating the sequential unifier of s and t from each combination.

Step 6: Occur Check. We set up a directed graph G with a node labeled with each variable and each new constant symbol b_i. If x appears in any of the subterms in the equivalence class corresponding to b_i, there is a directed edge from the node labeled b_i to the node labeled variable x. If there is an equation $x = t'$, where t' is an admissible subterm, in the partial substitution built so far (i.e., the solved form) there is an edge from the node labeled x to the node labeled b_i, the equivalence class containing t'.

For every nonnegative solution ϕ of equations from step 5, the graph G is extended to a graph $\phi(G)$ to include an edge from the node labeled x to the node labeled b_i if b_i appears in the substitution for x in ϕ.

If a solution ϕ to the above equations can be found such that the extended graph $\phi(G)$ is acyclic, then the terms are ac-unifiable.

Note that the graph G that is built using the equivalence relation is basically a refinement of the graph of the flattened tree representations where additional edges corresponding to the equivalence have been added. This is similar to the occur check in the standard unification problem as in (Paterson and Wegman, 1978).

2.4 Proof of Correctness

A proof of correctness of the algorithm follows from the following two lemmas:

Lemma 1: Every unifier in a complete set of unifiers generated at a leaf node is also a unifier of the original input terms.

Lemma 2: Any unifier of the input terms must satisfy the conditions imposed (such as equivalence classes among admissible subterms and occur-check constraints) at one of the leaf nodes of the tree, and is an instance of a unifier in a complete set of unifiers in that leaf mode.

Sketch of Proof: Without loss of generality, consider a ground unifier. Now, as in the case of ac-unifiability in (Kapur and Narendran, 1986b), t can be shown that the constraints this unifier imposes on the admissible subterms (e.g., which subterms get unified) the outermost function symbols etc. correspond to one of the paths in the decision tree. □

The completeness of the algorithm follows from the completeness of the algorithm for finding a complete basis of nonzero positive solutions of the diophantine equations.

The above algorithm has many interesting features:

1. It is conceptually simple to present and also to understand.

2. The proof of termination of the algorithm is obvious.

3. Given that solving diophantine equations is being deferred to the maximum possible extent, we would have gathered most of the constraints on variables placed in a unification problem.

4. With additional constraints, a diophantine equation is likely to have a smaller number of solutions than an unconstrained diophantine equation. This will drastically reduce the number of cases we have to consider for terms with many levels of ac function symbols.

5. Many steps in the algorithm can be parallelized; in particular, each of the leaf nodes can be constructed and processed in parallel. Similarly, many operations within a leaf node can be potentially parallelized – for instance, the occur-check can be done in parallel for the different solutions obtained by solving the diophantine equations.

It should be noted that the benefit of simultaneously solving a system of diophantine equations over solving the equations sequentially was pointed out in (Kirchner, 1990). However, his algorithm exploited simultaneous equations arising from the same ac-function symbol, whereas we show how several ac-symbols can be handled simultaneously by separating the unification problem into different *independent* subproblems.

There are many improvements possible to prune the decision tree. For instance, equating variables can be postponed to the last step. There too one needs to equate only variables that occur under the same function symbol. Much of the occur-check can be done as the equivalences are chosen among the admissible subterms, using suitable data structures for the flattened tree representations of the terms. There is also scope for several heuristics that can be applied at various levels. These ideas will be tried in an implementation of this algorithm and discussed in an expanded version of the paper.

In the next section we illustrate the algorithm using an example.

3 Example

Consider the terms

$$\begin{aligned} s &= h(*(+(x,a),+(y,a),+(z,a)),x) \text{ and} \\ t &= h(*(+(w,w,w),z,z),x), \end{aligned}$$

where $+$ and $*$ are the only ac-function symbols. We replace $+(x,a)$, $+(y,a)$, $+(z,a)$, $+(w,w,w)$, $*(+(x,a),+(y,a),+(z,a))$, $*(+(w,w,w),z,z)$ by g_1, g_2, g_3, g_4, g_5 and g_6 respectively. The root node of the tree is:

$$1. \quad \{s=t, g_5 = *(g_1,g_2,g_3), g_6 = *(g_4,z,z), g_1 = +(x,a), g_2 = +(y,a), g_3 = +(z,a), g_4 = +(w,w,w)\}$$

By the decomposition step, we get

$$g_5 = g_6.$$

Below we shall repeat equations for g_1, g_2, g_3, g_4, g_5 and g_6 only when they change because of substitutions made.

