01 Jul 1988

The Role of Term Symmetry in E-Unification and E-Completion

Blayne E. Mayfield

Ralph W. Wilkerson
Missouri University of Science and Technology, ralphw@mst.edu

THE ROLE OF TERM SYMMETRY IN E-UNIFICATION AND E-COMPLETION

B. E. Mayfield* and R. W. Wilkerson

CSc-88-6

Department of Computer Science

University of Missouri-Rolla

Rolla, Missouri 65401 (314)341-4491

*This report is substantially the Ph.D. dissertation of the first author, completed July 1988.

ABSTRACT

A major portion of the work and time involved in completing an incomplete set of reductions using an E-completion procedure such as the one described by Knuth and Bendix [*KB*70] or its extension to associative-commutative equational theories as described by Peterson and Stickel [*PS*81] is spent calculating critical pairs and subsequently testing them for coherence. A pruning technique which removes from consideration those critical pairs that represent redundant or superfluous information, either before, during, or after their calculation, can therefore make a marked difference in the run time and efficiency of an E-completion procedure to which it is applied.

The exploitation of *term symmetry* is one such pruning technique. The calculation of redundant critical pairs can be avoided by detecting the term symmetries that can occur between the subterms of the left-hand side of the major reduction being used, and later between the unifiers of these subterms with the left-hand side of the minor reduction. After calculation, and even after reduction to normal form, the observation of term symmetries can lead to significant savings.

The results in this paper were achieved through the development and use of a flexible E-unification algorithm which is currently written to process pairs of terms which may contain any combination of Null-E, C (Commutative), AC (Associative-Commutative) and ACI (Associative-Commutative with Identity) operators. One characteristic of this E-unification algorithm that we have not observed in any other to date is the ability to process a pair of terms which have different ACI top-level operators. In addition, the algorithm is a modular design which is a variation of the Yelick model [*Ye*85], and is easily extended to process terms containing operators of additional equational theories by simply "plugging in" a unification module for the new theory.

TABLE OF CONTENTS

LIST OF ILLUSTRATIONS

LIST OF TABLES

I. INTRODUCTION

A. STRUCTURE

In chapter 2, the definitions and notation throughout the remainder of the paper are presented. Additional definitions are provided as needed to supplement this list.

Chapters 3 and 4 are reviews of literature pertaining to unification and completion procedures, respectively. The history of unification is profiled beginning with Herbrand's work of the 1930's and continuing to the present. Included is a discussion of the extension of unification to E-unification, that is, the unification of terms containing operators that have properties described by a set of equations. Particular attention is given to the E-unification of terms containing associative-commutative (AC) or associative-commutative-with-identity (ACI) operators, which has become an area of high research interest with the advent of commercially available symbolic mathematics manipulators, such as MACSYMA, REDUCE, and MAPLE. The work performed by such products is done, in part, through the use of *complete sets of reductions*, that is, sets of rules for simplifying terms of an algebraic system such that the equality of those terms can be quickly decided. Chapter 4 contains an overview of procedures that can generate complete sets of reductions for some classes of algebraic systems, from the early and rather restrictive procedure developed by Knuth and Bendix to the much more general procedure of Jouannaud and Kirchner.

Chapters 5, 6, and 7 describe the software that we implemented for this research project. Included are pseudo-code and descriptions of our E-unification algorithm, E-completion procedure and, in chapter 6, algorithms for the detection and exploitation of the property of *term symmetry* between syntactic structures such as terms and sets of substitutions. This portion of the paper represents original work,

and proofs of correctness of the theory and of correctness and termination of the associated algorithms are presented. Chapter 7 describes the examples used to test the viability of applying the theory and the results of those tests.

Chapter 8 contains our conclusions and ideas for future research.

B. GOALS AND MOTIVATION

The Knuth-Bendix type of completion (or E-completion) procedure operates by calculating and processing all *critical pairs* of terms that can be formed from all pairs of reductions in the set to be completed. This combinatorial behavior is made even worse, because if one of the critical pairs cannot be simplified to an identity, then it is used to form a new reduction that is added to the set of reductions, and then the entire process begins again.

The goal of this research is to find some method to reduce the amount of processing needed to complete a set of reductions. Early work in this area by Lankford was later extended by Kapur, Musser, and Narendran. Their technique involves discarding those *superpositions* and *unifiers*, the building blocks of the critical pair, that are not in simplest form, with respect to the set of reductions. This has proven to yield significant savings in processing time. Our approach is based on the concept of *term symmetry*, a variable renaming isomorphism that can exist between terms, unifiers, and other syntactic structures. It is our goal to show that structures exhibiting term symmetry represent redundant information, and that these superfluous structures can be discarded without causing any adverse changes in the results of the E-completion procedure. This idea will be tested on several example cases, and the results will be presented and analyzed.

II. DEFINITIONS AND NOTATION

A. TERMS

V is a countably infinite set of *variables*. The members of V are designated by the names u, v, w, x, y, z, u_i, v_i, w_i, x_i, y_i, and z_i, for $0 \leq i$.

F is a finite set of *functions*, or *operators*. The members of F are designated by the names $+$, $-$, $\times$, $/$, f, g, h, f_i, g_i, and h_i, for $0 \leq i$. The *degree* of an operator f is the number of operands that it requires, and is written as $\mathbf{deg}(f)$. The set C of constants is the subset of F containing exactly those operators that have a degree of 0. That is, $C = \{f \mid f \in F \wedge \mathbf{deg}(f) = 0\}$. C is assumed to be non-empty, and its members are designated by the names 0, 1, a, b, c, d, e, a_i, b_i, c_i, d_i, and e_i, for $0 \leq i$.

The set of all *terms* constructed from members of V and F, written as $T(V,F)$, or simply T if no ambiguity arises, is defined recursively as follows:

(1) Variables are terms.

(2) Constants are terms.

(3) If $f \in F$, $\mathbf{deg}(f) = n$, and $t_1, \ \ldots, \ t_n$ are terms, then $f(t_1, \ \ldots, \ t_n)$ is a term.

(4) Only those syntactic structures defined by (1) through (3) are terms.

Terms may be represented as trees. The *domain* of a term t, written as $\mathbf{dom}(t)$, is the set of node occurrences in the tree, designated by dotted sequences of integers, following the notation of Huet and Oppen [*HO*80]. The empty sequence is designated as ε. The domain is recursively defined as follows:

(1) $(\forall x \in V)\ \mathbf{dom}(x) = \{\varepsilon\}$.

(2) $(\forall c \in C)\ \mathbf{dom}(c) = \{\varepsilon\}$.

(3) $(\forall f(t_1, \ \ldots, \ t_n) \in T)\ \mathbf{dom}(f(t_1, \ \ldots, \ t_n)) = \{\varepsilon\} \bigcup \{i.j \mid 1 \leq i \leq n \wedge j \in \mathbf{dom}(t_i)\}$.

The *subterm* of a term, t, at a position, or occurrence, $i \in \mathbf{dom}(t)$, is written as t/i. It follows that

(1) $t/\varepsilon = t$, and

(2) $f(t_1, \ldots, t_n)/i.j = t_i/j$.

The *strict domain* of a term, t, written as $\mathbf{sdom}(t)$, is the set of all non-variable occurrences in t. That is, $\mathbf{sdom}(t) = \{i \mid \mathbf{dom}(t) \wedge t/i \notin V\}$. The set of all variables occurring in t is written as $\mathbf{vars}(t)$.

B. SUBSTITUTIONS

A *set of substitutions* is a set of ordered pairs, each of which has the form $x \leftarrow t$, such that $x \in V$ and $t \in T$, and no variable occurs as the left-hand side of more than one pair. Sets of substitutions are designated by the names δ, θ, λ, σ, δ_i, θ_i, λ_i, and σ_i, for $0 \leq i$. The *left-variables* of a set of substitutions, θ, written as $\mathbf{lvars}(\theta)$, is the set containing the left-hand side of each member of θ. The *right-variables* of θ, written as $\mathbf{rvars}(\theta)$, is the set of variables occurring in the right-hand side of any member of θ. Stated another way, $\mathbf{lvars}(\theta) = \{x \mid x \leftarrow t \in \theta\}$ and $\mathbf{rvars}(\theta) = \{y \mid x \leftarrow t \in \theta \wedge y \in \mathbf{vars}(t)\}$.

A set of substitutions, θ, is *applied* to a term, t, written as $\theta(t)$, by simultaneously replacing each variable occurring in t, that also occurs in $\mathbf{lvars}(\theta)$, by the term paired with the variable in θ. This can be restated as follows:

(1) If $t = x$ and $x \leftarrow s \in \theta$, then $\theta(t) = s$.

(2) If $t = x$ and $x \leftarrow s \notin \theta$, then $\theta(t) = t$.

(3) If $t = f(t_1, \ldots, t_n)$, then $\theta(t) = f(\theta(t_1), \ldots, \theta(t_n))$.

Two sets of substitutions, θ_1 and θ_2, are *equivalent* if $(\forall x \in V)\ \theta_1(x) = \theta_2(x)$.

The *composition* of the sets of substitutions, λ and θ, written as $\lambda \circ \theta$, is a combination of the two sets such that

$$\lambda \circ \theta = \{x \leftarrow \lambda(t) \mid x \leftarrow t \in \theta\} \cup \{y \leftarrow s \mid y \leftarrow s \in \lambda \wedge y \notin \mathbf{lvars}(\theta)\}.$$

The application of a composition, $\lambda \circ \theta$, to a term, t, has the same effect as first applying θ to t, then applying λ to the result. That is, $\lambda \circ \theta(t) = \lambda(\theta(t))$.

A *variable-only set of substitutions* is a set of substitutions, $\{x \leftarrow t \mid t \in V\}$.

C. EQUATIONAL THEORIES

Let E be a set of equations, or axioms. The *equational theory* presented by E, written as E^*, is the finest congruence over T that contains E. That is, E^* is exactly the set of equations derivable from E by a finite proof, using reflexivity, symmetry, transitivity, and replacement of equals. The congruence relation on terms is written as $s =_E t$, where $s = t \in E^*$. F_E is the subset of F containing exactly those members of F described by E. For example, F_{AC} is the subset of F for which E contains an associative and a commutative axiom.

Nested occurrences within a term, t, of an operator, f, for which E contains an associativity axiom may be *flattened*, that is, f may be treated as an operator of arbitrary degree, and the nested occurrences of the operator and its associated parentheses be removed. For example, $f(x, f(y, z)) =_E f(f(x, y), z) =_E f(x, y, z)$.

D. FIRST-ORDER LOGIC

A *predicate* is a function that has as its range the set $\{TRUE, FALSE\}$. Predicates are designated by the names P, Q, R, P_i, Q_i, and R_i, for $0 \leq i$. The *degree* of a predicate, P, is written as $\mathbf{deg}(P)$.

A *literal* is defined as follows:

(1) If P is a predicate and $\mathbf{deg}(P) = 0$, then P is a literal.

(2) If P is a predicate, $\mathbf{deg}(P) = n$, and $t_1, \ldots, t_n$ are terms, then $P(t_1, \ldots, t_n)$ is a literal.

(3) If l is a literal, then its *negation*, $\neg l$, is also a literal, such that if $l = TRUE$, then $\neg l = FALSE$, and if $l = FALSE$, then $\neg l = TRUE$.

(4) Only those syntactic structures defined by (1) through (3) are literals.

A *clause* is a disjunction of literals. A *proposition* is a conjunction of clauses.

III. AN OVERVIEW OF UNIFICATION

Unification is a pattern matching process which identifies a match between all elements of a set of terms only if they can be made equal by substituting values (that are also terms) for variables occurring in them.

More formally stated, the unification problem is that of searching for a set of term-for-variable substitutions, θ, that, when applied to a set of terms, $S = \{s_1, \ldots, s_n\}$, reduces S to a singleton; that is, $\theta(s_1) = \theta(s_2) = \cdots = \theta(s_n)$. If such a set θ exists, it is called a *unifier* of S.

One of the areas in which unification has proven to be important is that of automated theorem proving. Early attempts to automate the theorem proving process were based upon the work of Herbrand; his proof method uses a form of unification on one class of propositions. However, in other cases, the process of unification is nothing more than an elaborate "generate-and-test" process, instantiating the variables of a proposition from progressively larger subsets of the Herbrand universe of the proposition. If the proposition is satisfiable, this process will eventually halt. However, if the proposition is not satisfiable, then the process will never terminate. Later efforts, based upon the work of Robinson, were much more successful due to the computationally effective unification algorithm that Robinson introduced.

Another area in which unification has shown itself to be a valuable tool is that of term rewriting systems (for example, symbolic mathematics packages such as MACSYMA and REDUCE). Term rewriting itself can be viewed as a very generalized form of unification, in which term-for-term substitutions are performed instead of term-for-variable substitutions. A good example of a *biological* term rewriting system is a human trigonometry student attempting a proof of an identity. The student begins with a pair of unlike terms and, through a series of term

rewritings on part (or all) of either or both of the terms, tries to derive a pair of identical terms. Automated term rewriting systems have been applied to problems in logic programming, programming language interpreters, and operating systems.

More recently, unification and other automated theorem proving tools have been applied to diagnostic expert systems. These tools give a firm mathematical foundation to the sometimes shaky experiential nature common to many expert systems. Hybrid systems, combining the best features of both rule-based and logic-based approaches to expert systems, are being investigated and developed.

A. HERBRAND'S UNIFICATION PROCEDURE

Many people attribute the "discovery" of unification to J. A. Robinson [*Ro*65]. However, the concept of unification predates Robinson's definition by at least thirty years.

In chapter 5 of his 1930 thesis at the University of Paris, Jacques Herbrand [*He*30] discusses the provable satisfiability of first-order predicate calculus propositions (this chapter is the source of Herbrand's theorem on the the satisfiability of propositions). In his paper, he states that he knows of no uniform procedure that would render the satisfiability of arbitrary propositions decidable, but he goes on to write

> "However, there is a class of propositions for which we have such a procedure, namely, the class of propositions such that the matrix of each is a disjunction of atomic propositions and of negations of atomic propositions."

Specifically, the procedure that Herbrand was writing about is one which can decide the satisfiability of a proposition which contains positive and negative occurrences of the same predicate symbol--that is, a proposition that includes a "sub-proposition" of the form

$$P(s_1, \ldots, s_n) \vee \neg P(t_1, \ldots, t_n),$$

such that P is a predicate symbol and $s_1, \ldots, s_n, t_1, \ldots, t_n$ are terms. This procedure is a search for instantiation values for the terms in the sub-proposition that will make the two predicates identical, except for their sign. If the search is successful, the sub-proposition is satisfiable and, thus, the original proposition (which is a disjunction of literals) is also satisfiable. Herbrand describes how to perform this search, which is a unification procedure.

However, as Herbrand pointed out, his unification procedure applies only to that class of propositions that contains both positive and negative occurrences of the same predicate symbol. For all other propositions, he took a brute force approach. An iterative process is begun, and with each pass, the variables of the proposition are instantiated from an increasingly larger subset of the Herbrand universe of the proposition. The *Herbrand universe* of a proposition is the set of all ground (variable-free) terms which can be formed from the function and constant symbols that occur in the proposition (if no constants occur, an arbitrary one is introduced). If any function symbols (other than constants) occur in the proposition, the Herbrand universe will contain an infinite number of terms. If the proposition is satisfiable, Herbrand's procedure will terminate. If the proposition is unsatisfiable, Herbrand's procedure will never terminate.

B. ROBINSON'S UNIFICATION ALGORITHM

In 1965, Robinson published a landmark paper [*Ro*65] in which he introduced *resolution* as the single inference rule needed to prove a set (conjunction) of clauses to be unsatisfiable, where each clause is a disjunction of literals. The resolution rule is very similar to modus ponens; in fact, modus ponens is an instance of resolution. Resolution infers a new clause called a *resolvent* from two other clauses in the following manner:

clause 1: $C \vee P(s_1, \ldots, s_n)$

clause 2: $\underline{C' \vee \neg P(t_1, \ldots, t_n)}$

resolvent: $\theta(C \vee C')$

where C and C' are (possibly empty) disjunctions of positive and/or negative literals, P is a predicate symbol, and θ is a unifier of $P(s_1, \ldots, s_n)$ and $P(t_1, \ldots, t_n)$.

A proof system that is based on resolution is a *refutational* system; that is, proofs are performed by contradiction. To use such a system, the clause to be proven is negated--that is, assumed to be false--and added to a (possibly empty) set of supporting clauses (axioms). Clause pairs are resolved until all possible clause pairs have been resolved or a contradiction is encountered. A contradiction occurs when two clauses of the form $P(s_1, \ldots, s_n)$ and $\neg P(t_1, \ldots, t_n)$ are resolved, producing the empty clause as a resolvent. Robinson proved that a resolution-based proof system will derive the empty clause if and only if the set of clauses being resolved embodies a contradiction.

Considering the combinatorial number of resolutions that can take place on a set of clauses in a resolution-based proof system, it is evident that unification is going to be called upon very frequently. Thus, it is important to make it as efficient as possible. Efficiency will be even more critical in E-unification (unification under an equational theory).

There can be an infinite number of different unifiers for a particular set of terms. However, Robinson proved that there is only one most general unifier for a set of terms, modulo variable renaming.

Definition 3.1: Let σ and θ be two unifiers of a set of terms S. If there exists a (possibly empty) set of substitutions, λ, such that

$\theta = \lambda \circ \sigma$,

then σ *is more general than* θ, written as $\theta \leq \sigma$. The unifier σ is called the *most general unifier (mgu)* of S if $\theta \leq \sigma$ for all unifiers, θ, of S. That is, any unifier of a set of terms can be obtained through the composition of some set of substitutions with the mgu.

```
ROBINSON-UNIFY(S);
begin
   σ0 := {};
   k := 0;
   Status := LOOP;
   while (Status = LOOP) do begin
      if (σk(S) is a singleton)
      then Status := SUCCESS;  /*σk is the mgu */
      else begin
         Dk := disagreement set of σk(S);
         sort Dk so that all variables appear first;
         Vk := first element of sorted Dk;
         Uk := second element of sorted Dk;
         if (Vk is a variable and does not occur in Uk)          (1)
         then begin
            σk+1 := {Vk ← Uk}∘σk                                 (2)
            k := k + 1;
         end
         else Status := FAIL;
      end;
   end;
   return(Status, σk);
end;
```

Figure 1. Robinson's unification algorithm

Figure 1 contains the pseudo-code for Robinson's unification algorithm. The algorithm attempts to unify a set of terms, S, returning either a unifier of the set, or failure if the set has no unifier. Subterms are unified iteratively in a left-to-right manner such that, unless failure has occurred, the set of substitutions, σ_k, calculated in the k^{th} iteration unifies some prefix of all terms in S. The set $\sigma_k(S)$ is the result of applying σ_k to each element of S. The *disagreement set*, D_k, of $\sigma_k(S)$ is the set consisting of the subterm of each term in $\sigma_k(S)$ at the leftmost position where not all of those subterms are identical; thus D_k represents the leftmost subterms that must

still be unified. In statement (2) of the pseudo-code, new substitution pairs are added to the set of substitution pairs by way of *composition*, as defined in chapter 2.

Robinson proved in his paper that the above procedure always terminates, is correct, and returns a unifier of the set of terms if and only if the set will unify. He also proved that the unifier returned by the algorithm is the mgu of the set of terms.

Example 3.1: Unify the set $S = \{f(a, x, g(h(y))), f(z, g(z), g(w))\}$ using Robinson's algorithm.