We can split based on making g_1 equivalent to g_4, which gives us two nodes:

$$1.1 \quad \{*(g_2,g_3) = *(z,z), +(x,a) = +(w,w,w)\}.$$

$$1.2 \quad \{*(g_1,g_2,g_3) = *(g_4,z,z), g_1 \neq g_4\}.$$

Node 1.1 can be extended further by splitting based on identifying g_2 with z.

$$1.1.1 \quad \{g_2 = z, g_3 = z, +(y,a) = +(z,a), +(x,a) = +(w,w,w)\}.$$

$$1.1.2 \quad \{*(g_2,g_3) = *(z,z), g_2 \neq z, +(x,a) = +(w,w,w)\}.$$

Node 1.1.1 can be further simplified by processing the third equation, which gives $y = z$, so we can replace y by z in 1.1.1 and find that the first equation cannot be solved because of failure of occur check. So 1.1.1 is a dead end as it does not give any unifier. Node 1.1.2 can be further extended by identifying g_3 with z, but both branches ($g_3 = z$ and $g_3 \neq z$) can easily be seen to be dead-ends. Thus the paths from node 1.1 do not result in any unifiers.

Consider node 1.2. It can be extended based on identifying g_2 with g_4 resulting in

1.2.1 $\{*(g_1, g_3) = *(z, z),$
$g_2 = g_4, g_1 \neq g_4\}.$

1.2.2 $\{*(g_1, g_2, g_3) = *(g_4, z, z),$
$g_2 \neq g_4, g_1 \neq g_4\}.$

Node 1.2.1 can be extended based on identifying g_1 with z:

1.2.1.1 $\{g_3 = z, g_1 = z, +(x, a) = (z, a),$
$+(y, a) = +(w, w, w), g_1 \neq g_4\}.$

1.2.1.2 $\{*(g_1, g_3) = *(z, z), +(y, a) = +(w, w, w),$
$g_1 \neq z, g_1 \neq g_4\}.$

Similar to nodes 1.1.1 and 1.1.2, neither of these nodes give any unifier either.

Node 1.2.2 can be extended by identifying g_3 with g_4:

1.2.2.1 $\{*(g_1, g_2) = *(z, z),$
$+(z, a) = +(w, w, w), g_2 \neq g_4, g_1 \neq g_4\}.$

1.2.2.2 $\{*(g_1, g_2, g_3) = *(g_4, z, z),$
$g_3 \neq g_4, g_2 \neq g_4, g_1 \neq g_4\}.$

Node 1.2.2.2 is a leaf node; it will not give any unifier since the first equation cannot be solved. So, all the unifiers will be generated from node 1.2.2.1.

Node 1.2.2.1 can be further extended by identifying g_1 with z giving:

1.2.2.1.1 $\{g_2 = z, g_1 = z, +(x, a) = +(y, a),$
$+(z, a) = +(w, w, w), g_2 \neq g_4, g_1 \neq g_4\}.$

1.2.2.1.2 $\{*(g_1, g_2) = *(z, z), +(z, a) = +(w, w, w),$
$g_1 \neq z, g_2 \neq g_4, g_1 \neq g_4\}.$

Node 1.2.2.1.2 does not give any unifier because as before, the first equation cannot be solved. Node 1.2.2.1.1 can be further processed to give a partial unifier $x = y$ which is substituted everywhere, so we have:

1.2.2.1.1 $\{z = g_1, x = y, g_1 = +(y, a),$
$+(z, a) = +(w, w, w), g_2 \neq g_4, g_1 \neq g_4\}.$

Figure 1: Equivalences at Node 1.2.2.1.1

See figure 1 for a pictorial representation of the equivalences in terms of the original terms.

These can be split further on $y = w$ and $y \neq w$, giving rise to

1.2.2.1.1.1 $\{z = +(y, a), +(z, a) = +(y, y, y),$
$x = y = w, g_2 \neq g_4, g_1 \neq g_4\}.$

1.2.2.1.1.2 $\{z = +(y, a), +(z, a) = +(w, w, w),$
$x = y, y \neq w, g_2 \neq g_4, g_1 \neq g_4\}.$

These could be split further, but at this stage we can see that the node 1.2.2.1.1.1 will lead to the unifier

$$\{z = +(a, a), x = y = w = a\},$$

and 1.2.2.1.1.2 will lead to

$$\{z = +(v_1, v_1, v_1, a, a), w = +(v_1, a),$$
$$x = y = +(v_1, v_1, v_1, a)\}.$$

4 Doubly exponential complexity of the algorithm

The complexity of the algorithm depends on two factors: (i) the size of the decision tree, in particular the number of leaf nodes, and the input size of the diophantine equations and the occur-check graph; (ii) the complexity of finding a complete set of nonzero positive solutions for the diophantine equations that satisfy the occur-check constraints.