For k = 0:

$\sigma_0 = \{\}$

$\sigma_0(S) = \{f(a, x, g(h(y))), f(z, g(z), g(w))\}$

$D_0 = \{z, a\}$

For k = 1:

$\sigma_1 = \{z \leftarrow a\}$

$\sigma_1(S) = \{f(a, x, g(h(y))), f(a, g(a), g(w))\}$

$D_1 = \{x, g(a)\}$

For k = 2:

$\sigma_2 = \{x \leftarrow g(a)\} \circ \{z \leftarrow a\} = \{z \leftarrow a, x \leftarrow g(a)\}$

$\sigma_2(S) = \{f(a, g(a), g(h(y))), f(a, g(a), g(w))\}$

$D_2 = \{w, h(y)\}$

For k = 3:

$\sigma_3 = \{w \leftarrow h(y)\} \circ \{z \leftarrow a, x \leftarrow g(a)\} = \{z \leftarrow a, x \leftarrow g(a), w \leftarrow h(y)\}$

$\sigma_3(S) = \{f(a, g(a), g(h(y)))\}$

Thus, the mgu of S is $\sigma_3 = \{z \leftarrow a, x \leftarrow g(a), w \leftarrow h(y)\}$. Note the left-to-right manner in which the terms are unified, as discussed earlier. It can also be observed that all of the terms in $\sigma_k(S)$ are identical to the left of the elements of D_k.

At the point in Robinson's algorithm when a new substitution pair is being calculated, a check is made to see if the variable to be replaced, V_k, occurs within the term which is to replace it, U_k. (See statement (1) in figure 1) If so, the algorithm halts immediately with failure. This operation is called the *occurs check*, and its presence is necessary to make Robinson's unification a sound procedure. The soundness of unification will be further explained in the discussion of PROLOG, below. The occurs check gives the algorithm a worst-case complexity which is exponential based on the size of the terms being unified. This exponential behavior can be illustrated by a simple example.

Example 3.2: Unify the set $S = \{f(g(x_0, x_0), x_2, g(x_2, x_2)), f(x_1, g(x_1, x_1), x_3)\}$.

For k = 0:

$\sigma_0 = \{\}$

$\sigma_0(S) = \{f(g(x_0, x_0), x_2, g(x_2, x_2)), f(x_1, g(x_1, x_1), x_3)\}$

$D_0 = \{x_1, g(x_0, x_0)\}$

For k = 1:

$\sigma_1 = \{x_1 \leftarrow g(x_0, x_0)\}$

$\sigma_1(S) = \{f(g(x_0, x_0), x_2, g(x_2, x_2)), f(g(x_0, x_0), g(g(x_0, x_0), g(x_0, x_0)), x_3)\}$

$D_1 = \{x_2, g(g(x_0, x_0), g(x_0, x_0))\}$

For k = 2:

$\sigma_2 = \{x_1 \leftarrow g(x_0, x_0), x_2 \leftarrow g(g(x_0, x_0), g(x_0, x_0))\}$

$\sigma_2(S) = \{f(g(x_0, x_0), g(g(x_0, x_0), g(x_0, x_0)), g(g(g(x_0, x_0), g(x_0, x_0)), g(g(x_0, x_0), g(x_0, x_0)))),$

$f(g(x_0, x_0), g(g(x_0, x_0), g(x_0, x_0)), x_3)\}$

$D_2 = \{x_3, g(g(g(x_0, x_0), g(x_0, x_0)), g(g(x_0, x_0), g(x_0, x_0)))\}$

For k = 3:

$\sigma_3 = \{x_1 \leftarrow g(x_0, x_0), x_2 \leftarrow g(g(x_0, x_0), g(x_0, x_0)),$

$x_3 \leftarrow g(g(g(x_0, x_0), g(x_0, x_0)), g(g(x_0, x_0), g(x_0, x_0)))\}$

$\sigma_3(S) = \{f(g(x_0, x_0), g(g(x_0, x_0), g(x_0, x_0)), g(g(g(x_0, x_0), g(x_0, x_0)), g(g(x_0, x_0), g(x_0, x_0))))\}$

Note that the mgu, σ_3, of the terms in S contains 2^i occurrences of the variable x_0 in the term that is to replace each variable x_i, for $1 \leq i \leq 3$.

C. IMPROVEMENTS ON THE EFFICIENCY OF ROBINSON'S ALGORITHM

Since unification is such an important and frequently used component of applications such as automated theorem provers and term rewriting systems, the exponential nature of Robinson's unification algorithm prompted a great deal of research into methods of improving its efficiency or replacing it with some other, faster unification algorithm. We now review some of these efforts.

1. PROLOG.

The statements of a program written in PROLOG (PROgramming in LOGic) are actually first-order predicate logic clauses. Specifically, they are *Horn clauses*--disjunctions of literals with, at most, one positive literal. The program itself is a collection of *definite clauses*--clauses with exactly one positive literal. Execution of a PROLOG program is a series of resolution steps, and begins by resolving a distinguished *goal clause*--a clause with no positive literals--with one of the definite clauses. Each successful resolution produces a new goal clause which is then used as a parent clause along with one of the definite clauses in the next resolution step. Execution continues until a resolution yields a resolvent null clause (successful completion of the program), or the goal clause cannot be successfully be resolved with any definite clause (failure).

Early in its development, the designers of PROLOG realized that an exponential unification algorithm would render PROLOG useless for any sizable applications; unification is a basic operation in PROLOG that is generally invoked many times for each successful unification. A solution had to be found, and one was. The designers

chose to completely omit the occurs check! The reason is that unification without occurs check is linear on the size of the smallest term being unified. However, the speedup comes with a price; without the occurs check, unification is an unsound procedure. That is, without the occurs check, it is possible to generate a unifier in which a particular variable appears on both sides of one substitution pair. This may cause PROLOG to go into an infinite loop or, even worse, to return answers that are wrong.

Example 3.3: Consider the two-clause PROLOG program

lt(X, $X+1$).

lt(3, 2) :– lt($Y+1$, Y).

The first clause of this program can be read as "X is less than $X+1$." The second clause can be read as "3 is less than 2 if $Y+1$ is less than Y." Both of these are true statements. To begin program execution the goal clause

? – lt(3, 2).

is introduced. The goal clause can be read as "is 3 less than 2?" The execution of the program proceeds as follows:

(1) Unification is attempted between the goal clause and the first program clause--unification fails.

(2) Unification is attempted between the goal clause and the second program clause--unification succeeds with a mgu {}.

(3) The resolvent goal clause "? – lt($Y+1$, Y)", is produced, which can be read as "is $Y+1$ less than Y?"

(4) Unification is attempted between the new goal clause and the first program clause.

It is in step (4) of the execution of the logic program that things begin to go awry. During the iteration for $k=1$ in Robinson's algorithm, $\sigma_1 = \{X \leftarrow Y+1\}$, and $D_1 = \{Y, Y+1+1\}$. Clearly, an occurs check on D_1 reveals that the variable Y

occurs in the term $Y+1+1$, so the unification should halt with failure. However, since the unification algorithm being used is devoid of an occurs check, it will not fail, but will instead add the substitution $Y \leftarrow Y+1+1$ to the partial unifier. What happens at this point depends on the particular implementation of PROLOG being used. Some versions, such as Micro-PROLOG for the IBM PC, will go into an infinite loop trying to replace all occurrences of the variable Y with the term Y+1+1. Others, such as Quintus PROLOG running under the VMS operating system on a Micro-VAX II will just return the answer "YES", which is obviously wrong.

Thus, PROLOG is a language with a message, and that message is "user beware!" It is left entirely to the programmer to avoid situations that would cause problems such as that cited above.

2. The Paterson-Wegman Algorithm.

Many of the successful attempts to improve or replace Robinson's unification algorithm have been aimed at modifying the data structures representing the terms to be unified. One such effort is the algorithm of Paterson and Wegman [*PW*78]. Their algorithm unifies a pair of terms with a space and time complexity which is linear based on the size of the terms to be unified.

In order to use the Paterson-Wegman algorithm, the terms to be unified must be expressed as a *directed acyclic graph (dag)* in which common subexpressions are represented by a single subgraph. Nodes labelled by an n-ary function name will have an outdegree of n (thus nodes labelled by constants name will have an outdegree of 0). Nodes labelled by a variable name will have an outdegree of 0. Nodes with an indegree of 0 are roots. Figure 2 depicts the dag representation for the pair of terms, $f(g(x_1), g(x_2))$ and $f(x_2, x_3)$.

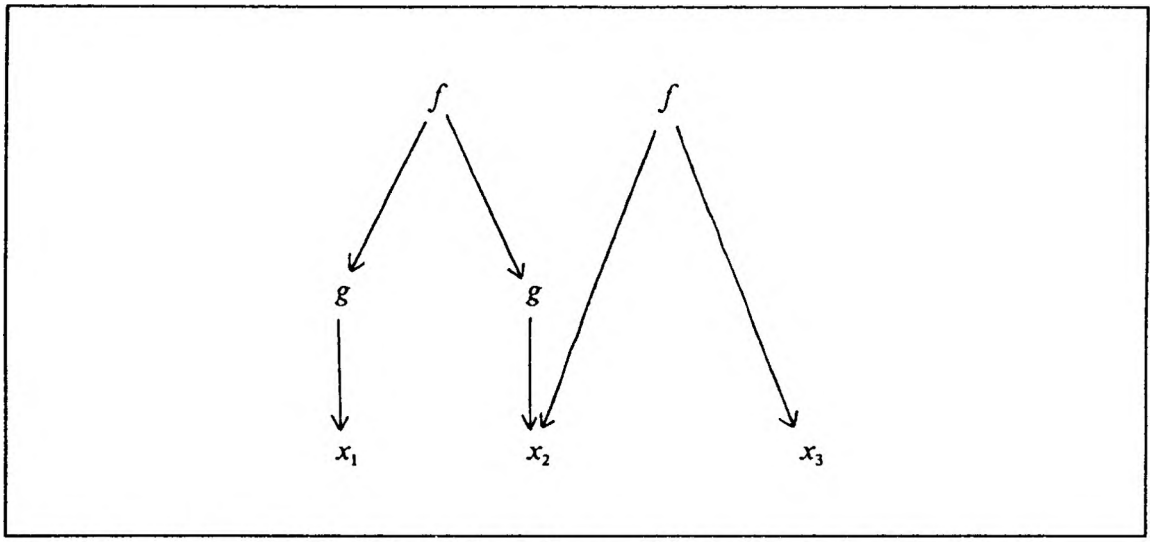


Figure 2. An example of directed acyclic graph representation.

Once the pair of terms has been transformed into a dag, the input to the Paterson-Wegman algorithm is a set consisting of the root nodes of the two terms. This set is actually an equivalence class, since for the two terms to be unifiable, the roots must be unifiable. The algorithm proceeds in a top-down manner through the dag, working only with one equivalence class of root nodes at a time. When the nodes in a root class have been processed, they are removed from the dag, along with the edges leading from them. This exposes new root nodes which are then divided into equivalence classes. When all nodes have been removed from the dag, the pair of terms has been unified. Because of the data representation used, no occurs check is needed; an occurs check situation will manifest itself as a cycle in the graph, and the algorithm will fail since the nodes in the cycle can never become a root node, and will never be processed and removed from the dag.

3. The Martelli-Montanari Algorithm.

In 1982, Martelli and Montanari published a paper in which they outlined an "almost-linear" algorithm for the unification of a pair of terms [*MM*82]. Like Paterson and Wegman, Martelli and Montanari approached the efficiency problem of unification by changing the data structure used to represent terms. A pair of terms, s and t, to be unified by the Martelli-Montanari algorithm are represented as a singleton set, $S = \{s = t\}$, of simultaneous equations; unification is then reduced to the problem of solving this set of simultaneous equations. The set of equations expands and contracts according to the application of two transformations described by Martelli and Montanari:

(1) Let $f(t_1, \ldots, t_n) = f(u_1, \ldots, u_n) \in S$. *Term reduction* is the process of replacing this equation in S by the equations $t_1 = u_1, \ldots, t_n = u_n$. If f is a constant (i.e. if $n = 0$), simply delete the equation from S.

(2) Let $x = t \in S$, such that x is a variable and t is a term. Replace all occurrences of x in all other equations of S by t.

Martelli and Montanari claim that their algorithm, when implemented with sets of variables represented as lists, has a complexity of O($n$log n), where n is the number of distinct variables in the pair of terms. They also claim that, when implemented with sets of variables represented as trees and when using the UNION-FIND algorithm [*AH*74] to add and to access elements that the complexity drops to O(mG(m)), where G(m) is the inverse of Ackermann's function[1] and m is the number of variable occurrences in the pair of terms. Thus, the Martelli-Montanari unification algorithm is indeed almost linear, and uses more "standard" data structures than that

---

[1]Ackermann's function is defined by:
$F(0) = 1$,
$F(i) = 2^{F(i-1)}$.
Thus, F(0) = 1, F(1) = 2, F(2) = 4, F(3) = 16, F(4) = 65536, etc.

found in the Paterson-Wegman algorithm. The reason that the Martelli-Montanari algorithm is mentioned here, even though its complexity is theoretically worse than that of the Paterson-Wegman algorithm, is that Martelli and Montanari claim that, when actually implemented, their algorithm usually outperforms that of Paterson and Wegman.

4. The Linear Nature of Unification.

It has been shown by Dwork, Kanellakis, and Mitchell [*DK*84] that unification is an inherently linear process, that is, even when run in a parallel environment, the best results that can be achieved are log space and linear time complexities.

With a lower bound defined on the complexities of unification, and the existence of algorithms that are at or near that complexity level, other methods have been investigated for increasing the speed of unification. These include the integration of unification algorithms into the microcode of computers [*Ca*85] and the design of a parallel unification integrated circuit chip [*YB*86].

D. TERM MATCHING

A useful subset of the unification problem is the term matching problem. The *term matching* problem for a pair of terms, s and t, is a search for a set of substitutions, θ, such that

$$\theta(s) = t,$$

that is, a set of substitutions which, when applied to just one of the terms being matched, transforms it into a term identical to the second term. Such a set θ is called a *match* of s and t. A match is a unifier, but a unifier is not always a match. Term matching, or rather, its extension to E-matching, is used extensively in term rewriting applications.

E. E-UNIFICATION

Recall from the definition of unification in the introduction to this chapter that the elements of a set of terms unify only if they can be made equal by substituting values for variable occurrences in those terms. Equality has been interpreted up to this point as meaning *identity*. Now, however, the definition of unification will be broadened by extending the definition of term equality.

E-unification is the search for a set of substitutions, θ, which, when applied to each member of a set of terms, $S = \{s_1, \ldots, s_n\}$, makes the elements of S *provably equal* under some equational theory, E, that is, $\theta(s_1) \underset{E}{=} \theta(s_2) \underset{E}{=} \ldots \underset{E}{=} \theta(s_n)$.
The set of axioms describing E could be the empty set, in which case E-unification is exactly that performed by Robinson's unification algorithm; from this point, we shall call this *null-E unification.*

There are many situations in which the ability to operate under a non-empty equational theory is useful. For example, a software system designed to solve problems in symbolic mathematics needs to have the ability to recognize and process operators which exhibit the associativity, commutativity, identity, and/or idempotency properties. Resolution-based proof systems and term rewriting systems designed around a null-E unification algorithm can still be forced to deal with non-empty equational theories, but not without introducing new problems. The obvious way is to include the axioms describing the equational theory as part of the set of terms input to the system. However, such systems may already be taxed by the combinatorics of the pairwise processing of clauses or terms, and must now deal with an even larger set of clauses. In addition, this solution tends to make such system "wander"; that is, many trivial and unnecessary intermediate results may be generated (since the solution search space has increased in size). An even worse consequence is that certain axioms, such as the commutativity axiom, can cause systems to go into

an infinite loop (more detail is given about these problems with respect to term rewriting systems in chapter 4).

An alternative solution is to build all or some of the axioms describing the equational theory into the unification algorithm rather than including them in the set of input axioms. The inclusion of an E-unification algorithm reduces the combinatorics overhead of processing input axioms, thereby resulting in a more focused search of the solution space of a problem. In addition the potential looping behavior associated with certain "troublesome" axioms of the equational theory is avoided. However, this solution, too, is not without its problems.

The major drawback of E-unification is that a different unification algorithm will be needed for each equational theory. This entails a change in program code whenever another equational theory is to be used. The use of a null-E unification algorithm merely requires a change to the set of input axioms in order to change equational theories. Some progress has been made in the creation of a "general" E-unification algorithm for certain classes of equational theories, but there is still no solution for the general case. Another problem is that there are some equational theories which can be described as axioms, but for which there exists no unification procedure (since E-unification is equivalent to the decision procedure for equivalence of terms under an equational theory).

1. Early work in E-unification.

One of the earliest researchers of E-unification was Plotkin [*Pl72*]. He investigated many of the advantages and problems of developing unification algorithms for various equational theories. Most of his work deals with resolution-based proof systems. It was he who first showed that to guarantee the

completeness of a resolution-based proof system using E-unification, the set, Σ, of unifiers calculated for a set of terms, S, must exhibit two properties:

(1) Correctness: All $\sigma \in \Sigma$ must unify S.

(2) Completeness: If θ unifies S, then there exists a $\sigma \in \Sigma$ and a λ such that $\theta = \lambda \circ \sigma$.

Plotkin also described an additional property that is desirable for efficiency reasons, but which is not necessary for the completeness of a resolution-based proof system:

(3) Minimality: If θ and σ are both members of Σ, then there is no λ such that $\theta = \lambda \circ \sigma$.

A set of unifiers for a pair of terms, s and t, that has the properties (1) and (2) described above is said to be a *complete set of unifiers of s and t*, written as **csu**(s, t). If, in addition, the set of unifiers has property (3), it is called a *minimal complete set of unifiers*, written as μ**csu**(s, t).

2. Unitary, Finitary, and Infinitary Complete Sets of Unifiers.

Plotkin categorized equational theories for which unification is decidable into four classes, based upon the maximum cardinality of their minimal complete set of unifiers: unitary, finitary, infinitary, and nullary. A *unitary* theory is one for which the minimal complete sets of unifiers can contain no more than one member. The empty equational theory (that is, null-E) is in this category. Robinson proved that a set of null-E terms will have, at most, a single mgu, modulo variable renaming by composition.

A *finitary* equational theory is one for which a minimal complete set of unifiers may contain more than one , but a finite number of *maximally general unifiers*, that is, unifiers which are mutually most general. A commutative equational theory, whose operators are described by a set of axioms of the form

$f(x, y) = f(y, x)$,

is a finitary theory. The existence of multiple unifiers for a set of terms will increase the complexity of a solution search space.

Example 3.4: Let f be an commutative (C) operator, and let $s = f(x, y, z)$ and $t = f(a, b, c)$ be terms. Then $\mu\mathbf{csu}(s, t) = \{\{x \leftarrow a, y \leftarrow b, z \leftarrow c\}, \{x \leftarrow a, y \leftarrow c, z \leftarrow b\}, \{x \leftarrow b, y \leftarrow a, z \leftarrow c\}, \{x \leftarrow b, y \leftarrow c, z \leftarrow a\}, \{x \leftarrow c, y \leftarrow a, z \leftarrow b\}, \{x \leftarrow c, y \leftarrow b, z \leftarrow a\}\}$.

An *Infinitary* equational theory is one that may have an infinite number of maximally general unifiers. One such theory is an associative equational theory, whose operators are described by a set of axioms of the form

$f(f(x, y), z) = f(x, f(y, z))$.

Infinitary equational theories re-introduce a problem that existed in Herbrand's unification, namely, the possible non-termination of the corresponding E-unification procedure. One can either calculate the complete set of unifiers (in which case the unification procedure may never halt) or calculate a finite, but incomplete set of unifiers. Neither of these choices is an attractive one.

Example 3.5: Let f be an associative (A) operator, and let $s = f(a, x)$ and $t = f(x, a)$, be terms. Then $\mu\mathbf{csu}(s, t)$ is the infinite set $\{\{x \leftarrow a\}, \{x \leftarrow f(a, a)\}, \{x \leftarrow f(a, a, a)\}, \ldots\}$.

The class of *nullary* equational theories is the strangest of the four types. A set of terms under a nullary theory may have a unifier, but a minimal complete set of unifiers for the terms will never exist! This is true because if a set of terms, $S = \{s_1, \ldots, s_n\}$, will unify under a nullary theory, there may be an infinite chain of unifiers,

$\sigma_1 \leq \sigma_2 \leq \sigma_3 \leq \ldots$&,

such that each σ_{i+1} is more general than σ_i, for $1 \leq i$. Plotkin wrote this about Γ, his notation for a minimal complete set of unifiers:

> "We also know of no example of a theory T ... for which there is no such Γ, although we expect that one exists."

Manfred Schmidt-Schauss [*SS*86] did, however, find an example of a first-order equational theory which he proves to be of type nullary. It is an equational theory whose operators have the properties of associativity and idempotency, that is, an AId theory:

$f(f(x, y), z) = f(x, f(y, z))$ and (associativity)

$f(x, x) = x$. (idempotency)

3. AC Unification.

One class of E-unification algorithms that has received much attention over the past few years is that class designed to unify terms using equational theories consisting of axioms of associativity and commutativity for a set of operators. This is due mainly to the application of resolution-based automated theorem provers to mathematical problems, and also to the commercialization of several term-rewriting-based symbolic mathematics packages (such as MACSYMA and REDUCE). Some of the most commonly required unification algorithms are for associative-commutative (AC), associative-commutative with identity (ACI), and associative-commutative with idempotency (ACId) theories. The topic of this section will be that of AC unification.

a. The Diophantine Process.

All AC unification algorithms that have been developed to date exploit one common factor: A pair of terms involving only one AC operator and any number of variables can be associated with a linear diophantine equation, and the non-negative

integral solutions to that equation correspond to the unifiers of the terms. In order to gain an understanding of AC unification, which is also the foundation for ACI and ACId unification, we shall explore this relationship more closely.

An AC term which consists of one operator and any number of variables is said to be a *variable-only AC term.* In order to make the connection between the pair of variable-only AC terms and the diophantine equation more apparent, it will also be required that the terms be flattened. A *flattened* AC term is one in which all nested levels of associativity have been removed, treating the AC operator as one with an arbitrary number of operands.

Example 3.6: Let f be an AC operator. Then

$$s = f(f(u, f(v, w)), f(x, f(f(y, x), v)))$$

is a variable-only AC term, and

$$s' = f(u, v, w, x, y, x, v)$$

is the flattened form of s.

The following example illustrates the transformation of a pair of flattened, variable-only terms into its corresponding diophantine equation. by example.

Example 3.7: Let f be an AC operator and let $s = f(y, x, x, y, x)$ and $t = f(u, v, v)$ be a pair of flattened, variable-only AC terms. The diophantine equation corresponding to the unordered pair of terms, $< s, t >$, is

$$3x + 2y = u + 2v.$$

It can be observed that each of the AC terms maps to one side of the diophantine equation. Each side of the equation is a sum of products, where each product is composed of a distinct variable from the term associated with the side and a coefficient that is equal to the multiplicity of that variable in the term.

A solution to the diophantine equation is a set of number-for-variable substitutions that makes the two sides of the equation equivalent. In a like manner, a unifier of the pair of AC terms is a set of term-for-variable substitutions that make the two terms provably equal. It can be shown that there is a correspondence between the non-negative integral solutions of a diophantine equation and its associated pair of flattened, variable-only AC terms. The solutions sought must be non-negative and integral because each variable in the AC terms can only be replaced by a non-negative and integral number of term occurrences. That is, one cannot replace a variable by negative or a fractional number of term occurrences.

There are an infinite number of non-negative integral solutions to a diophantine equation. However, each of these solutions can be represented as a sum of members of a finite *basis set* of solutions to the equation. A basis set can be algorithmically constructed by generating solutions for the equation in ascending value order. As each solution is generated, it is checked to see if it is equal to a sum of solutions already in the basis. If so, it is discarded; otherwise, it is added to the basis set. This generation process continues until some predetermined limit is reached for the value of the equation. The only requirement on the size of this limit is that it must be large enough that all solutions that are part of the basis are generated before it is reached. However, it is desirable to make the limit as low as possible, so that the basis generation process runs as quickly as possible. Several authors have described methods to calculate this limit, including Huet [*Hu*78] and Lankford [*La*87]. Zhang [*Zh*87] describes a method of basis generation which works more efficiently for a diophantine equation in which many coefficients have a value of 1.

Table I contains the basis set for the diophantine equation of example 3.7, above. The basis set was calculated using Huet's limiting factor. The column labelled "Introduced Variable" is for use in the discussion of Stickel's AC unification algorithm.

Table I. THE BASIS SET FOR THE DIOPHANTINE EQUATION OF EXAMPLE 3.7.

Solution Vector	x	y	u	v	Solution Value	Introduced Variable
b_1	0	1	0	1	2	z_1
b_2	0	1	2	0	2	z_2
b_3	1	0	1	1	3	z_3
b_4	1	0	3	0	3	z_4
b_5	2	0	0	3	6	z_5

b. The Restricted Stickel AC Unification Algorithm.

Very similar unification algorithms for terms containing AC operators have been developed by Stickel [*St*75], and by Livesey and Siekmann [*LS*76]. Because of their similarity, only the Stickel algorithm will be described in this paper, because it is the one used in the implementation developed for use in this research.

Stickel's restricted AC algorithm is one which unifies a pair of flattened, variable-only AC terms. It is designed around the diophantine equation solution process described above. Once the basis set of non-negative integral solutions has been determined for the diophantine equation associated with the pair of terms, each solution is associated with an *introduced variable* (that is, a variable not appearing in either of the AC terms being unified). This can be seen for the diophantine equation of example 3.7 in the last column of table I. As stated earlier, each non-negative integral solution to the equation can be expressed as a sum of members of the basis set. Thus, if the set is the basis from example 3.7, $\{b_1, b_2, b_3, b_4, b_5\}$, then each solution of the equation will be of the form

$$z_1b_1 + z_2b_2 + z_3b_3 + z_4b_4 + z_5b_5,$$

where each coefficient, z_1, z_2, z_3, z_4, and z_5, is a non-negative integer. Then, any solution to the equation will have as solutions for its individual variables,

$x = z_3 + z_4 + 2z_5$,

$y = z_1 + z_2$,

$u = 2z_2 + z_3 + 3z_4$, and

$v = z_1 + z_3 + 3z_5$.

These values are obtained by reading down the column for each variable in table I. This generalized form for a solution to the equation corresponds to a *general unifier* of the AC terms $f(y, x, x, y, x)$ and $f(u, v, v)$:

$\{x \leftarrow f(z_3, z_4, z_5, z_5), y \leftarrow f(z_1, z_2), u \leftarrow f(z_2, z_2, z_3, z_4, z_4, z_4), v \leftarrow f(z_1, z_3, z_5, z_5, z_5)\}$.

However, not all non-negative integral solutions correspond to a valid unifier. Solutions in which some combination of the introduced variables are set to zero correspond to AC unifiers in which those same variables have been replaced by the identity, or null, term. If this causes one of the original term variables, x, y, u, or v, in the example, to be set to the identity term, then that set of substitutions is not a valid unifier of the AC terms, since identity is not one of the properties of the equational theory.

Thus, the generalized unifier form presented above is not sufficient. In addition, it must be combined with each member of the power set of $\{z_1 = 0, \ldots, z_k = 0\}$ and each combination must be examined in order to determine which correspond to valid unifiers and which do not. For the pair of terms in example 3.7, this means that there are 2^5 or 32 possible unifiers, of which 19 prove to be valid unifiers; these are listed in figure 3.

Pseudo-code for the E-unification algorithm developed by Stickel for variable-only AC terms is presented in figure 4. The symbol ϕ in statement (5) represents an identity term. In statements (1) and (2) the input terms are flattened. In statement (3) the operands common to both flattened terms are removed before

$\{x \leftarrow f(z_3, z_4, z_5, z_5), y \leftarrow f(z_1, z_2), u \leftarrow f(z_2, z_2, z_3, z_4, z_4, z_4), v \leftarrow f(z_1, z_3, z_5, z_5, z_5)\}$
$\{x \leftarrow f(z_3, z_4, z_5, z_5), u \leftarrow f(y, y, z_3, z_4, z_4, z_4), v \leftarrow f(z_3, z_5, z_5, z_5)\}$
$\{x \leftarrow f(z_3, z_4, z_5, z_5), u \leftarrow f(z_3, z_4, z_4, z_4), v \leftarrow f(y, z_3, z_5, z_5, z_5)\}$
$\{x \leftarrow f(z_4, z_5, z_5), y \leftarrow f(z_1, z_2), u \leftarrow f(z_2, z_2, z_4, z_4, z_4), v \leftarrow f(z_1, z_5, z_5, z_5)\}$
$\{x \leftarrow f(z_3, z_5, z_5), y \leftarrow f(z_1, z_2), u \leftarrow f(z_2, z_2, z_3), v \leftarrow f(z_1, z_3, z_5, z_5, z_5)\}$
$\{x \leftarrow f(z_3, z_4), y \leftarrow f(z_1, z_2), u \leftarrow f(z_2, z_2, z_3, z_4, z_4, z_4), v \leftarrow f(z_1, z_3)\}$
$\{x \leftarrow f(z_4, z_5, z_5), y \leftarrow f(z_2), u \leftarrow f(y, y, z_4, z_4, z_4), v \leftarrow f(z_5, z_5, z_5)\}$
$\{x \leftarrow f(z_3, z_5, z_5), u \leftarrow f(y, y, z_3), v \leftarrow f(z_3, z_5, z_5, z_5)\}$
$\{x \leftarrow f(v, z_4), u \leftarrow f(y, y, v, z_4, z_4, z_4)\}$
$\{x \leftarrow f(z_4, z_5, z_5), u \leftarrow f(z_4, z_4, z_4), v \leftarrow f(y, z_5, z_5, z_5)\}$
$\{x \leftarrow f(u, z_5, z_5), v \leftarrow f(y, u, z_5, z_5, z_5)\}$
$\{x \leftarrow f(z_3, z_4), u \leftarrow f(z_3, z_4, z_4, z_4), v \leftarrow f(y, z_3, z_5, z_5, z_5)\}$
$\{x \leftarrow f(z_5, z_5), y \leftarrow f(z_1, z_2), u \leftarrow f(z_2, z_2), v \leftarrow f(z_1, z_5, z_5, z_5)\}$
$\{y \leftarrow f(z_1, z_2), u \leftarrow f(z_2, z_2, x, x, x), v \leftarrow f(z_1, z_5, z_5, z_5)\}$
$\{x \leftarrow f(z_3), y \leftarrow f(z_1, z_2), u \leftarrow f(z_2, z_2, z_3), v \leftarrow f(z_1, z_3)\}$
$\{x \leftarrow f(z_5, z_5), u \leftarrow f(y, y), v \leftarrow f(z_5, z_5, z_5)\}$
$\{u \leftarrow f(y, y, x), v \leftarrow x\}$
$\{u \leftarrow f(x, x, x), v \leftarrow y\}$
$\{u \leftarrow x, v \leftarrow f(y, x)\}$

Figure 3. The AC unifiers for the term pair of example 3.7.

the diophantine equation is generated. It is easy to see that this does not change the solutions to the problem, since this corresponds to subtracting an identical quantity from both sides of the diophantine equation. Common operands are removed to make the solution process more efficient; fewer products on each side of the diophantine equation means fewer solutions that need to be examined to calculate the basis set of solutions. In statement (4) of the pseudo-code, a call is made to a function that solves the diophantine equation for its basis set.

Stickel gives a proof of the correctness and completeness of his restricted AC unification algorithm by proving that the diophantine process is correct and complete, that is, that the set of solutions to the diophantine equation is exactly the set which can be produced from the basis set of solutions.

```
AC-UNIFY-VO(Term1, Term2);
begin
   if Term1.root = Term2.root
   then begin
      NewTerm1 := FLATTEN(Term1);                                    (1)
      NewTerm2 := FLATTEN(Term2);                                    (2)
      remove arguments common to NewTerm1 and NewTerm2;              (3)
      Equation := diophantine equation created from
                  NewTerm1 and NewTerm2;
      Basis := basis solution set for Equation;                      (4)
      BaseUnifier := EmptySet;
      for i := 1 to |Basis|
         Unifier := Unifier + v_i ← BasisTerm_i,
                    where v_i is the i^th leftmost variable in Equation,
                    and BasisTerm_i is the i^th column of Basis;
      UnifierSet := EmptySet;
      for σ ∈ the power set of {z_1 ← φ, ... , z_|Basis| ← φ} begin   (5)
         Unifier := σ∘BaseUnifier;
         If (Unifier is valid)
         then UnifierSet := UnifierSet + Unifier;
      end;
      return(UnifierSet);
   end
   else
      /* Term1 and Term2 have different AC operators and do not unify */
      return(EmptySet);
end;


Notes:
   FLATTEN(Term) returns the flattened form of Term.
```

Figure 4. Stickel's AC unification algorithm for variable-only terms

c. The Generalized Stickel Algorithm.

Stickel's variable-only AC unification algorithm is certainly interesting, but is of limited utility: Most AC terms in a real application will have an outer operator that is AC, but the arguments will be terms of different AC operators and/or non-AC operators. This is exactly the universe in which Stickel's generalized AC unification algorithm is designed to operate. It assumes, however, the existence of a finite and

complete E-unification algorithm for each non-AC equational theory to be represented.

The core of the generalized algorithm is an idea called variable abstraction. *Variable abstraction* is the process of uniformly replacing each operand of a term by a new variable (one that does not appear in the term), and forming a set of substitutions, called an *abstraction set* , in which each pair consists of one of the new variables and the operand that it replaces in the original term.

Example 3.8: Let f and g be AC operators, and h be a null-E operator. The variable abstraction of the term $s = f(g(w, a), b, h(x), y)$ is a new term,

$$s' = f(x_1, x_2, x_3, y),$$

and its abstraction set is

$$\{x_1 \leftarrow g(w, a), x_2 \leftarrow b, x_3 \leftarrow h(x)\}.$$

The original term can be obtained by applying the abstraction set to the variable abstraction of the term. Thus, the variable abstraction is a generalization of the original term. The original term should be flattened before it is abstracted (that is, flattened with respect to the outer AC operator of the term). Thus, the variable abstractions of a pair of AC terms will be a pair of flattened, variable-only AC terms, which can then be unified using Stickel's restricted unification algorithm.

However, there is another step to complete the generalized AC unification algorithm. Each unifier of the two abstracted terms must then be unified with the abstraction set, for the latter represents a set of constraints on the values that the new variables may take on in each unifier. This means that in order for a unifier of the variable abstractions to lead to one or a set of unifiers of the original AC terms, the values assigned to each new variable in the abstraction set must unify with their respective assigned values in the unifiers. Each such set of substitutions that

simultaneously unifies the pair of abstracted terms and the value pair for each new variable is thus a unifier of the original AC terms.

The pseudo-code for Stickel's generalized AC unification algorithm is presented in figure 5. In statements (1) and (2) the input terms are flattened, then abstracted. In statements (3) and (4), recursive calls are made to unify the two values assigned to a new variable, x; note that $\sigma(x)$ will be the value assigned to x in the unifier, σ, of the abstracted terms. The partially built unifier, θ, is passed into the next level of recursion so that it may be updated at that level, also. The parameter *PartialUnifier* is given an initial value equal to the identity unifier. The algorithm may return a sizable set of unifiers, especially if the unification of value pairs from the variable abstraction unifier and the abstraction set requires the recursive invocation of the algorithm, as is the case when the two values are terms of a common AC operator. Stickel only proved that the generalized algorithm terminates, is correct, and is complete for a subclass of general AC terms. However, the proof of these properties for the entire class of general AC terms has since been provided by Fages [*Fa*84].

Example 3.9: Let f be an AC operator and h be a null-E operator. Further, let $s = f(u, v, b)$ and $t = f(h(x, a), y)$ be terms. The variable abstractions of these terms are

$s' = f(u, v, x_1)$ and

$t' = f(x_2, y)$,

and the abstraction set is

$\theta = \{x_1 \leftarrow h(x, a), x_2 \leftarrow b\}$.

The unification of s' and t' yields a set, Σ, of 25 unifiers. When rectified with the values assigned to the new variables, x_1 and x_2, in the abstraction set, one obtains a complete set of unifiers for s and t:

$\{u \leftarrow h(x, a), y \leftarrow f(v, b)\}$,

$\{v \leftarrow h(x, a), y \leftarrow f(u, b)\}$,

$\{v \leftarrow f(z_2, h(x, a)), y \leftarrow f(z_2, u, b)\}$, and

$\{u \leftarrow f(z_1, h(x, a)), y \leftarrow f(z_1, v, b)\}$.

```
AC-UNIFY(Term1, Term2, PartialUnifier);
begin
   NewTerm1 := ABSTRACT(FLATTEN(Term1));                          (1)
   NewTerm2 := ABSTRACT(FLATTEN(Term2));                          (2)
   AbstractSet := the abstraction set from the previous two statements;
   AbstractUnifiers := AC-UNIFY-VO(NewTerm1, NewTerm2);
   if AbstractUnifiers exist
   then begin
      FinalUnifiers := {PartialUnifier};
      for σ ∈ AbstractUnifiers
         for x ← t ∈ AbstractSet begin
            Unifiers := EmptySet;
            for θ ∈ FinalUnifiers
               if (σ(x).root is AC) and (σ(t).root is AC)
               then Unifers := Unifiers ⋃ AC-UNIFY(σ(x), σ(t), θ)   (3)
               else Unifers := Unifiers ⋃ UNIFY(σ(x), σ(t), θ);     (4)
            FinalUnifiers := Unifiers;
         end;
   end
   else
      /* There are no unifiers of Term1 and Term2 */
      FinalUnifiers := EmptySet;
   return(FinalUnifiers);
end;


Notes:
   FLATTEN(Term) returns the flattened form of Term.
   ABSTRACT(Term) returns a variable abstraction of Term.
   UNIFY(Term1, Term2, PartialUnifier) is a recursive form of
      Robinson's unification algorithm.
```

Figure 5. Stickel's generalized AC unification algorithm

d. The Christian-Lincoln AC Algorithm.

Stickel's AC unification algorithm and its derivatives (for example, ACI unification) can be very inefficient: Many potential unifiers are generated and then thrown out because they violate the constraints of the problem. However, Christian and Lincoln have developed an algorithm for unifying linear pairs of AC terms

[*CL*88]. A *linear pair of AC terms* is a pair of terms in which each variable occurs only once. The algorithm is based upon Stickel's algorithm and reduces the run time, for this class of terms, by a factor of 3 to 4.

Christian and Lincoln observed that when a linear pair of AC terms is abstracted, the resulting pair will also be linear. They also observed that all coefficients in the diophantine equation corresponding to a linear pair of terms will have a value of 1. This means that the basis set of solutions for the equation will consist of only those solutions in which exactly one variable on each side of the solution has a value of 1, and all others have a value of 0. With such a regular pattern of solutions in the basis, we do not have to go through the costly process of solving the diophantine equation. Rather, a set of solutions matching this pattern can be quickly generated. Since there are exactly two variables in each solution of the basis that have non-zero values, the basis can be represented as a matrix. Table II shows the basis and the matrix for the diophantine equation,

$$x_1 + x_2 + x_3 + x_4 = y_1 + y_2 + y_3 + y_4,$$

as presented by Christian and Lincoln. Table III shows the matrix representation of the basis of table II. Before variable abstraction, each AC term to be unified is sorted in the following order: constants, terms, and then variables. The basis matrix can then be divided into nine *regions*, as shown in table IV. By performing some computationally simple analyses on the entries within each region, the valid unifiers can be generated from the matrix. For example, as seen in table IV, the introduced variables in the constant/constant region of the matrix must be set to 0, since a value of 1 would mean the unification of a constant with a different constant. (Remember, arguments common to both terms to be unified are removed before variable abstraction takes place.) A similar argument shows that the introduced variables in the constant/term and term/constant regions must also be set to 0.