Lemma 3: The number of leaf nodes in the decision tree is exponential in the size of the input terms.

Sketch of proof: Each path to a leaf node in the decision tree effectively partitions the set of admissible subterms of the flattened representations of input terms.

Lemma 4: The input size of the system of diophantine equations is polynomial in the size of the input terms.

We now show that the worst-case complexity of finding a complete set of nonzero positive solutions for a set of word equations is doubly exponential in the size of the input. It is this step of the algorithm in the previous section that determines the worst-case complexity of the algorithm.

Lemma 5: The coefficients in the set of basis solutions for a system of linear equations over the nonnegative integers are bounded by a number that is exponential in the size of the input.

From this it also follows that

Lemma 6: The number of basis solutions over the nonnegative integers for systems of linear equations is $O(2^n)$ in their size.

A complete basis of nonnegative solutions of a system of simultaneous linear diophantine equations can be computed using algorithms in (Gathen and Sieveking, 1978; Papadimitriou, 1981; Schrijver, 1986 (Theorem 17.1)). Even though these papers on the integer programming problem are concerned with deriving a bound on a *single* basis solution, the constructions there are general enough to derive the same bound on every minimal solution in a complete basis. These constructions typically involve deriving from a solution in which the values of some variables are bigger than the bound, another solution in which the values of all the variables are smaller than the bound. It will be interesting to develop algorithms based on Hermite and Smith normal form constructions which work better than the brute-force algorithm suggested by the bounds.

Violation of occur-check can only cause some solutions to be discarded from the complete basis of nonzero solutions. Thus the occur-check step cannot add to the number of solutions in the basis. For generating a complete basis of nonzero positive solutions, since all possible combinations (i.e., all possible subsets of the set of solutions) might have to be tried, another exponentiality is added. Thus the complexity of the algorithm is no worse than doubly exponential in the sizes of the input terms.

Lemma 7: Given a set of diophantine equations, we can find a complete set of nonzero positive solutions for them over the positive integers in time doubly exponential in the input size.

We have thus shown that

- the decision tree has exponentially many leaf nodes, and the size of the data (i.e., the equations) at these nodes is polynomial in the size of the inputs, and
- solving the equations at these nodes along with the constraints is of double-exponential worst-case complexity.

Thus,

Theorem 1: Given two terms, the complexity of finding a complete set of (sequential) ac-unifiers for them is doubly exponential in the size of the input. Furthermore, the number of unifiers in a complete set is bounded by a number that is doubly exponential in the size of the input terms.

This bound is tight because of the following connection observed by Domenjoud between the partitioning problem and a simple ac unification problem.

Theorem (Domenjoud, 1989): The number of minimal unifiers of the equation

$$\alpha x_1 + \cdots + \alpha x_p =_{ac} \beta y_1 + \cdots + \beta y_q$$

is doubly exponential in $\alpha p + \beta q$.

It is possible to represent positive solutions obtained by combining different subsets of a complete basis of nonnegative solutions in a compact fashion. For each minimal combination needed to ensure positive values for each variable, other elements in the basis can be either included or not included. Each such minimal combination need to have at most k elements, where k is the number of variables in the system of diophantine equations, and k is bounded by the size of the input. This implies that all combinations (doubly exponentially many in the size of the input) can be represented compactly in $O(2^{nk})$ space which is a single exponential in the size of the input.

5 AC Unification in the Presence of Unity

The case when the ac function symbols in the terms also have an identity (or unity, denoted by '1') is a little easier in terms of the worst case complexity. The decision tree must be extended a little, since we make different choices for the variables that are set to be equal to 1 right at the beginning. Once this is done the problem reduces to ac-unification. Though this step is exponential in complexity, it does not alter the asymptotic complexity of building the decision tree, which remains single-exponential. Note that for a most general unifier in which a variable gets the substitution 1, that substitution is made right at the beginning while constructing a decision tree. The new variables introduced in the process of 'fixing' outermost ac-function symbols can be assumed to be not equal to 1. For instance, when f is chosen to be the outermost ac function symbol for x, the assumption is that f is the outermost function symbol of the *normalized* substitution. Hence the variables x_1 and x_2 (in the term $f(x_1, x_2)$ substituted for x) will not get 1 as the substitution. Thus the occur check of the form $x = f(..., x, ...)$ still causes failure.