Table II. BASIS SET PRESENTED BY CHRISTIAN AND LINCOLN.

x_1	x_2	x_3	x_4	y_1	y_2	y_3	y_4	Introduced Variable
0	0	0	1	0	0	0	1	z_1
0	0	0	1	0	0	1	0	z_2
0	0	0	1	0	1	0	0	z_3
0	0	0	1	1	0	0	0	z_4
0	0	1	0	0	0	0	1	z_5
0	0	1	0	0	0	1	0	z_6
0	0	1	0	0	1	0	0	z_7
0	0	1	0	1	0	0	0	z_8
0	1	0	0	0	0	0	1	z_9
0	1	0	0	0	0	1	0	z_{10}
0	1	0	0	0	1	0	0	z_{11}
0	1	0	0	1	0	0	0	z_{12}
1	0	0	0	0	0	0	1	z_{13}
1	0	0	0	0	0	1	0	z_{14}
1	0	0	0	0	1	0	0	z_{15}
1	0	0	0	1	0	0	0	z_{16}

Table III. THE MATRIX REPRESENTATION OF A BASIS.

	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 IV. THE REGIONS OF A BASIS MATRIX.

	C	T	V
C	0	0	...
T	0	...	...
V	...	...	...

e. An ACI Unification Algorithm.

The diophantine process, and thus the AC unification algorithms of Stickel, can be easily adapted to unify terms containing operators that exhibit the properties of associativity, commutativity, and identity (ACI). Recall that in the diophantine process associated with flattened, variable-only AC terms, the general form of a solution is the sum of some multiple (zero or more) of each of the solutions in the basis set,

$$z_1 b_1 + z_2 b_2 + \cdots z_k b_k.$$

Also recall that the solution corresponding to each potential unifier of the AC terms is obtained by setting a subset of the coefficients, $z_1, \ \ldots, \ z_k$, in the general solution to a value of 0. Any of these solutions which would cause one of the equation variables to be assigned a value of 0 is discarded, since it would cause the same variable in the unifier of the terms to be assigned the identity term, or null term, which is not possible in an AC theory.

However, these troublesome solutions are no problem when dealing with an ACI equational theory. Since variables may be assigned an identity value and "disappear" from a term, the solutions discarded as invalid for an AC theory are valid for an ACI theory. The general solution to the diophantine equation, given above, corresponds to a unifier, σ_0, of the pair of ACI terms. It can be seen that any solution obtained by setting to 0 some of the coefficients in the general solution corresponds to a unifier obtained by applying a subset of

$$\{z_1 \leftarrow e, \ \ldots, \ z_k \leftarrow e\}$$

to σ_0. This means that any unifier of the ACI terms can be obtained by composing some subset of the above set of substitutions with σ_0. Thus, σ_0, the unifier corresponding to the general solution to the diophantine equation, is the single most general unifier of a pair of flattened, variable-only ACI terms.

Stickel's restricted AC unification algorithm, presented in figure 4, can be transformed into a function ACI-UNIFY-VO($Term_1$, $Term_2$) which unifies a pair of variable-only ACI terms, $Term_1$ and $Term_2$, by deleting the code of the final "for" loop and returning the value of the variable *BaseUnifier* as the value of the function. Stickel's generalized AC unification algorithm, presented in figure 5, can be changed into a generalized ACI unification algorithm, ACI-UNIFY($Term_1$, $Term_2$), by replacing the call to ACI-UNIFY($Term_1$, $Term_2$) with a call to ACI-UNIFY-VO($Term_1$, $Term_2$).

4. The Yelick Model of E-Unification.

Given two equational theories, A and B, for which correct and complete E-unification algorithms are known, the problem of finding an E-unification algorithm for the combined equational theory, $A \cup B$, is not a trivial task. However, Yelick [*Ye85*] has shown that for *confined regular equational theories,* a top-level program can be written to invoke the individual, finite, complete, recursive E-unification algorithms and return a complete and correct set of unifiers for terms containing operators from some or all of the involved equational theories.

A *non-confining equation* is one of the form $x = t$, where x is a variable and t is a non-variable term. An equational theory containing no non-confining equations is a *confined* theory. An example of a non-confining equation is one defining an identity element, e, for an operator, f:

$f(x, e) = x$.

An equational theory is *regular* if , for each equation, $s = t$, in the definition of the theory, **vars**(s) = **vars**(t).

Yelick's model of unification is the basis for the E-unification algorithm used in this research. The pseudo-code for this implementation is given in chapter 5.

5. Computational Complexities of E-unification.

Kapur and Narendran [*KN*86.1] gathered complexity statistics for quite a variety of unification algorithms. The complexities for unification corresponding to some commonly occurring equational theories appear in table V, along with the references in which the complexities first appeared.

Table V. E-UNIFICATION COMPLEXITIES OF SOME COMMONLY USED THEORIES.

Equational Theory	Unification Complexity	Reference
Null-E	Linear	[*PW*78]
C	NP-Complete	[*Se*79]
A	Decidable	[*Ma*77]
AC	NP-Complete	[*KN*86.2]
ACI	NP-Complete	[*KN*86.2]
ACId	NP-Complete	[*KN*86.2]

IV. A REVIEW OF COMPLETION PROCEDURES

It is common practice for human mathematicians to rewrite a mathematical term into another term to which it is equal. The simplification of an algebraic expression and the solution of a trigonometric identity are two examples in which terms are iteratively changed through a sequence of rewrites until a goal is reached (those goals being the achievement of a normal form for algebraic simplification and the discovery of identical terms for the solution of an identity). Whether implicitly or explicitly stated, rewriting is performed via a set of rewrite rules, or identities, each of which have the form $t_1 = t_2$.

However, when term rewriting is automated and a finite set of identities is used as the set of rewrite rules, problems are encountered. One problem is that, if a term is rewritten using some rule in a left-to-right manner, that is, replacing a term matching the form of the left-hand side of the rule with one matching the right-hand side of the rule, the system may immediately rewrite the result back to the original term using the same rule in a right-to-left application. If this were to continue, the result would be an infinite sequence of rewrites oscillating between a pair of terms. Another problem arises because of the presence of a rule in which the left-hand side of the rule is a term contained as a proper subterm of the right-hand side (or vice versa). If such a rule is applied in a left-to-right manner, the resulting term is more complex than the original, but contains an instance of the original. The same rule could be applied repeatedly to each resultant term, leading to an infinite sequence of terms, each more complex than the one from which it was rewritten.

Example 4.1: Let $s = f(f(a, a), e)$ be a term. Applying the rule describing associativity for f,

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

iteratively to s in a left-to-right then right-to-left manner results in the looping, infinite sequence of rewritten terms,

$$f(a, f(a, e)),\ \ f(f(a, a), e),\ \ f(a, f(a, e)),\ \ldots .$$

Applying a rewrite rule describing an identity element, e, for f,

$$f(x, e) = x,$$

repeatedly to s in a right-to-left manner results in an infinite sequence of rewritten terms,

$$f(f(f(a, a), e), e),\ \ f(f(f(f(a, a), e), e), e),\ \ f(f(f(f(f(a, a), e), e), e), e),\ \ldots .$$

These problems can be overcome, however, by transforming the set of identities used as rewrite rules into a set of reductions. A *reduction* is an ordered pair of terms of the form $\lambda \rightarrow \rho$, such that $\lambda = \rho$ is an identity and λ is, in some sense, simpler than ρ. A reduction can be used to rewrite a term, t, only if there exists a *match* between λ and t/i, that is, if there exists a set of substitutions, σ, such that $\sigma(\lambda)$ is equal to the subterm of t at some position, $i \in \mathbf{dom}(t)$. The rewritten term is $t[i \leftarrow \sigma(\rho)]$, the result of replacing subterm t/i with $\sigma(\rho)$. The relation $\underset{r}{\rightarrow}$ specifies the rewriting of one term to another by a single application of a reduction, r, to the first term. Thus, $t \underset{r}{\rightarrow} t'$ specifies that one application of r rewrites term t into term t'. In a like fashion, the relation $\underset{R}{\rightarrow}$ specifies the rewriting of one term to another by a single application of a reduction from a set of reductions, R. The transitive closures of $\underset{r}{\rightarrow}$ and $\underset{R}{\rightarrow}$ are the relations $\underset{r}{\rightarrow}^{+}$ and $\underset{R}{\rightarrow}^{+}$, respectively. Likewise, their reflexive, transitive closures are the relations $\underset{r}{\rightarrow}^{*}$ and $\underset{R}{\rightarrow}^{*}$. A term which cannot be rewritten by any reduction in a set of reductions R is said to be *irreducible* with respect to R. An *irreducible form* or *terminal form* of a term, t, with respect to R, written as $t{\downarrow}^{R}$, is an irreducible term, t', such that $t \underset{R}{\rightarrow}^{*} t'$.

Example 4.2: Let $s = f(f(a, a), e)$. Applying a reduction,

$$f(f(x, y), z) \rightarrow f(x, f(y, z)),$$

formed from the first rule of example 4.1, to s results in the term

$s' = f(a, f(a, e))$,

which is irreducible with respect to the reduction; so, the oscillation displayed in example 4.1 has disappeared. Applying a second reduction,

$f(x, e) = e$,

formed from the second rule of example 4.1, to s produces the term

$s'' = f(a, a)$,

which is irreducible with respect to the second reduction; thus the second infinite sequence that was seen in example 4.1 has been eliminated.

A. COMPLETE SETS OF REDUCTIONS

The *word problem* is that of deciding whether or not two terms are provably equal with respect to some relation. In general, the word problem is undecidable [*KB*70]. However, the word problem can be easily solved with respect to a relation if there exists, for that relation, a finite complete set of reductions.

Definition 4.1: A set of reductions is a *complete set of reductions* if each term has exactly one irreducible term and no distinct irreducible terms are equivalent, with respect to the set of reductions.

The first restriction of this definition is actually a consequence of the second; since distinct irreducible terms are not equivalent, and all new terms produced by reducing a term are considered equivalent, there can be only one irreducible term produced.

B. THE KNUTH-BENDIX COMPLETION PROCEDURE

In 1970, Knuth and Bendix published a pioneering paper in the study of complete sets of reductions [*KB*70]. In their paper, they investigated the conditions under which a set of reductions is complete and, as a consequence, derived an algorithm for testing the completeness of a set of reductions, and extended it to a procedure for completing an incomplete set of reductions (in many cases).

1. The Conditions for a Complete Set of Reductions.

In order to meet the conditions for completeness specified in definition 4.1, a set of reductions must exhibit the finite termination property and be a Church-Rosser set of rewrite rules, as explained below.

a. The Finite Termination Property.

A set of reductions, R, has the *finite termination property* if there exists no infinite chain of rewrites,

$$t = t_0 \underset{R}{\rightarrow} t_1 \underset{R}{\rightarrow} t_2 \underset{R}{\rightarrow} \ldots .$$

If R has this property, then the process of rewriting a term to an irreducible term, with respect to R, is a finite process. Every term will rewrite to at least one irreducible term.

To guarantee the finite termination property for a set of reductions, a well-founded partial order on the set of all terms must be found. A *well-founded partial order (wpo)* is a partial order which has no infinitely descending chains. The wpo will be based upon a *weighting function*, which associates with each term a measure of its complexity. The value of the weighting function for a term, t, is designated as **weight**(t). The well-founded partial order, $\succ$, relative to the weighting function, is defined as follows for the set of all terms, T:

(1) $(\forall s,\ t \in T)$ If **weight**$(s) >$ **weight**(t), then $s \succ t$.

(2) $(\forall s,\ t \in T)$ If **weight**$(s) =$ **weight**(t), then $s \not\succ t$, that is, s and t are not related by the wpo $\succ$.

There are some restrictions that must be met by any weighting function chosen. For a weighting function to be applicable, the following conditions must hold:

(1) There must not be an infinite set of terms, $\{t_1, t_2, t_3, \ldots\}$, such that **weight**$(t_1) >$ **weight**$(t_2) >$ **weight**$(t_3) > \cdots$ This insures that $\succ$ is indeed a wpo.

(2) $(\forall s,\ t \in T)$ If **weight**$(s) >$ **weight**(t) and σ is a set of substitutions, then **weight**$(\sigma(s)) >$ **weight**$(\sigma(t))$, that is, term ordering must be preserved by substitution.

(3) $(\forall s,\ t_1,\ t_2 \in T)$ If **weight**$(t_1) >$ **weight**(t_2), then it must be true that $(\forall i \in \mathbf{dom}(s))$ **weight**$(s[i \leftarrow t_1]) >$ **weight**$(s[i \leftarrow t_2])$, that is, term ordering must be preserved by subterm replacement.

(4) $(\forall \lambda \rightarrow \rho \in R)$ **weight**$(\lambda) >$ **weight**(ρ), that is, $\lambda \succ \rho$.

b. The Church-Rosser Property.

A finite set of reductions possessing the finite termination property alone is sufficient to solve the word problem, with respect to the set of reductions. Every term has a finite number of subterms, so there are only a finite number of ways to rewrite a given term by a single reduction application. Due to the finite termination property, every possible rewrite sequence is finite in length. Therefore, a complete *rewrite tree* can be developed for any given term. Branches of the tree correspond to rewriting sequences, and the leaves of the tree correspond to all irreducible terms that can be produced from the *root* term. It can be decided, then, whether or not two terms are equivalent, with respect to the set of reductions, by generating the rewrite tree for each of the terms and then searching the trees for a common irreducible term.

However, if the branching factor or depth of the trees is very large, this will be a very expensive search process. This is the reason for adding the requirement of the Church-Rosser property to a set of reductions.

Definition 4.2: A set of rewrite rules is *Church-Rosser* if terms that are equivalent, with respect to the set of rules, have a common rewriting.

Note that the definition of the Church-Rosser property does not state that the common rewriting must be irreducible; thus, it could be that the common rewriting can be further rewritten several ways into several different terms. These terms, however, are equivalent and must, therefore, have a common rewriting. This fluctuating behavior could continue indefinitely if not for the finite termination property, which requires that each rewriting sequence halts. Because the set is Church-Rosser, there must exist a common irreducible term at which all rewriting sequences halt. Therefore, it can be seen that a Church-Rosser set of reductions possessing the finite termination property does indeed satisfy the definition of a complete set of reductions.

c. The Lattice Condition.

The finite termination property is assured by the selection of a term weighting function that produces a well-founded partial order on terms and meets the requirements specified earlier. But how is a set of reductions shown to be Church-Rosser? The proof is based on the fact that a set of reductions is Church-Rosser if it has the finite termination property and is confluent.

Definition 4.3: A set of reductions, R, is *confluent* if, for all terms t, t_1, and t_2, where $t \underset{R}{\rightarrow}^* t_1$ and $t \underset{R}{\rightarrow}^* t_2$, there exists a term, t', such that $t_1 \underset{R}{\rightarrow}^* t'$ and $t_2 \underset{R}{\rightarrow}^* t'$, that is, if all rewritten forms of a given term have a common rewriting.

Confluence is pictorially described in figure 6(a). Even though the finite termination property guarantees that a term has a finite rewrite tree, it can be difficult to prove that a set of reductions is confluent. Since t is rewritten into terms t_1 and t_2 using the relation $\underset{R}{\rightarrow}^*$, the set of terms which take on the roles of t_1 and t_2 could be quite large, and the pairwise testing of these terms could be expensive.

Fortunately, it is not necessary to prove confluence in order to show a set of reductions to be Church-Rosser. It has also been shown that a set of reductions is confluent if it has the finite termination property and is locally confluent.

Definition 4.4: A set of reductions R is *locally confluent* if, for all terms t, t_1, and t_2, where $t \underset{R}{\rightarrow} t_1$ and $t \underset{R}{\rightarrow} t_2$, there exists a term, t', such that $t_1 \underset{R}{\rightarrow}^* t'$ and $t_2 \underset{R}{\rightarrow}^* t'$, that is, if all terms derived from a given term by a single application of a reduction have a common rewriting.

Local confluence is diagrammed in figure 6(b). The proof that a set is locally confluent is easier than the proof that the set is confluent since, in general, the number of rewritten terms derivable from a term by a single reduction application will be fewer than the number of those derivable from the same term by any number of reduction applications.

The relationships between complete, Church-Rosser, confluent, and locally confluent sets of reductions are summarized in theorem 4.1.

Theorem 4.1: The following statements about a set R of reductions possessing the finite termination property are equivalent:

(1) R is a complete set of reductions.

(2) R is Church-Rosser and has the finite termination property.

(3) R is confluent and has the finite termination property.

(4) R is locally confluent and has the finite termination property.

Thus, to show that a set of reductions, *R*, is a complete set of reductions, one needs only to show that R is a locally confluent set and possesses the finite termination property. Knuth and Bendix call the local confluence property the *lattice condition.* It is the lattice condition upon which the superposition process, that is, the Knuth-Bendix test for completeness, is based.

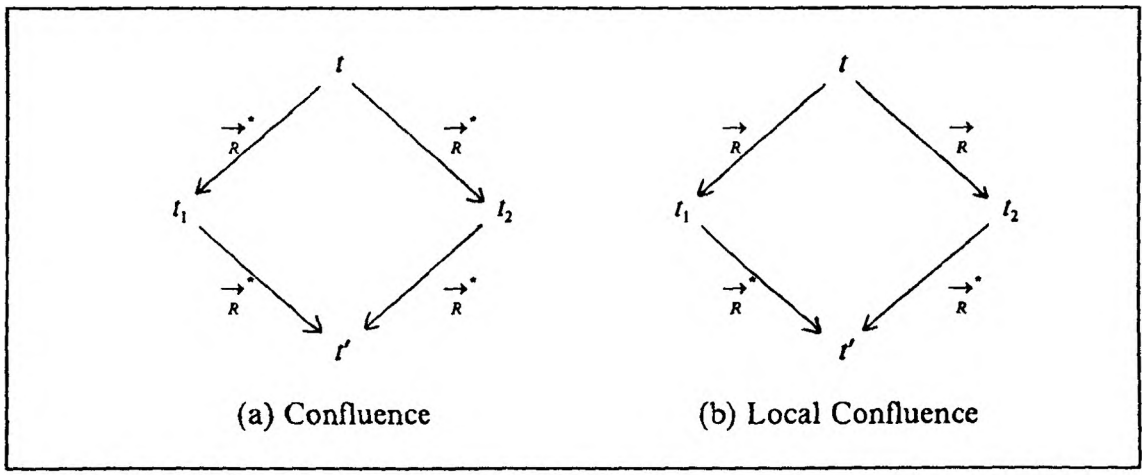


Figure 6. Confluence and local confluence.

2. The Test for Completeness.

Testing whether or not local confluence holds for each term, with respect to a finite set of reductions possessing the finite termination property, constitutes a test for the local confluence of the set of reductions and, consequently, a test for the completeness of the set. However, it is not a viable test; although the rewrite tree associated with each term is finite, there are an infinite set of terms to be tested! In their paper, Knuth and Bendix described a procedure for deciding the local confluence of a set of reductions that avoids this problem. It is called the *superposition process,* and it needs only to test the finite set of left-hand terms of the reductions for local confluence. It will now be shown that proving local confluence by the superposition

process is sufficient to prove local confluence for all terms, with respect to the set of reductions.

Let t be an arbitrary term to be tested for local confluence. Referring to figure 6(b), local confluence will hold for t only if every pair of terms, t_1 and t_2, produced by a single application to t of a reduction will *conflate*, that is, reduce to a common irreducible term. Let $r_1 = \lambda_1 \rightarrow \rho_1$ and $r_2 = \lambda_2 \rightarrow \rho_2$ be (possibly identical) members of the set of reductions, R, such that $t \underset{r1}{\rightarrow} t_1$ and $t \underset{r2}{\rightarrow} t_2$. This implies that there exist matches, σ_1 and σ_2, and positions, $i, j \in \mathbf{dom}(t)$, such that $t/i = \sigma_1(\lambda_1)$ and $t/j = \sigma_2(\lambda_2)$. One of three relationships must hold between subterms t/i and t/j:

(1) t/i and t/j are disjoint subterms of t. In this case, t_1 and t_2 trivially and unconditionally conflate, since the two rewrites do not interfere with one another in any way, that is, it will always be true that $t \underset{r1}{\rightarrow} t_1 \underset{r2}{\rightarrow} t'$ and $t \underset{r2}{\rightarrow} t_2 \underset{r1}{\rightarrow} t'$.

(2) t/i and t/j overlap, but not completely. This case is impossible, which is apparent from the tree structure of terms.

(3) t/i and t/j overlap completely, that is, t/i is a subterm of t/j, or vice versa. This is the only one of the three cases which must be further investigated.

We shall assume, without loss of generality, that t/j is a subterm of t/i, that is, that there exists a position, k, such that $j = i.k$. We shall also assume, without loss of generality, that r_1 and r_2 are variable disjoint, implying that $\mathbf{lvars}(\sigma_1)$ and $\mathbf{lvars}(\sigma_2)$ are also variable disjoint.

Since $t/i = \sigma_1(\lambda_1)$ and $t/j = t/i.k = \sigma_2(\lambda_2)$, it follows that $\sigma_1(\lambda_1)/k = \sigma_2(\lambda_2)$. It can also be shown that there exists some position, $k' \in \mathbf{dom}(\lambda_1)$, such that $\sigma_1(\lambda_1)/k = \sigma_1(\lambda_1/k')$.

Because r_1 and r_2 are variable disjoint (as are **lvars**(σ_1) and **lvars**(σ_2)), it will also be true that $\sigma_1(\lambda_1/k') = \sigma_1 \circ \sigma_2(\lambda_1/k')$ and $\sigma_2(\lambda_2) = \sigma_1 \circ \sigma_2(\lambda_2)$. Thus, $\sigma_1 \circ \sigma_2(\lambda_1/k') = \sigma_1 \circ \sigma_2(\lambda_2)$, which makes $\sigma_1 \circ \sigma_2$ a unifier of λ_1/k' and λ_2. Therefore, there must exist a most general unifier, θ, for λ_1/k' and λ_2. The forms of the rewritten terms, t_1 and t_2, are

$t_1 = t[i \leftarrow \sigma_1(\rho_1)]$ and

$t_2 = t[i \leftarrow \sigma_1(\lambda_1[k' \leftarrow \sigma_2(\rho_2)])]$.

Using the facts stated above, we can replace these by the equivalent forms,

$t_1 = t[i \leftarrow \sigma_1 \circ \sigma_2(\rho_1)]$ and

$t_2 = t[i \leftarrow \sigma_1 \circ \sigma_2(\lambda_1[k' \leftarrow \rho_2])]$.

The mgu, θ, is more general than the unifier $\sigma_1 \circ \sigma_2$, so we can replace the forms of t_1 and t_2 once more by the forms

$t_1 = t[i \leftarrow \theta(\rho_1)]$ and

$t_2 = t[i \leftarrow \theta(\lambda_1[k' \leftarrow \rho_2])]$.

It can be seen that these last forms of t_1 and t_2 are identical, with the exception of the terms replacing subterm t/i. So the problem of deciding whether or not t_1 and t_2 conflate is simplified to deciding whether or not t_1/i and t_2/i conflate. Thus, a term, t, is locally confluent if all pairs of terms,

$$< \theta(\rho_1),\ \theta(\lambda_1[k' \leftarrow \rho_2]) >,$$

conflate, where $\lambda_1 \rightarrow \rho_1$ and $\lambda_2 \rightarrow \rho_2$ are reductions, $k' \in$ **dom**(λ_1), and $\theta(\lambda_1/k') = \theta(\lambda_2)$.

Pairs of the form $< \theta(\rho_1),\ \theta(\lambda_1[k' \leftarrow \rho_2]) >$ are called *critical pairs*. (This terminology was not actually used by Knuth and Bendix, but was introduced later.) The process of forming and reducing all critical pairs is called the *superposition process*. Note that the same set of critical pairs is formed, regardless of the term being tested for local confluence. Thus, performing the superposition process for one term is equivalent to performing it for all terms. Therefore, the problem of testing an infinite number of terms is reduced to testing the finite set of left-hand sides of the

reductions from the set of reductions. This constitutes a decision procedure for the completeness of a set of reductions.

3. The Completion Procedure.

Knuth and Bendix extended this completeness decision procedure to one for completing an incomplete set of reductions, as pictured in the pseudo-code of figure 7. The input to the procedure is a set of equations, S, that is formed into the initial set of reductions, R. Note in statement (1) that only the members of the strict domain (**sdom**) of λ_1 are unified with λ_2, rather than the entire domain (**dom**) of λ_1. This is because those critical pairs formed from the variable subterms of λ_1 trivially conflate.

The critical pairs generated from the set of reductions are iteratively produced. As each critical pair is calculated, its two component terms are reduced to irreducible forms, t_1 and t_2, using R. (See statements (2) and (3) in the pseudo-code.) If $t_1 = t_2$, then the critical pair has conflated, and the next critical pair is calculated and processed. If all critical pairs conflate, then the set R is a complete set of reductions, and a success status is returned along with R.

If, however, $t_1 \neq t_2$, then the pair of irreducible terms needs to be added as a reduction to R to make it "more complete." If **weight**(t_1) = **weight**(t_2), then t_1 and t_2 are not related by the well-founded partial order on terms, $\succ$; thus, the pair cannot be ordered into a reduction, and the procedure must return a failure status. But, if **weight**$(t_1) >$ **weight**(t_2) or **weight**$(t_1) <$ **weight**(t_2), then the reduction $t_1 \rightarrow t_2$ or $t_2 \rightarrow t_1$, respectively, is added to R. After the new reduction is added, *inter-reduction simplification* takes place, in which the two terms comprising each reduction in R are reduced to irreducible form, with respect to the other reductions in the set. If a reduction is reduced to a pair of identical terms, it is dropped from R. Finally, after

inter-reduction simplification has been completed, the entire completion process must be started again, using the newly updated set, *R*.

If all critical pairs generated from any version of *R* conflate, then that version is a complete set of reductions equivalent to *S*. However, there is also a possibility that the completion procedure will never halt; some complete sets of reductions are infinite in size. An example of one such complete set of reductions is given in the discussion of the work of Peterson and Stickel.

4. Failure-Resistance.

Several years after the development of the Knuth-Bendix completion procedure, Forgaard and Guttag conceived the notion of a *failure-resistant* completion procedure [*FG*84]. Their method does not always prevent the completion procedure from failing, but it can in some cases. It is based on a surprisingly simple idea. When a critical pair based on a surprisingly simple idea. In the original Knuth-Bendix procedure, when a critical pair is reduced to two distinct terms that have identical weights, the procedure halts with failure. In the failure-resistant Knuth-Bendix procedure, such a critical pair is *shelved*, or put aside, and work continues on the next critical pair. When all critical pairs have been processed, those that were shelved are reprocessed, since a reduction added to the set of reductions after a shelved pair was set aside may now enable it to be conflated or ordered into a reduction; if not, the pair will be reshelved. This iterative process continues either until all shelved pairs have been successfully handled, producing a complete set of reductions, or until no shelved pair can be conflated or ordered, leading to failure of the procedure.

```
KB-COMPLETION(S);
begin
   R := the set of reductions formed from the equations of S;
   repeat
      Status := SUCCESS;
      for (∀r1 = λ1 → ρ1 ∈ R)
         for (∀r2 = λ2 → ρ2 ∈ R)
            for (∀i ∈ sdom(λ1)) begin                                  (1)
               θ := UNIFY(λ1/i, λ2);
               if θ exists
               then begin
                  t1 := REDUCE*(θ(ρ1), R);                             (2)
                  t2 := REDUCE*(θ(λ1[i ← ρ2]), R);                     (3)
                  case
                     t1 = t2:
                        /* Successful Conflation */ ;
                     weight(t1) > weight(t2): begin
                        add t1 → t2 to R;
                        inter-reduce R;
                        Status := LOOP;
                        exit outer "for" loop;
                     end;
                     weight(t1) < weight(t2): begin
                        add t2 → t1 to R;
                        inter-reduce R;
                        Status := LOOP;
                        exit outer "for" loop;
                     end;
                     weight(t1) = weight(t2): begin
                        Status := FAILURE;
                        exit outer "for" loop;
                     end;
                  end;
               end;
            end;
   until (Status = SUCCESS) or (Status = FAILURE);
   return(Status, R);
end;


Notes:
   REDUCE*(Term, Reductions) returns an irreducible form of Term,
      with respect to Reductions.
   UNIFY(Term1, Term2) is Robinson's unification algorithm for a
      pair of terms, Term1 and Term2.
```

Figure 7. The Knuth-Bendix completion procedure

C. THE PETERSON-STICKEL E-COMPLETION PROCEDURE

Although the Knuth-Bendix completion procedure is certainly interesting, it has some serious limitations. For example, two axioms which are common to many equational theories are those of associativity and commutativity. But, the Knuth-Bendix procedure cannot properly address either of them without destroying the finite termination property of the set of reductions. A commutativity axiom, such as $f(x, y) = f(y, x)$, quite obviously cannot be ordered into a reduction, and will cause the completion procedure to fail. An axiom of associativity, such as $f(f(x, y), z) = f(x, f(y, z))$, can be transformed into a reduction, $f(f(x, y), z) \rightarrow f(x, f(y, z))$. But the Knuth-Bendix procedure is not totally general in its treatment of associativity as a reduction, and can lead to non-termination of the procedure. The following example, 4.3, was given by Peterson and Stickel.

Example 4.3: Let the set of equations input to the Knuth-Bendix completion procedure be the equations,

$$f(f(x, y), z) = f(x, f(y, z)), \tag{1}$$

$$f(a, b) = b, \text{ and} \tag{2}$$

$$f(a, f(x, b)) = f(x, b). \tag{3}$$

The Knuth-Bendix procedure will produce an infinite set of reductions,

$$f(f(x, y), z) \rightarrow f(x, f(y, z)),$$

$$f(a, b) \rightarrow b,$$

$$f(a, f(x, b)) \rightarrow f(x, b),$$

$$f(a, f(x, f(x_0, b))) \rightarrow f(x, f(x_0, b)),$$

$$f(a, f(x, f(x_0, f(x_1, b)))) \rightarrow f(x, f(x_0, f(x_1, b))),$$

However, Peterson and Stickel observed that if equation (1), the associativity axiom for operator f, is removed from the set of equations and f is assumed to be an associative operator, then the two reductions,

$f(a, b) \rightarrow b$ and

$f(a, f(x, b)) \rightarrow f(x, b)$,

constitute a complete set of reductions for the set containing equations (2) and (3).

This observation was used by Peterson and Stickel to develop extensions of the Knuth-Bendix procedures. The set of equations, S, input to the Peterson-Stickel procedure is divided into two sets, E and R. The set E is a subset of S for which there exists a finite, complete E-unification algorithm. All other members of S are ordered into reductions to form R. There is a restriction on the members of E, however: All reductions in E must be *linear, collapse-free* equations, that is, every variable occurring in an equation must appear exactly twice, once in each side of the equation.

In addition to the necessity of an E-unification algorithm for the set E, an E-matching algorithm and an algorithm for proving E-equality, with respect to E, are also needed in order to implement an E-completeness decision procedure or an E-completion procedure. Peterson and Stickel showed that the existence of an E-unification algorithm for E implies the existence of the other two algorithms.

1. E-Complete Sets of Reductions.

An equational theory, E, partitions the set of all terms into equivalence classes. Further, since the equations are linear and collapse-free, the equivalence classes are finite in size. A new relation, $\underset{R/E}{\rightarrow}$, which is equivalent to the composition of relations, $\underset{E}{=} \circ \underset{R}{\rightarrow} \circ \underset{E}{=}$, will be used to specify the rewriting of any member of one equivalence class to any member of another. The transitive closure and reflexive, transitive

closure of $\underset{R/E}{\rightarrow}$ are the relations, $\underset{R/E}{\rightarrow^+}$ and $\underset{R/E}{\rightarrow^*}$, respectively. The relation $\underset{R/E}{\rightarrow}$ may also be written as R/E, and its closures written as R/E^+, and R/E^*. The definition given previously for complete sets of reductions can now be extended to provide for a non-empty equational theory.

Definition 4.5: Let E be a linear, collapse-free equational theory. A set of reductions R is an *E-complete set of reductions* if, for all terms s and t which are equivalent with respect to R, $s \underset{R/E}{\rightarrow^*} s'$, $t \underset{R/E}{\rightarrow^*} t'$, and $s' \underset{E}{=} t'$.

In a manner similar to that used by Knuth and Bendix, it can be proven that a set of reductions R is E-complete if and only if all critical pairs of the members of R conflate and R is an E-compatible set of reductions. The critical pairs used to test for E-completeness have the same form as those used to test for standard completeness: $< \theta(\rho_1),\ \theta(\lambda_1[i \leftarrow \rho_2]) >$. However, it is almost certain that the number of critical pairs will be greater in the E-completeness test. This is because the null-E unification of each pair λ_1/i and λ_2 in the completeness test produces, at most, one most general unifier, and thus, one critical pair. However, the E-unification algorithm used in the E-completeness test returns a (finite) set of maximally general unifiers, each corresponding to a critical pair. This fact once again emphasizes the need for a minimal E-unification algorithm, or at least one that is as minimal as possible.

2. E-compatibility.

The second requirement for E-completeness is the E-compatibility of the set of reductions. This property is defined as follows.

Definition 4.6: Let E be a linear, collapse-free equational theory and R be a set of reductions. Assume, without loss of generality, that the elements of $E \cup R$ are variable disjoint. If, for all $l = r \in E$ and $\lambda_1 \rightarrow \rho_1 \in R$ such that $i \in \mathbf{sdom}(l)$, $i \neq \varepsilon$, and l/i and λ_1 are E-unifiable, there exists a reduction $\lambda_2 \rightarrow \rho_2 \in R$ and a set of

substitutions σ such that $l[i \leftarrow \lambda_1] =_E \sigma(\lambda_2)$ and $l[i \leftarrow \rho_1] \rightarrow^{\cdot}_{R/E} \sigma(\rho_2)$, then R is *E-compatible.*

The goal of Peterson and Stickel was to develop an E-completeness decision procedure and E-completion procedure for AC theories. An *AC equational theory* is one containing both an associativity axiom and a commutativity axiom for a set of operators. In order to insure E-compatibility for a set R, of reductions, with respect to an AC equational theory, they developed the concept of reduction extension.

Definition 4.7: Let $r = \lambda \rightarrow \rho \in R$, such that $\lambda.\text{root} = f$ is an AC operator. The *AC extension of* r is the reduction $r^{\cdot}_{AC} = f(x, \lambda_1) \rightarrow f(x, \rho_1)$, where $x \notin \text{vars}(r)$. The *AC extension of* R is the set $R^{\cdot}_{AC} = \{r^{\cdot}_{AC} \mid r \in R \wedge \lambda.\text{root is an AC operator}\} \cup R$.

It can be proven that if E is an AC theory and R is a set of reductions, then $R^{\cdot}_{AC}$ is E-compatible. Therefore, if E is an AC theory, the E-completeness of a set of reductions, R, possessing the finite termination property can be decided solely by checking for the conflation of all critical pairs produced from the reductions.

3. The AC Completion Procedure.

The Peterson-Stickel E-completeness decision procedure can be extended to an E-completeness procedure in much the same way that the Knuth-Bendix completeness decision procedure was extended. The pseudo-code for this procedure is given in figure 8.

```
PS-COMPLETION(S);
begin
   Reductions := EmptySet;
   Pairs := EmptySet;
   Eqs := S;
   while ((Pairs ≠ EmptySet) or (Eqs ≠ EmptySet)) do begin
      if Eqs = EmptySet
      then MAKE-CRITICAL-PAIRS(Pairs, Eqs);
      else begin
         <s, t> := the member of Eqs with the Smallest weight;
         Eqs := Eqs.– {<s, t>};
         s_1 := REDUCE*(s, Reductions);
         t_1 := REDUCE*(t, Reductions);
         if s_1 = t_1
         then  /* Successful conflation */
         else begin
            ADD-REDUCTION(s_1, t_1, Reductions, Pairs);
            INTER-REDUCE(Reductions, Pairs);
         end;
      end;
   end;
   return(Reductions);
end;


Notes:
   REDUCE*(Term, Reductions) returns an irreducible form of Term,
      with respect to Reductions.
```

Figure 8a. The Peterson-Stickel AC completion procedure, part 1 of 3.

```
ADD-REDUCTION(s, t, Reductions, Pairs);
begin
   case
      weight(s) > weight(t):
         λ → ρ := s → t;
      weight(s) < weight(t):
         λ → ρ := t → s;
      weight(s) = weight(t):
         HALT WITH FAILURE;
   end;
   Reductions := Reductions ∪ {λ → ρ};
   for r ∈ Reductions
      Pairs := Pairs∪ { < λ → ρ, r > , < λ → ρ, r'AC > ,
            < λ'AC → ρ'AC, r > , < λ'AC → ρ'AC, r'AC > };
end;


MAKE-CRITICAL-PAIRS(Pairs, Eqs);
begin
   {λ1 → ρ1, λ2 → ρ2} := the member of Pairs with the smallest value of
      weight(λ1) + weight(λ2);
   Pairs := Pairs − {λ1 → ρ1, λ2 → ρ2};
   Eqs :  { < σ(ρ1), σ(ρ2) > | σ ∈ csu(λ1, λ2)}
      ∪ { < σ(ρ1), σ(λ1[i ← ρ2]) > | λ1 → ρ1 is not an extension
         ∧ i ∈ sdom(λ1) ∧ σ ∈ csu(λ1/i, λ2)}
      ∪ { < σ(ρ2), σ(λ2[i ← ρ1]) > | λ2 → ρ2 is not an extension
         ∧ i ∈ sdom(λ2) ∧ σ ∈ csu(λ2/i, λ1)};
end;
```

Figure 8b. The Peterson-Stickel AC completion procedure, part 2 of 3.