The rest of the decision tree as well as the solution matrix for the diophantine equations are constructed as before. By Lemma 6, there are exponentially many basis solutions in a complete basis. However, unlike the case in Lemma 7, we can generate a most general unifier by just taking a single combination of all the basis solutions, which corresponds to the "sum" of all the rows in the solution matrix. This is so because any combination can be obtained from this single combination by substituting '1' for the variables corresponding to those basis solutions not included in the combination. Thus the overall complexity of generating a complete set of unifiers is single-exponential.

Lemma 8: Let B be a complete basis of nonnegative integer solutions of a system of linear diophantine equations corresponding to an ACU function symbol f at a leaf node. The substitution consisting of $x_i \leftarrow f(z_1, ..., z_1, z_2, ..., z_2, ..., z_j, ..., z_j, ...)$ for each x_i is the most general unifier for the ACU unification problem corresponding to the system of diophantine equations, where the variable z_j corresponds to the j-th solution in B and z_j occurs as many times in the term substituted for x_i as the integer value of x_i in the j-th solution (if x_i is 0 in j-th basis solution, then z_j does not appear in the substution for x_i).

It is easy to see that a unifier corresponding to any combination of a subset of basis solutions in B can be easily obtained from the above most general unifier by instantiating '1' for the variables corresponding to basis solutions not included in the combination. Thus,

Theorem 2: Given two terms, the complexity of finding a complete set of (sequential) acu-unifiers for them is exponential in the size of the input. Furthermore, the number of unifiers in a complete set is bounded by a number that is exponential in the size of the input terms.

6 ACI Unification

The above paradigm of using a decision tree to "unravel" nondeterministic computations can also be applied for unification in the presence of associative-commutative-idempotent (aci) operators. A nondeterministic algorithm for aci-unifiability was given in (Kapur and Narendran, 1986b). The basic paradigm used there was mostly the same as that used for ac-unifiability. However, in the case of aci too (as for acu), we look for substitutions which are *normalized,* i.e. for all variables x, $\theta(x)$ is irreducible using the idempotency rules modulo ac. (The idempotency rules by themselves constitute a canonical rewrite system modulo associativity and commutativity.) The main steps are, again:

(a) choice of equivalence classes of admissible subterms that get unified.

(b) Choice of the outermost function symbol in the *normalized* substitution for a variable.

(c) Decomposition into systems of word equations, solving the word equations, performing occur check, and generating a (sequential) unifier corresponding to a solution, except that here the word equations are solved over a commutative-idempotent semigroup.

Thus the deterministic algorithm also works in roughly the same way as that outlined in Section 2. Steps 1, 2, 3, 4 and 6 can be carried over without any significant change. In Step 5 we have to solve word equations over a commutative-idempotent monoid and combine them to form most general solutions over a commutative-idempotent semigroup. In other words, we first have to solve diophantine equations where the solutions are either 0 or 1, and then take combinations of those such that entries for all variables are nonzero.

This step can be done in time no worse than doubly exponential. Thus,

Theorem 3: Given two terms, the complexity of finding a complete set of (sequential) aci-unifiers for them is doubly exponential in the size of the input. Furthermore, the number of unifiers in a complete set is bounded by a number that is doubly exponential in the size of the input terms.

This bound also happens to be tight. A class of hard examples of equations over a commutative idempotent monoid can be obtained as follows. Let x, x_1, x_2, x_3, y_1, y_2, y_3, z_1, z_2, z_3 be variables. Consider the equations

$$\begin{aligned} x &= x_1 x_2 x_3 \\ x &= y_1 y_2 y_3 \\ x &= z_1 z_2 z_3. \end{aligned}$$

In every minimal solution, x is 1, exactly one of x_1, x_2, x_3, exactly one of y_1, y_2, y_3 and exactly one of z_1, z_2, z_3 is 1. Clearly there are $3^3 = 9$ minimal solutions for these equations. This example can be generalized to n equations and m variables, with m^n solutions.

Combining these to get positive solutions will add another exponential to the number of solutions. Each such combination corresponds to a most general unifier. This can be seen informally as follows: first of all note that every minimal solution in the monoid case is independent, since it cannot be obtained from other solutions. Now let θ_1 and θ_2 be two distinct combinations of these basis solutions that make all entries positive, and further assume that the unifier produced by θ_2 is an instance of that produced by θ_1. Clearly one cannot be a subset of another, because we do not have an identity. On the other hand, θ_2 cannot contain a basis solution that θ_1 does not, since this would violate the independence mentioned above.