The variable *Eqs* is a list of term pairs which must either conflate or be transformed into reductions; its initial value is the set of input equations, *S*. *Pairs* contains all reduction pairs that have not yet been through the *superposition process*, that is, the generation and attempted conflation of all critical pairs that can be formed from the pair of reductions. Whenever *Eqs* has been emptied, it is replenished by a call to the procedure MAKE-CRITICAL-PAIRS(*Pairs*, *Eqs*), which picks a member of *Pairs* and stores all critical pairs generated from that member into *Eqs*. If *Eqs* is empty and *Pairs* is also empty, then all reduction pairs have successfully passed

```
INTER-REDUCE(Reductions, Pairs);
begin
   repeat
      Status := SUCCESS;
      for λ → ρ ∈ Reductions begin
         λ1 := REDUCE*(λ, Reductions – {λ → ρ});
         ρ1 := REDUCE*(ρ, Reductions – {λ → ρ});
         if (λ ≠ λ1) or (ρ ≠ ρ1)
         then begin
            for r ∈ Reductions
               Pairs := Pairs– { < λ → ρ, r > , < λ → ρ, r*AC > ,
                     < λ*AC → ρ*AC, r > , < λ*AC → ρ*AC, r*AC > };
            Reductions := Reductions – {λ → ρ};
            if λ1 = ρ1
            then  /* Successful conflation */
            else begin
               ADD-REDUCTION(λ1, ρ1, Reductions, Pairs);
               Status := LOOP;
               exit "for" loop;
            end;
         end;
      end;
   until Status = SUCCESS;
end;
```

Figure 8c. The Peterson-Stickel AC completion procedure, part 3 of 3.

through the superposition process, and *Reductions* is a complete set of reductions equivalent to the input set of equations, *S*.

When a reduction is added to *Reductions*, it is paired with all reductions (including itself) and their AC extensions, and these pairs are added to *Pairs*. In a similar fashion, if a reduction is removed from *Reductions* during inter-reduction simplification, all pairs incorporating that reduction or its AC extension are removed from *Pairs*.

D. THE JOUANNAUD-KIRCHNER EXTENSIONS

The procedures developed by Peterson and Stickel subsume the work of Knuth and Bendix. In a like manner, the work of Jouannaud and Kirchner [*JK*86]. subsumes that of Peterson and Stickel, and others. Their work represents no major stride forward in the study of complete sets of reductions, as did that of Knuth and Bendix, and Peterson and Stickel. Rather, it is an attempt to "tidy up" and generalize the work that had come before.

1. Confluence and Local Confluence Revisited.

Jouannaud and Kirchner found that the investigation of E-complete sets of reductions could be made simpler and more general by replacing the relation, $\underset{R/E}{\rightarrow}$, by a new relation , $\underset{R^E}{\rightarrow}$, which can be any relation satisfying the inequality , $\underset{R}{\rightarrow} \subseteq \underset{R^E}{\rightarrow} \subseteq \underset{R/E}{\rightarrow}$. The transitive closure and the reflexive, transitive closure of $\underset{R^E}{\rightarrow}$ are the relations $\underset{R^E}{\rightarrow^+}$ and $\underset{R^E}{\rightarrow^*}$, respectively. These relations may be written as R^E, R^{E+}, and R^{E*}. Among other things, this permits an easing of the restriction placed on the equational theory, E, requiring it to be linear and collapse-free, to one simply requiring that it generate finite equivalence classes.

The properties of Church-Rosser, confluence, and local confluence, which are so important in the Peterson-Stickel procedures, can be formally restated for E-completeness in terms of $\underset{R^E}{\rightarrow}$.

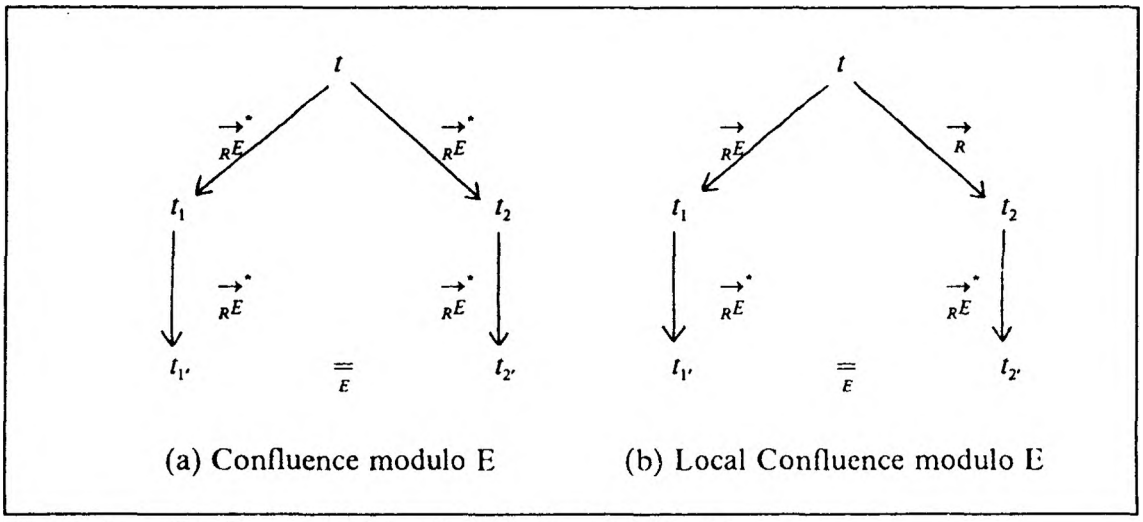
Definition 4.8: Let R be a set of reductions and let E be an equational theory defining finite equivalence classes. Let T be the set of all terms.

(1) R is R^E *-Church-Rosser modulo* E iff $(\forall s, t, s', t' \in T)$ s and t are considered equivalent, $s \underset{R^E}{\rightarrow^*} s'$, and $t \underset{R^E}{\rightarrow^*} t'$ imply that $s' \underset{E}{=} t'$.

(2) R^E is *confluent modulo* E iff $(\forall t, t_1, t_2 \in T)$ $t \underset{R^E}{\rightarrow^*} t_1$ and $t \underset{R^E}{\rightarrow^*} t_2$ imply that $(\exists t_1', t_2' \in T)$ $t_1 \underset{R^E}{\rightarrow^*} t_1'$, $t_1 \underset{R^E}{\rightarrow^*} t_2'$, and $t_1' \underset{E}{=} t_2'$.

(3) R^E is *locally confluent modulo E with R* iff $(\forall t, t_1, t_2 \in T)$ $t \underset{R^E}{\rightarrow} t_1$ and $t \underset{R}{\rightarrow} t_2$ imply that $(\exists t_1', t_2' \in T)$ $t_1 \underset{R^E}{\rightarrow}^* t_1'$, $t_1 \underset{R^E}{\rightarrow}^* t_2'$, and $t_1' \underset{E}{=} t_2'$.

The properties of definitions 4.8.(2) and 4.8.(3) are illustrated in figure 9.



(a) Confluence modulo E (b) Local Confluence modulo E

Figure 9. Confluence and local confluence modulo E.

2. Coherence and Local Coherence.

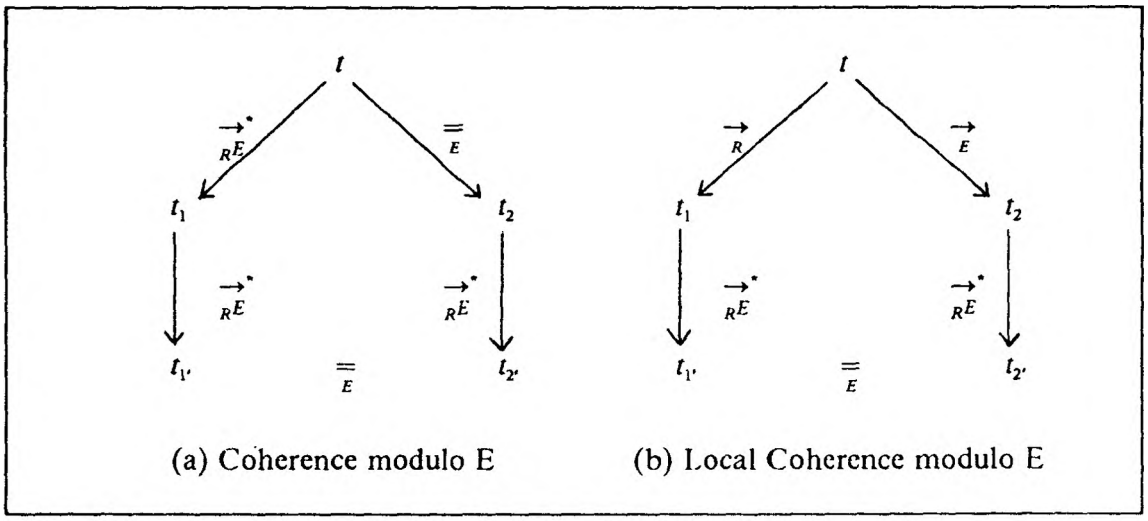
Recall that Peterson and Stickel defined the property of *E-compatibility*, and showed it to be a necessary property to insure the E-completeness of a set of reductions. This property was generalized by Jouannaud and Kirchner into a property called *coherence*. As can be seen by comparing confluence in figure 9(a) with coherence in figure 10(a), these two properties are both instances of the *lattice condition* defined by Knuth and Bendix. In fact, just as confluence can be deduced by proving local confluence, coherence can be inferred from local coherence, pictured in figure 10(b).

Definition 4.9: Let R be a set of reductions and let E be an equational theory defining finite equivalence classes. Let T be the set of all terms.

(1) R^E is *coherent modulo* E iff $(\forall t, s_1, t_2 \in T)$ $t \underset{R^E}{\rightarrow^*} t_1$ and $t \underset{E}{=} t_2$ imply that $(\exists t_1', t_2' \in T)$ $t_1 \underset{R^E}{\rightarrow^*} t_1'$, $t_1 \underset{R^E}{\rightarrow^*} t_2'$, and $t_1' \underset{E}{=} t_2'$.

(2) R^E is *locally coherent modulo* E iff $(\forall t, t_1, t_2 \in T)$ $t \underset{R}{\rightarrow} t_1$ and $t \underset{E}{\rightarrow} t_2$ imply that $(\exists t_1', t_2' \in T)$ $t_1 \underset{R^E}{\rightarrow^*} t_1'$, $t_1 \underset{R^E}{\rightarrow^*} t_2'$, and $t_1' \underset{E}{=} t_2'$.

The relation, $\underset{E}{\rightarrow}$, used in definition 4.9.(2) specifies a rewrite performed using a member of E, rather than a member of R.



(a) Coherence modulo E (b) Local Coherence modulo E

Figure 10. Coherence and Local coherence for E-completion.

An *E-terminating set of reductions, R, modulo E* is a set of reductions, R, for which $\underset{R/E}{\rightarrow}$ has the finite termination property. With these definitions in place, theorem 4.1 can now be extended from complete sets of reductions to E-complete sets of reductions.

Theorem 4.2: The following statements about a set, R, of E-terminating reductions and an equational theory, E, which defines finite equivalence classes are equivalent:.

(1) R is an E-complete set of reductions.

(2) R is R^E-Church-Rosser modulo E.

(3) R^E is confluent modulo E and R^E is coherent modulo E.

(4) R^E is locally confluent modulo E and R^E is locally coherent modulo E .

3. Confluence and Coherence Critical Pairs.

Theorem 4.2 states that a proof of local confluence and local coherence constitutes a decision procedure for the E-completeness of a set of reductions, R. As shown by Peterson and Stickel, local confluence can be proven by generating and successfully conflating all critical pairs of reductions from R, now to be called *confluence critical pairs.* Jouannaud and Kirchner prove that local coherence can likewise be proved by generating and successfully conflating all *coherence critical pairs.* A coherence critical pair is formed from an equation, $l = r \in E \cup E^c$, and a reduction, $\lambda \rightarrow \rho \in R$, and has the form $< \theta(l[i \leftarrow \rho]),\ \theta(r) >$, where $i \in \mathbf{sdom}(l)$, $\theta \in \mathbf{csu}(l/i,\ \lambda)$, and $E^c = \{b = a \mid a = b \in E\}$.

4. Dynamic Extensions.

Peterson and Stickel created an *AC extension* of each member of R with an AC left-hand term to insure the E-compatibility, or coherence, property for the AC completion procedure. Jouannaud and Kirchner introduced a more refined definition called *dynamic extensions;* no extensions are added unless absolutely necessary. If an equation, $l = r$, and a reduction, $\lambda \rightarrow \rho$, fail to *cohere,* that is, one of their coherence critical pairs , $< \theta(l[i \leftarrow \rho]),\ \theta(r) >$, fails to conflate, then an extended reduction of the form $l[i \leftarrow \lambda] \rightarrow l[i \leftarrow \rho]$ is added; the procedure then starts over using the updated set of reductions. The newly added reduction guarantees coherence for the equation and the reduction from which it was formed. This is a better approach than that used by Peterson and Stickel, since fewer reductions added to R mean fewer critical pairs to manipulate.

V. IMPLEMENTATION NOTES ON E-UNIFICATION AND E-COMPLETION.

A. E-UNIFICATION

The E-unification algorithm implemented for this research operates upon terms that are composed of variables, constants, commutative (C) operators, associative-commutative (AC) operators, and associative-commutative-with-identity (ACI) operators. Upon entry into the algorithm, the two terms are assumed to be flattened with respect to associativity and identity. The function E-UNIFY, described in figure 11, is the top level function and the interface to application programs requiring E-unification. In it, some simple analyses are performed on the terms to be unified and, as a result, the terms are passed to the appropriate E-unification "module". Each of these modules may in turn recursively invoke E-UNIFY or some other module to assist in its work. A third input parameter, *PartialUnifier*, is passed along with the terms (or subterms) to be unified. It is a partially constructed unifier that either will be updated at each level of recursion to reflect the successful unification of its accompanying terms, $Term_1$ and $Term_2$, or will be terminated and discarded if the terms cannot be unified without violating the substitutions already in *PartialUnifier*. The initial value of *PartialUnifier* is the identity unifier, that is, the empty set. Upon termination, E-UNIFY or any of the E-unification modules returns a set of unifiers; if this set is empty, then the pair of terms have no unifier. The recursive approach used in this implementation is loosely based on the E-unification model described by Yelick [*Ye*85]. However, Yelick's model was designed only to work with confined, regular equational theories, and ACI theories do not fall into that category.

Figure 12 contains the pseudo-code for a null-E unification module. It is really a recursive version of Robinson's unification algorithm. If the two terms to be unified are non-atomic, that is, they are not variables and not constants, processing

```
E-UNIFY(Term1, Term2, PartialUnifier);
begin
   case
      Term1 =E Term2:
         /* Term1 and Term2 unify by the identity unifier */
         return(PartialUnifier);
      Term1 and Term2 are both atomic terms:
         /* Call upon the recursive Robinson algorithm */
         NULL-E-UNIFY(Term1, Term2, PartialUnifier);
      Term1 is an atomic term:
         if Term2.root ∈ F_ACI
         then ACI-UNIFY(Term1, Term2, PartialUnifier)
         else NULL-E-UNIFY(Term1, Term2, PartialUnifier);
      Term2 is an atomic term:
         /* Reverse the roles of the two terms and come in again */
         E-UNIFY(Term2, Term1, PartialUnifier);
      Term1.root ∈ F_C and Term2.root ∈ F_C:
         C-UNIFY(Term1, Term2, PartialUnifier);
      Term1.root ∈ F_AC and Term2.root ∈ F_AC:
         AC-UNIFY(Term1, Term2, PartialUnifier);
      Term1.root ∈ F_ACI or Term2.root ∈ F_ACI:
         ACI-UNIFY(Term1, Term2, PartialUnifier);
      Otherwise:
         /* All other term combinations are handled as null-E */
         NULL-E-UNIFY(Term1, Term2, PartialUnifier);
   end;
end;
```

Figure 11. The top level function of the recursive E-unification algorithm.

proceeds left-to-right through the operands of the terms, which are pairwise unified through a recursive call to E-UNIFY. The unifiers of each operand pair are used to update the unifiers returned from previous pairs, such that upon completion, the set of unifiers represents those unifiers that will unify all operand pairs, simultaneously.

Siekmann's algorithm [*Si*79], as depicted in figure 13, was implemented to permit the unification of commutative terms. If $Term_1$ and $Term_2$ have the same commutative operator, then unification can be attempted. Commutativity is simulated by generating the set of all terms that are C-equal to $Term_1$ through the permutation of the top-level operands of the term. Then, each of these permuted

```
NULL-E-UNIFY(Term1, Term2, PartialUnifier);
begin
   case
      Term1 = Term2:
         /* Term1 and Term2 unify by the identity unifier */
         return(PartialUnifier);
      Term1 is a variable:
         if Term1 occurs in Term2
         then
            /* Occurs check failure */
            return(EmptySet)
         else return({Term1 ← Term2}∘PartialUnifier);
      Term2 is a variable:
         /* Reverse the roles of the two terms and come in again */
         NULL-E-UNIFY(Term2, Term1, PartialUnifier);
      (Term1 is a constant) or (Term2 is a constant):
         /* If they were equal constants, the first case would have caught it */
         return(EmptySet);
      Term1.root = Term2.root: begin
         FinalPartials := {PartialUnifier};
         /* Pairwise unify the operands of Term1 and Term2*/
         for i := 1 to OPERANDS(Term1) begin
            WorkPartials := EmptySet;
            for σ ∈ FinalPartials
               WorkPartials :=
                  WorkPartials ⋃ E-UNIFY(σ(Term1/i), σ(Term2/i), σ);
            FinalPartials := WorkPartials;
         end;
         return(FinalPartials);
      end;
      Otherwise:
         /* All other cases are terms with different root operators */
         return(EmptySet);
   end;
end;


Notes:
   OPERANDS(Term) returns the number of top-level operands of Term.
```

Figure 12. A recursive null-E unification algorithm.

terms is paired with $Term_2$ and unified as though its common root operator was a null-E operator.

Example 5.1: Let f be a commutative operator and g be a null-E operator. Let

$s = f(w, x, y)$ and

$t = f(a, b, c)$ be terms. Then

$$\mathbf{csu}(s, t) = \mathbf{csu}(s_1, t) \cup \mathbf{csu}(s_2, t) \cup \mathbf{csu}(s_3, t) \cup \mathbf{csu}(s_4, t) \cup \mathbf{csu}(s_5, t) \cup \mathbf{csu}(s_6, t)$$

such that

$s_1 = g(w, x, y)$,

$s_2 = g(w, y, x)$,

$s_3 = g(x, w, y)$,

$s_4 = g(x, y, w)$,

$s_5 = g(y, w, x)$, and

$s_6 = g(y, x, w)$.

```
C-UNIFY(Term1, Term2, PartialUnifier);
begin
   if Term1.root = Term2.root
   then begin
      /* This is a heuristic to speed up C unification */
      if C-OPERATORS(Term1) > C-OPERATORS(Term2)
      then
         /* Swap Term1 and Term2 */
         Term1 ⇔ Term2;
      FinalPartials := EmptySet;
      do t ∈ PERMUTED-TERMS(Term1)
         /* Unify the permuted term and Term2 as null-E terms */
         FinalPartials :=
            FinalPartials ∪ NULL-E-UNIFY(t, Term2, PartialUnifier);
      return(FinalPartials);
   end
   else
      /* Term1 and Term2 have different root operators */
      return(EmptySet);
end;


Notes:
   C-OPERATORS(Term) returns the count of commutative operators at
      all levels within Term.
   PERMUTED-TERMS(Term) returns a list of all permutations of the
      commutative term Term.
```

Figure 13. Siekmann's C unification algorithm.

The AC unification algorithm implemented is not presented here. It is Stickel's general AC unification algorithm, and is described in chapter 3. The ACI unification algorithm implemented is depicted in figure 14. It is a modification of Stickel's AC unification algorithm. Stickel briefly described some of the changes necessary to perform this transformation [*St*75], but two important cases are not discussed: the first is how to proceed when one of the terms has an ACI root operator and the other does not, and the second is how to proceed when the two terms have different ACI root operators. We developed a method for handling both cases, only to discover after further investigation that Fages had mentioned the same method several years earlier [*Fa*84]. The method for the first case, as seen in function ACI-UNIFY2 of figure 14b, entails constructing from the non-ACI term, a term that has the same root operator and the same number of operands as the ACI term. The non-ACI term acts as one of the operands of this new term, and the identity of the ACI operator acts as all other operands. The two ACI terms are then unified by recursively invoking ACI-UNIFY. When unifying two terms with different ACI root operators, the same method is used twice. In each case, one of the ACI terms plays the part of the non-ACI term. The results are then joined. (See statement (1) in figure 14a.) A proof of correctness, completeness, and termination of the ACI unification algorithm is given by Fages in the same paper.