Theorem 4: There are instances of aci-unifiable terms for which the number of unifiers in any complete, minimal set is doubly exponential.

References

Benanav, D., Kapur, D., and Narendran, P. (1985). Complexity of matching problems. In: Proc. of the first international conference on *Rewriting Techniques and Applications (RTA-85),* Dijon, France, LNCS 202, Springer Verlag. A revised version appeared in *J. of Symbolic Computation* 3 (1987) 203-216.

Boudet, A., Contejean, E., and Devie, H. (1990). A new AC unification algorithm with an algorithm for solving systems of diophantine equations. Proc. *5th Annual IEEE Symp. on Logic in Computer Science,* Philadelphia, 289-299.

Bürckert, H.-J., Herold, A., Kapur, D., Siekmann, J.H., Stickel, M., Tepp, M., and Zhang, H. (1988). Opening the AC-unification race. *J. of Automated Reasoning,* 4, 465-474.

Domenjoud, E. (1989). Number of Minimal Unifiers of the Equation $\alpha x_1 + \cdots + \alpha x_p =_{AC} \beta y_1 + \cdots + \beta y_q$. Unpublished report, CRIN, Nancy, France.

Fages, F. (1984). Associative-commutative unification. In: Proceedings of *7th Conference on Automated Deduction (CADE-7),* Napa Valley, California, LNCS 170, Springer Verlag. A revised version appeared in *J. of Symbolic Computation* 3(3) June 1987.

Fortenbacher, A. (1985). An algebraic approach to unification under associativity and commutativity. In: Proc. of the first international conference on *Rewriting Techniques and Applications (RTA-85),* Dijon, France, LNCS 202, Springer Verlag. A revised version appeared in *J. of Symbolic Computation* 3, June 1987.

Gathen, Joachim von zur, and Sieveking, M. (1978). A Bound on Solutions of Linear Integer Equalities and Inequalities. Proc. of the *American Mathematical Society* 72(1) 155-158.

Herold, A. and Siekmann, J. (1985). Unification in abelian semigroups. Memo SEKI-85-III-KL, Universitaet Kaiserslautern. A revised version appeared in *J. of Automated Reasoning,* 3, 3, 1987.

Huet, G. (1978). An algorithm to generate the basis of solutions of homogeneous linear Diophantine equations. *Information Processing Letters,* 7(3), 144-147.

Kapur, D. and Narendran, P. (1986a). NP-completeness of the set unification and matching problems. In: Proceedings of *8th Conference on Automated Deduction (CADE-8),* Oxford, U.K., LNCS 230, Springer Verlag.

Kapur, D., and Narendran, P. (1986b). Complexity of Unification Problems with Associative-Commutative Operators, Unpublished Manuscript, G.E. R&D Center, December 1986. A revised version to appear in *J. Automated Reasoning.*

Kapur, D. and Narendran, P. (1987). Matching, unification, and complexity. *SIGSAM Bulletin,* October 1987.

Kapur, D. and Zhang, H. (1991). A case study of the completion procedure: proving ring commutativity problems. In: *Computational Logic: Essays in Honor of Alan Robinson,* (Lassez and Plotkin, eds.), MIT Press, 360-394.

Kirchner, C. (1990). From unification in a combination of equational theories to a new AC-unification algorithm. In: *CREAS* (Nivat and Ait-Kaci, eds.), Academic Press.

Livesey, M., and Siekmann, J. (1976). Unification in sets and multisets. Memo SEKI-76-11, Universitaet Karlsruhe, Germany.

Martelli A., and Montanari, U. (1982). An efficient unification algorithm. *TOPLAS,* 4(2).

Papadimitriou, C.H. (1981). On the Complexity of Integer Programming. *J. of Assoc. of Comp. Mach.* 28, 765-768.

Paterson, M.S., and Wegman, M.N. (1978). Linear unification. *J. of Computer and System Sciences* 16, 158-167.

Schrijver, A. (1986). *Theory of Linear and Integer Programming.* John Wiley and Sons.

Stickel, M.E. (1975). A complete unification algorithm for associative-commutative functions. *4th International Joint Conf. on Artificial Intelligence,* Tbilisi, USSR.

Stickel, M.E. (1981). A unification algorithm for associative-commutative functions. *J. of Assoc. of Comp. Mach.* 28, 423-434.