Example 5.2: Let f be an ACI operator with an identity, e, and let g be an AC operator. Let $s = f(w, x, y)$ and $t = g(u, v)$ be terms. Then the set of unifiers for s and t is the set of unifiers for the terms s and t', where

$$t' = f(g(u, v), e, e).$$

This could be represented as

$$t'' = f(g(u, v)).$$

Example 5.3: Let f and g be ACI operators with identities e_1 and e_2, respectively. Let $s = f(w, x, y)$ and $t = g(u, v)$ be terms. Then the set of unifiers for s and t is the set of unifiers for the terms s and t' added to the set of unifiers for the terms s' and t, where

$$s' = g(f(w, x, y), e) = g(f(w, x, y))$$

and

$$t' = f(g(u, v), e, e) = f(g(u, v)).$$

```
ACI-UNIFY(Term1, Term2, PartialUnifier);
begin
   case
      (Term1 is a variable) or (Term2 is a variable):
         return(NULL-E-UNIFY(Term1, Term2,PartialUnifier));
      (Term1 is a constant) or (Term1.root∉F_ACI):
         return(ACI-UNIFY2(Term1, Term2,PartialUnifier));
      (Term2 is a constant) or (Term2.root∉F_ACI):
         /* Switch the roles of Term1 and Term2 */
         return(ACI-UNIFY2(Term2, Term1,PartialUnifier));
      Term1.root ≠ Term2.root:
         return(ACI-UNIFY2(Term1, Term2,PartialUnifier)                (1)
              ⋃ ACI-UNIFY2(Term2, Term1,PartialUnifier));
      Otherwise: begin
         NewTerm1 := ABSTRACT(Term1);
         NewTerm2 := ABSTRACT(Term2);
         AbstractSet := abstraction set from previous two statements
         AbstractUnifiers := ACI-UNIFY-VO(NewTerm1, NewTerm2);
         if AbstractUnifiers exist
         then begin
            NewPartials := {PartialUnifier};
            for σ ∈ AbstractUnifiers
               for x ← t ∈ AbstractSet begin
                  Unifiers := EmptySet;
                  for θ ∈ NewPartials
                     if (σ(x).root is AC) and (σ(t).root is AC)
                     Unifers := Unifiers ⋃ E-UNIFY(σ(x), σ(t), θ)
                  NewPartials := Unifiers;
               end;
            return(NewPartials);
         end
         else
            /* There are no unifiers of Term1 and Term2 */
            return(EmptySet);
      end;
   end;
end;


ABSTRACT(Term) returns a variable abstraction of Term.
ACI-UNIFY-VO(Term1, Term2) is Stickel's variable-only ACI
   unification algorithm.
```

Figure 14a. The ACI-unification algorithm implemented, part 1 of 2.

```
ACI-UNIFY2(Term1, Term2, PartialUnifier);
begin
   /* Term2 is assumed to be an ACI term of the form f(t1, ... , tn) */
   NewTerm := Term2[1 ← Term1];
   for i = 2 to n
      NewTerm := NewTerm[i ← IDENTITY(Term2.root)];
   return(ACI-UNIFY(NewTerm, Term2, PartialUnifier));
end;


Notes:
   IDENTITY(ACIOperator) returns the identity element of ACIOperator.
```

Figure 14b. The ACI-unification algorithm implemented, part 2 of 2.

B. THE E-COMPLETION PROCEDURE

The E-completion procedure implementation used in this research is that developed by Peterson and Stickel [*PS*81]. *Failure-resistance,* as described by Forgaard and Guttag [*FG*84], was added to the procedure to increase its likelihood of success. (Failure-resistance is discussed in chapter 4 of this paper.) The top level E-completion procedure is depicted in figure 15a, and the modified versions of PS-COMPLETION (now called CSR) and ADD-REDUCTION (now called CSR-ADD-REDUCTION) are presented in figure 15b.

Our implementation of the E-completion procedure also incorporated the concept of conditional reductions. A *conditional reduction* is a reduction of the form

$(conditions)\lambda \rightarrow \rho$,

such that (*conditions*) is a set of restrictions, in conjunctive normal form, on the variable values in a term match between λ and any term. Thus, in order to rewrite a term using a conditional reduction, a term match must be found that does not violate the conditions of the reduction. The topic of conditional reductions is outside of the

scope of this paper; Baird gives a detailed presentation on the subject of E-completion procedures involving conditional reductions [*Ba*88].

Example 5.4: Let + be an ACI operator with an identity element, 0, and let − be a null-E operator. Let

$$r = (((x \neq 0) \vee (y \neq 0))) - (+(x, y)) \rightarrow +(-(x), -(y))$$

be a conditional reduction. Any term match, σ, between a term and the left-hand side of r may be used to rewrite the term using r if at least one of x or y is assigned a non-zero value.

```
E-COMPLETION(S, Reductions);
begin
   NewReductions, Shelved := CSR(S, Reductions);
   while (NewReductions ≠ Reductions  do begin
      Reductions := NewReductions;
      NewReductions, Shelved := CSR(Shelved, Reductions);
   end;
   if Shelved = EmptySet
   then return(NewReductions)
   else HALT with FAILURE;
end;
```

Figure 15a. The E-completion procedure implemented, part 1 of 2.

```
CSR(S, Reductions);
begin
   Pairs := EmptySet;
   Eqs := S;
   Shelved := EmptySet;
   while ((Pairs ≠ EmptySet) or (Eqs ≠ EmptySet)) do begin
      if Eqs = EmptySet
      then MAKE-CRITICAL-PAIRS(Pairs, Eqs);
      else begin
         <s, t> := the member of Eqs with the Smallest weight;
         Eqs := Eqs − {<s, t>};
         s_1 := REDUCE*(s, Reductions);
         t_1 := REDUCE*(t, Reductions);
         if s_1 = t_1
         then  /* Successful conflation */
         else begin
            CSR-ADD-REDUCTION(s_1, t_1, Reductions, Pairs, Shelved);
            if <s_1, t_1> was not added to Shelved
            then INTER-REDUCE(Reductions, Pairs);
         end;
      end;
   end;
   return(Reductions, Shelved);
end;


CSR-ADD-REDUCTION(s, t, Reductions, Pairs, Shelved);
begin
   case
      weight(s) > weight(t):
         λ → ρ := s → t;
      weight(s) < weight(t):
         λ → ρ := t → s;
      weight(s) = weight(t):
         Shelved := Shelved ⋃ {<s_1t_1>};
   end;
   Reductions := Reductions ⋃ {λ → ρ};
   for r ∈ Reductions
      Pairs := Pairs⋃ {<λ → ρ, r>, <λ → ρ, r'_AC>,
            <λ'_AC → ρ'_AC, r>, <λ'_AC → ρ'_AC, r'_AC>};
end;


Notes:
   REDUCE*(Term, Reductions) returns an irreducible form of Term,
      with respect to Reductions.
   INTER-REDUCE(Reductions, Pairs) is as described for use by the
      procedure PS-COMPLETION.
```

Figure 15b. The E-completion procedure implemented, part 2 of 2.

VI. TERM SYMMETRY

Experience has shown that a major portion of the time and processing effort required to complete an incomplete set of reductions using an E-completion procedure, such as those discussed in chapter 4, is spent in the calculation and subsequent testing for confluence and coherence of critical pairs. Pruning techniques that remove from consideration those critical pairs that represent redundant or superfluous information, either before, during, or after their calculation, can therefore make a marked difference in the run time and efficiency of an E-completion procedure to which it is applied. These potential savings are, however, dependent upon the efficiency of the pruning technique invoked. If it takes longer to decide that a particular critical pair may be discarded than it would take to process the critical pair, then the pruning technique is probably of little use, other than to reduce the size of the solution space.

In this chapter, a new technique is proposed for removing critical pairs from consideration at various points before, during, or after their formation. This method is based on the property of *term symmetry*, which will be defined and explored with respect to E-unification and E-completion procedures.

A. ALTERNATIVE PRUNING TECHNIQUES

Kapur, Musser, and Narendran [*KM*86] developed and implemented a technique for identifying and discarding redundant critical pairs during the E-completion process. It is based upon earlier work performed by Lankford [*La*75]. In their procedure, the superposition associated with each critical pair is examined in order to decide whether the critical pair should be processed or discarded. They define a *superposition* as a 4-tuple,

$(\lambda_1 \rightarrow \rho_1,\ i,\ \lambda_2 \rightarrow \rho_2,\ \theta)$,

such that $\lambda_1 \rightarrow \rho_1$ and $\lambda_2 \rightarrow \rho_2$ are reductions, and such that $\theta(\lambda_1/i) =_E \theta(\lambda_2)$, that is, $\theta \in \mathbf{csu}(\lambda_1/i, \lambda_2)$. Associated with each superposition is a critical pair of the form $< \theta(\rho_1), \theta(\lambda_1[i \leftarrow \rho_2]) >$.

The bag (multiset) of superpositions for a given pair of reductions can be divided into two classes: composite superpositions and prime superpositions. A *composite superposition* is one for which $\theta(\lambda_2)$, *thatis*, $\theta(\lambda_1/i)$ has a proper reducible subterm. A *prime superposition* is one which is not composite.

Kapur et al. proved that if a superposition is composite, it has an equivalent superposition which can be factored into two prime superpositions with which the original composite superposition can be replaced. They also show that the bag of critical pairs corresponding to the prime superpositions is sufficient for use in an E-completion procedure. This technique decreases the processing time spent reducing critical pairs to terminal form, since the critical pairs corresponding to composite superpositions are discarded before they are simplified. However, no unification time is saved: Complete sets of unifiers must still be generated, and each unifier must still be applied to λ_1 in order to form the superpositions.

A variation of the composite/prime superposition pruning technique identifies and eliminates unblocked superpositions. An *unblocked superposition* is a superposition which contains an unblocked unifier. An *unblocked unifier*, as described by Lankford, is a unifier, $\theta = \{x_1 \leftarrow t_1, \ldots, x_n \leftarrow t_n\}$, in which at least one of the terms $t_1, \ldots, t_n$ is reducible. A unifier in which all right-hand terms are in terminal form is a *blocked unifier*, and the corresponding superposition is a *blocked superposition*.

Every unblocked superposition is also a composite superposition. This is because, if a right-hand term in θ is reducible, and λ_2 is non-trivial (being the left-hand side of a reduction), then $\theta(\lambda_2)$ will also contain that same right-hand term

as a proper, reducible subterm. Thus, unblocked superpositions can be discarded without affecting the results of E-completion.

However, the converse is not true; a blocked superposition may be either composite or prime. Thus, the cardinality of the bag of prime superpositions will be less than or equal to that of the bag of blocked superpositions. This would appear to be an advantage in favor of the composite/prime method. However, one must consider that the unblocked/blocked method has an additional savings in processing; since only the unifier is examined to determine the worth of a superposition, the superposition does not actually have to be constructed, that is, the unifier does not have to be applied to $\lambda_1[i]$ or λ_2.

Unfortunately, Kapur et al. did not give comparisons of the two pruning techniques that they describe. However, they did discuss their implementation and results for the unblocked/blocked technique. When dealing only with null-E operators they found that, in general, the processing time saved by discarding unblocked critical pairs prior to their reduction to terminal form is less than that spent searching for those critical pairs. But, their tests show a significant savings when AC operators are present (as much as 70% savings on total critical pair reduction times, for some examples). They attribute the difference between the null-E and AC cases, at least in part, to the facts that AC unification usually results in multiple mgus (most general unifiers) and AC unification algorithms are not usually minimal (that is, redundant unifiers are present in the complete sets returned by the algorithms).

B. THE DEFINITION OF TERM SYMMETRY

The concept of term symmetry is a simple one. It is based on the realization that variable names used in a term are just symbols acting as placeholders for actual variables, and mapping those symbols to a different set of symbols will not change any aspect of the term, other than the variable names. This is the same idea that permits variables to be renamed in order to assure that terms involved in unification are variable-name disjoint. We begin by defining variable renaming.

Definition 6.1: A *set of variable renaming substitutions* or a *variable renaming* is a set of substitutions,

$$\sigma = \{x_1 \leftarrow y_1, \ \ldots, \ x_n \leftarrow y_n\},$$

which is a one-to-one, onto mapping from the set of variables, $\{x_1, \ \ldots, \ x_n\}$, to the set of variables , $\{y_1, \ \ldots, \ y_n\}$. Any substitution , $x_i \leftarrow y_i$, such that $x_i = y_i$ is an identity substitution and may be dropped from σ. The *identity variable renaming* is the empty set, {}. The application of a variable renaming, σ, to a syntactic entity, t, is written as t^{σ}.

Term symmetry exists between two terms when one can be transformed by a variable renaming into the other. This is stated more formally in definition 6.2.

Definition 6.2: Two terms, s and t, are *symmetric by* σ, written as $s \underset{\sigma}{\approx} t$, if there exists a (possibly empty) variable renaming from **vars**(s) to **vars**(t),

$$\sigma = \{x_1 \leftarrow y_1, \ \ldots, \ x_n \leftarrow y_n\},$$

and its inverse,

$$\sigma^{-1} = \{y_1 \leftarrow x_1, \ \ldots, \ y_n \leftarrow x_n\},$$

such that $s^{\sigma} \underset{E}{=} t$ and $s \underset{E}{=} t^{\sigma^{-1}}$. Such a variable renaming is said to be a *symmetry* of s and t. Two terms for which no symmetry exists are *asymmetric*.

Note that if σ is empty then $s =_E t$. Also, note that if s and t are variable disjoint, as is usually the case, then σ is a match between s and t.

Example 6.1: Let + be a commutative operator (C, AC, or ACI). The two terms $s = +(x_1, x_1, x_2, x_3)$ and $t = +(y_1, y_2, y_2, y_3)$ are symmetric by the variable renamings $\sigma_1 = (x_1 \leftarrow y_2, x_2 \leftarrow y_1, x_3 \leftarrow y_3)$ and $\sigma_2 = (x_1 \leftarrow y_2, x_2 \leftarrow y_3, x_3 \leftarrow y_1)$.

Another form of term symmetry that is of interest is the symmetry which can exist within a single term. Obviously, symmetry within a term is a consequence of the presence of commutative operators.

Definition 6.3: A term, s, is *self-symmetric by* σ, written as $s \underset{\sigma}{\approx} s$, if there exists a variable renaming from **vars**(s) to **vars**(s),

$$\sigma = \{x_1 \leftarrow y_1, \ \ldots, \ x_n \leftarrow y_n\},$$

such that $s^\sigma =_E s$. Such a variable renaming is said to be a *self-symmetry* of term s.

All terms are self-symmetric by the identity variable renaming. Since self-symmetry is a consequence of commutativity, it can only exist (other than the self-symmetry implied by the identity variable renaming) if the term contains one or more commutative operators.

Example 6.2: Let + be a commutative operator (C, AC, or ACI). Then the term $s = +(x_1, x_1, x_2, x_3, x_4)$ is self-symmetric by the variable renamings $\sigma_1 = \{x_2 \leftarrow x_3, x_3 \leftarrow x_2\}$, $\sigma_2 = \{x_2 \leftarrow x_4, x_4 \leftarrow x_2\}$, $\sigma_3 = \{x_3 \leftarrow x_4, x_4 \leftarrow x_3\}$, $\sigma_4 = \{x_2 \leftarrow x_3, x_3 \leftarrow x_4, x_4 \leftarrow x_2\}$, and $\sigma_5 = \{x_2 \leftarrow x_4, x_4 \leftarrow x_3, x_3 \leftarrow x_2\}$.

As illustrated by this example, there can be many self-symmetries within a term. Occasionally, it is desirable to express all self-symmetry relations in a term as one structure, for example, when deciding if a pair of subterms are symmetric with respect to the self-symmetries of their mutual superterm. In order to accomplish this, the variables of a term can be divided up into self-symmetry classes, as described below.

Definition 6.4: The *set of self-symmetry classes* of a term, s, written as ssc(s), is the collection of sets, each of which contains the mutually symmetric variables of s.

Example 6.3: Let + and s be the operator and term, respectively, described in example 6.2. Then $\text{ssc}(s) = \{\{x_1\}, \{x_2, x_3, x_4\}\}$.

This is a concise representation of all self-symmetry relations within a term. The value of ssc(s) is unique for a term s.

The concepts of term symmetry and self-symmetry can be naturally extended to deal with syntactic structures other than terms, such as pairs of terms, sets of substitutions, etc., by viewing such structures as terms.

Example 6.4: Let + be a commutative operator (C, AC, or ACI). Let $< s, t >$ be an unordered pair of the terms, $s = +(x_1, x_1, x_2, x_3)$ and $t = +(y_1, y_2, y_2, y_3)$. An unordered pair may be viewed as a term $f(s, t)$ in which f is a commutative operator not occurring in s or t. Then $\text{ssc}(< s, t >) = \text{ssc}(f(s, t)) = \{\{x_1, y_2\}, \{x_2, x_3, y_1, y_3\}\}$.

C. TERM SYMMETRY IN E-UNIFICATION AND IN E-COMPLETION

There are four types of term symmetry which may be observed in an E-completion procedure: symmetric reductions in the set of reductions being completed by the procedure, symmetric critical pairs, symmetric subterms used in the formation of critical pairs, and symmetric unifiers produced during the formation of critical pairs. The nature of term symmetry suggests that these symmetric syntactic structures may be redundant. If so, it should be possible to derive from the Peterson-Stickel E-completion procedure an *asymmetric E-completion procedure* that produces the same results without processing symmetric redundancies. Such an

asymmetric procedure could result in significant savings in processing if the identification and elimination of term symmetry can be performed efficiently.

It is the goal of this section to show that an asymmetric E-completion procedure can be developed. In order to accomplish this, two points must be proven: first, that symmetry between syntactic structures, such as reductions, critical pairs, subterms, and unifiers, can be detected, and second, that the processing of a set of pairwise symmetric syntactic structures can be replaced by the processing of any one member of the set without changing the results produced by the E-completion procedure.

One method for detecting symmetries between syntactic structures, albeit a very inefficient one, is to generate all matches that exist between the structures. If one of the matches is a variable renaming, there exists a symmetry between the structures. A more efficient algorithm for symmetry detection will be presented later.

Proof of the second point is more involved. It must be proven for each of the four possible types of term symmetry that may be encountered in E-completion. We begin by stating, with respect to term symmetry, two lemmas that are fundamental to automated deduction.

Lemma 6.0.1: If s, s', and t are terms such that $s \underset{\sigma}{\approx} s'$, then $\mathbf{csu}(s, t) \underset{\sigma}{\approx} \mathbf{csu}(s', t)$, that is, $(\forall \theta_1 \in \mathbf{csu}(s, t))\,(\exists \theta_2 \in \mathbf{csu}(s', t))\,\theta_1^{\sigma} = \theta_2$.

Proof: This is just a statement of the fact that renaming the variables in a term to be unified will change the resulting set of unifiers only be the same variable renaming. □

Lemma 6.0.2: If s and t are terms and r is a reduction such that $s \underset{\sigma}{\approx} t$ and $s \underset{r}{\rightarrow} s'$, then $t \underset{r}{\rightarrow} t'$ in such a way that $s' \underset{\sigma}{\approx} t'$.

Proof: In a manner similar to lemma 6.0.1, this is just a statement of the fact that renaming the variables in a term to be rewritten by a reduction will change the result of the rewriting only by the same variable renaming. □

1. Symmetric Reductions.

A reduction, $\lambda \rightarrow \rho$, is an ordered pair of the terms λ and ρ. Two reductions, $\lambda_1 \rightarrow \rho_1$ and $\lambda_2 \rightarrow \rho_1$, are *symmetric reductions* if there exists a variable renaming, σ, such that $\lambda_1 \underset{\sigma}{\approx} \lambda_2$ and $\rho_1 \underset{\sigma}{\approx} \rho_2$. The redundancies introduced into the E-completion procedure by reduction symmetry are removed by the process of inter-reduction simplification.

Inter-reduction simplification is an integral part of the E-completion procedure. Recall that when a new reduction is added to a set of reductions being completed, the two component terms of each reduction in the set are reduced to terminal form using the other reductions in the set. Any reduction reduced to an identity is discarded to preserve the finite termination property. If it reduces to an identity, then any information carried by the reduction must be embodied within the remainder of the set.

To demonstrate how this takes place, consider a member, $\lambda_1 \rightarrow \rho_1$, of the set of reductions that is symmetric by a variable renaming, σ, to a newly added reduction, $\lambda_2 \rightarrow \rho_2$. By the definition of reduction symmetry, $\lambda_1 \underset{\sigma}{\approx} \lambda_2$, or $\lambda_1^{\sigma} \underset{E}{=} \lambda_2$. The variable renaming is, therefore, a term match between λ_1 and λ_2, so $\lambda_1 \rightarrow \rho_1$ can be used to rewrite $\lambda_2 \rightarrow \rho_2$ into a new reduction, $\sigma(\rho_1) \rightarrow \rho_2$. But another consequence of the symmetry of the two reductions by σ is that $\rho_1 \underset{\sigma}{\approx} \rho_2$, or $\rho_1^{\sigma} = \sigma(\rho_1) \underset{E}{=} \rho_2$. Therefore, the new reduction is reduced to an identity and is discarded. Thus, the removal of reduction symmetry already takes place in the E-completion procedure as part of the inter-reduction simplification process.

Example 6.5: Let the set of reductions at some point in an execution of the E-completion procedure be the reductions describing an Abelian group,

r_1: $x + (-x) \rightarrow 0$,

r_2: $-(-x) \rightarrow x$, and

r_3: $-(x+y) \rightarrow (-x)+(-y)$,

such that $+$ is an ACI operator and $-$ is a null-E operator. Let

r_4: $y+(-y) \rightarrow 0$

be a reduction newly added to the set of reductions. It is the case that $r_1 \underset{\sigma}{\approx} r_4$ by $\sigma = \{x \leftarrow y\}$. Thus $\sigma(x+(-x)) = y+(-y)$, and the left-hand side of r_4 can be replaced by $\sigma(0)$, or 0. The reduced form of r_4 is $0 \rightarrow 0$, which is an identity and must be removed from the set of reductions.

2. Symmetric Critical Pairs.

A critical pair, $< s, t >$, is an unordered pair of the terms s and t. Two critical pairs, $< s_1, t_1 >$ and $< s_2, t_2 >$, are *symmetric critical pairs,* written as $< s_1, t_1 > \underset{\sigma}{\approx} < s_2, t_2 >$, if there exists a variable renaming, σ, such that $s_1 \underset{\sigma}{\approx} s_2$ and $t_1 \underset{\sigma}{\approx} t_2$, or $s_1 \underset{\sigma}{\approx} t_2$ and $t_1 \underset{\sigma}{\approx} s_2$. Without loss of generality, we shall assume the former for the duration of this discussion.

Critical pair symmetry is the lowest level of term symmetry in the E-completion procedure, that is, most term symmetries between reductions, subterms used in forming critical pairs, or unifiers will ultimately show up in the form of symmetric critical pairs. Removal of the other three types of term symmetry will result in the elimination of most, but not all, symmetric critical pairs.

In order to eradicate the remaining symmetric critical pairs, and to lay a foundation for use in proving that symmetric subterms and unifiers can be removed, it must be shown that discarding symmetric critical pairs will not change the results of the E-completion process. We shall begin by establishing some basic facts about the terminal forms of terms and critical pairs.

Lemma 6.1.1: If s and t are terms and R is a set of reductions such that $s \underset{\sigma}{\approx} t$, then $(\forall s{\downarrow^R})\,(\exists t{\downarrow^R})\, s{\downarrow^R} \underset{\sigma}{\approx} t{\downarrow^R}$.

Proof: The proof is a consequence of lemma 6.0.2. If $s \approx_{\sigma} t$, then for each sequence of rewrites,

$$s = s_0 \rightarrow s_1 \rightarrow \cdots \rightarrow s_n = s{\downarrow^R},$$

there must also exist a sequence of rewrites,

$$t = t_0 \rightarrow t_1 \rightarrow \cdots \rightarrow t_n = t{\downarrow^R},$$

for $0 \leq n$, such that

$$s_0 \approx_{\sigma} t_0 \Rightarrow s_1 \approx_{\sigma} t_1 \Rightarrow \ldots \Rightarrow s_n \approx_{\sigma} t_n.$$

Therefore, if $s \approx_{\sigma} t$, then $(\forall s{\downarrow^R})\,(\exists t{\downarrow^R})\; s{\downarrow^R} \approx_{\sigma} t{\downarrow^R}$. □

Lemma 6.1.2: If cp_1 and cp_2 are critical pairs such that $cp_1 \approx_{\sigma} cp_2$, then $(\forall cp_1{\downarrow^R})\,(\exists cp_2{\downarrow^R})\; cp_1{\downarrow^R} \approx_{\sigma} cp_2{\downarrow^R}$.

Proof: Let $cp_1 = < s_1,\ t_1 >$ and $cp_2 = < s_2,\ t_2 >$. Assume, without loss of generality, that $s_1 \approx_{\sigma} s_2$ and $t_1 \approx_{\sigma} t_2$. Then, as a consequence of lemma 6.1.1,

$(\forall s_1{\downarrow^R})\,(\exists s_2{\downarrow^R})\; s_1{\downarrow^R} \approx_{\sigma} s_2{\downarrow^R}$ and

$(\forall t_1{\downarrow^R})\,(\exists t_2{\downarrow^R})\; t_1{\downarrow^R} \approx_{\sigma} t_2{\downarrow^R}$.

Pairing these symmetric terminal forms also yields symmetric critical pairs in terminal form,

$$cp_1{\downarrow^R} = < s_1{\downarrow^R},\ t_1{\downarrow^R} > \approx_{\sigma} cp_2{\downarrow}{\downarrow^R} = < s_2{\downarrow^R},\ t_2{\downarrow^R} > .$$

Therefore, if $cp_1 \approx_{\sigma} cp_2$, then $(\forall cp_1{\downarrow^R})\,(\exists cp_2{\downarrow^R})\; cp_1{\downarrow^R} \approx_{\sigma} cp_2{\downarrow^R}$. □

If two symmetric critical pairs truly represent redundant information, then it will be possible to prove that either one of them is sufficient for the proper operation of the E-completion procedure.

Lemma 6.1.3: If cp_1 and cp_2 are critical pairs such that $cp_1 \approx_{\sigma} cp_2$, then either cp_1 or cp_2 may be discarded without changing the results produced by the E-completion procedure.

Proof: If $cp_1 \approx_{\sigma} cp_2$, then it follows from lemma 6.1.2 that $(\forall cp_1{\downarrow^R})\,(\exists cp_2{\downarrow^R})\; cp_1{\downarrow^R} \approx_{\sigma} cp_2{\downarrow^R}$. Let $cp_1 = < s_1,\ t_1 >$, $cp_1{\downarrow} = < s_1{\downarrow},\ t_1{\downarrow} >$,

$cp_2 = < s_2,\ t_2 >$, and $cp_2\downarrow = < s_2\downarrow,\ t_2\downarrow >$. When a critical pair is reduced to terminal form, its two component terminal forms exhibit one of three relationships:

(1) They are provably equal under the equational theory in use.

(2) They are not equal and have different weights.

(3) They are not equal and have identical weights.

In order to accept this lemma, it must be proven for each of these cases.

Case 1: If $s_1\downarrow^R \underset{E}{=} t_1\downarrow^R$, then $(s_1\downarrow^R)^\sigma \underset{E}{=} (t_1\downarrow^R)^\sigma$. Since $s_1\downarrow^R \underset{\sigma}{\approx} s_2\downarrow^R$ and $t_1\downarrow^R \underset{\sigma}{\approx} t_2\downarrow^R$, it follows that $(s_1\downarrow^R)^\sigma \underset{E}{=} s_2\downarrow^R$ and $(t_1\downarrow^R)^\sigma \underset{E}{=} t_2\downarrow^R$, and further that $s_2\downarrow^R \underset{E}{=} t_2\downarrow^R$. Thus, if cp_1 conflates, that is, reduces to an identity, then so will cp_2. Only those critical pairs which do not conflate affect the E-completion procedure, so either cp_1 or cp_2 may be discarded without affecting the results of the procedure.

Case 2: Since $s_1\downarrow^R \underset{\sigma}{\approx} s_2\downarrow^R$ and $t_1\downarrow^R \underset{\sigma}{\approx} t_2\downarrow^R$, it follows that $\mathbf{weight}(s_1\downarrow^R) = \mathbf{weight}(s_2\downarrow^R)$ and $\mathbf{weight}(t_1\downarrow^R) = \mathbf{weight}(t_2\downarrow)^R$ (due to the fact that variables, regardless of their name, have the same weight). Thus if $\mathbf{weight}(s_1\downarrow^R) \neq \mathbf{weight}(t_1\downarrow^R)$, then $\mathbf{weight}(s_2\downarrow^R) \neq \mathbf{weight}(t_2\downarrow^R)$, and the reductions, r_1 and r_2, formed by ordering the terms of $cp_1\downarrow^R$ and $cp_2\downarrow^R$, respectively, will also be symmetric by σ. Thus, if r_1 is added to the set of reductions, and then r_2 is added, the inter-reduction simplification process will remove r_2 from the set of reductions. Reversing the roles of the two reductions leads to the same results. Therefore, processing either cp_1 or cp_2 will produce the same result as processing both critical pairs.

Case 3: As in case 2, $\mathbf{weight}(s_1\downarrow^R) = \mathbf{weight}(s_2\downarrow^R)$ and $\mathbf{weight}(t_1\downarrow^R) = \mathbf{weight}(t_2\downarrow^R)$. Thus, if $\mathbf{weight}(s_1\downarrow^R) = \mathbf{weight}(t_1\downarrow^R)$, then it will also be true that $\mathbf{weight}(s_2\downarrow^R) = \mathbf{weight}(t_2\downarrow^R)$. Since a reduction cannot be formed from a pair of unequal terms with the same weight, both $cp_1\downarrow^R$ and $cp_2\downarrow^R$ will cause the E-completion procedure to fail. Therefore, processing either cp_1 or cp_2 will produce the same result as processing both critical pairs.

Therefore, regardless of the outcome of simplifying the two critical pairs to their terminal forms, either cp_1 or cp_2 may be discarded without changing the results produced by the E-completion procedure. □

This result may be generalized to deal with a set of symmetric reductions, rather than just a pair.

Theorem 6.1: A set of pairwise symmetric critical pairs encountered during the E-completion process may be replaced by any single member of that set without affecting the results of the process.

Proof: Let $\{cp_1, \ldots, cp_n\}$ be a set of pairwise symmetric critical pairs encountered during the E-completion process. Without loss of generality, assume that cp_1 is the critical pair that is to be retained. Since the set is pairwise symmetric, there are $n - 1$ symmetric pairs of critical pairs,

$$< cp_1, cp_2 >, < cp_1, cp_3 >, \ldots, < cp_1, cp_n >,$$

each of which contains cp_1. As a consequence of lemma 6.1.3, cp_i of each pair, $< cp_1, cp_i >$, for $2 \leq i \leq n$, may be discarded, leaving only cp_1. Therefore, a set of pairwise symmetric critical pairs encountered during the E-completion process it may be replaced by any single member of that set without affecting the results of the process. □

3. Symmetric Unifiers.

As shown in the previous section, symmetric critical pairs may be discarded without affecting the results of the E-completion procedure. However, creating critical pairs which are then thrown out is a waste of processing time: Unifiers must be generated and applied to form these unneeded critical pairs. A better approach is to search for symmetric redundancies and to remove them from the components from which the critical pairs are built before much processing effort has been expended.

One of the components that can be examined for term symmetry is the unifier associated with each critical pair. We would like to show that discarding symmetric unifiers has no effect on the results of the E-completion procedure. In order to prove this, it must be shown that symmetric unifiers produce symmetric critical pairs.

Definition 6.5: Let s and s' be terms. Assume, without loss of generality, that s and s' are variable disjoint. Two unifiers, θ_1, $\theta_2 \in \mathbf{csu}(s, s')$, are *symmetric unifiers*, written as $\theta_1 \underset{\sigma}{\approx} \theta_2$, if there exists a variable renaming, σ, such that $\theta_1^\sigma \underset{E}{=} \theta_2$, and, for all terms, t, to which θ_1 and θ_2 will be applied, $t \underset{\sigma}{\approx} t$ and $\theta_1(t) \underset{\sigma}{\approx} \theta_2(t)$.

The definition of symmetric unifiers is more complicated than those of symmetric critical pairs and symmetric terms. In fact, the final condition of the definition, that is, the requirement that for all terms t to which the unifiers will be applied $\theta_1(t) \underset{\sigma}{\approx} \theta_2(t)$, seems to be self-defeating: Checking this condition for a given value of σ requires the application of θ_1 and θ_2 to a term, which is exactly the process that detecting and discarding symmetric unifiers is supposed to eliminate. However, there is a way to show that any variable renaming that meets the first two conditions of the definition will meet the third condition.

Lemma 6.2.1: Let s and s' be variable disjoint terms. If $\theta_1, \theta_2 \in \mathbf{csu}(s, s')$ such that $\theta_1^\sigma \underset{E}{=} \theta_2$, and there exists a term, t, such that $t \underset{\sigma}{\approx} t$, then $\theta_1(t) \underset{\sigma}{\approx} \theta_2(t)$.

Proof: By definition, $t \underset{\sigma}{\approx} t$ implies that $t^\sigma \underset{E}{=} t$. Since $\theta_1^\sigma \underset{E}{=} \theta_2$, it follows that $\theta_2(t) \underset{E}{=} \theta_1^\sigma(t^\sigma)$. If it can be proven that $\theta_1^\sigma(t^\sigma) \underset{E}{=} (\theta_1(t))^\sigma$, then by transitivity, $\theta_2(t) \underset{E}{=} (\theta_1(t))^\sigma$, which is the definition of $\theta_1(t) \underset{\sigma}{\approx} \theta_2(t)$.

Assume that $\theta_1 = \{x_0 \leftarrow s_0, \ \ldots, \ x_n \leftarrow s_n\}$ and $\theta_1^\sigma = \{x_0^\sigma \leftarrow s_0^\sigma, \ \ldots, \ x_n^\sigma \leftarrow s_n^\sigma\}$. The nodes of the tree representation of t each fall into one of three categories:

(1) operators, including constants,

(2) variables, x_i, for $0 \leq i \leq n$, and

(3) variables, x_i, for $i > n$.

Only categories (2) and (3) need to be examined, since operators are not affected by substitutions. In the tree for $\theta_1(t)$:

(1) If $0 \leq i \leq n$, then x_i is replaced by s_i.

(2) If $i > n$, then x_i remains the same.

So, in the tree for $(\theta_1(t))^\sigma$:

(1) if $0 \leq i \leq n$, then x_i is replaced by s_i^σ.

(2) if $i > n$, then x_i is replaced by x_i^σ.

In the tree for t^σ:

(1) if $0 \leq i \leq n$, then x_i is replaced by x_i^σ.

(2) if $i > n$, then x_i is replaced by x_i^σ.

And, in the tree for $\theta_1^\sigma(t^\sigma)$:

(1) if $0 \leq i \leq n$, then x_i is replaced by s_i^σ.

(2) if $i > n$, then x_i is replaced by x_i^σ.

Thus, it can be seen that the tree representations of $(\theta_1(t))^\sigma$ and $\theta_1^\sigma(t^\sigma)$ are the same, and so $(\theta_1(t))^\sigma \underset{E}{=} \theta_1^\sigma(t^\sigma)$. Therefore, if $\theta_1^\sigma \underset{E}{=} \theta_2$ and $t \underset{\sigma}{\approx} t$, then $\theta_1(t) \underset{\sigma}{\approx} \theta_2(t)$. □

As will be shown in the proof of the following lemma, one result of lemma 6.2.1 is that the critical pairs produced by a pair of symmetric unifiers are also symmetric.

Lemma 6.2.2: Let $\lambda_1 \rightarrow \rho_1$ and $\lambda_2 \rightarrow \rho_2$ be reductions. If $\theta_1, \theta_2 \in \mathbf{csu}(\lambda_1/i, \lambda_2)$ such that $\theta_1 \underset{\sigma}{\approx} \theta_2$, then either θ_1 or θ_2 may be discarded without affecting the results of the E-completion procedure.

Proof: If $\theta_1 \underset{\sigma}{\approx} \theta_2$, then by the definition of symmetric unifiers $\theta_1^\sigma \underset{E}{=} \theta_2$, $\rho_1 \underset{\sigma}{\approx} \rho_1$, and $\lambda_1[i \leftarrow \rho_2] \underset{\sigma}{\approx} \lambda_1[i \leftarrow \rho_2]$. (The latter two terms are those to which θ_1 and θ_2 are applied to form critical pairs.) It then follows from lemma 6.2.1 that $\theta_1(\rho_1) \underset{\sigma}{\approx} \theta_2(\rho_1)$ and $\theta_1(\lambda_1[i \leftarrow \rho_2]) \underset{\sigma}{\approx} \theta_2(\lambda_1[i \leftarrow \rho_2])$. Thus, the critical pairs, $< \theta_1(\rho_1), \theta_1(\lambda_1[i \leftarrow \rho_2]) >$ and $< \theta_2(\rho_1), \theta_2(\lambda_1[i \leftarrow \rho_2]) >$, are also symmetric by σ. By theorem 6.1, either of these critical pairs may be safely discarded. Therefore, if $\theta_1, \theta_2 \in \mathbf{csu}(\lambda_1/i, \lambda_2)$ and

$\theta_1 \underset{\sigma}{\approx} \theta_2$, then either θ_1 or θ_2 may be discarded without affecting the results of the E-completion procedure. □

This result can be generalized to deal with sets of pairwise symmetric unifiers, just as lemma 6.1.3 was generalized to theorem 6.1.

Theorem 6.2: Let $\lambda_1 \rightarrow \rho_1$ and $\lambda_2 \rightarrow \rho_2$ be reductions. A pairwise symmetric subset of $\mathbf{csu}(\lambda_1/i, \lambda_2)$, for $i \in \mathbf{sdom}(\lambda_1)$, encountered during the E-completion process may be replaced by any single member of that set without affecting the results of the process.

Proof: The proof of this theorem proceeds like that of theorem 6.1. □

4. Symmetric Subterms.

Another component of the critical pair that can be examined for term symmetry is the subterm chosen from the left-hand side of a reduction.

Lemma 6.0.2 states that if s and t are terms and $r = \lambda \rightarrow \rho$ is a reduction, such that $s \underset{\sigma}{\approx} t$ and $s \underset{r}{\rightarrow} s'$, then $t \underset{r}{\rightarrow} t'$ such that $s' \underset{\sigma}{\approx} t'$. Since σ is merely a variable renaming, it follows that there must exist an $i \in \mathbf{dom}(s)$ and a $j \in \mathbf{dom}(t)$ such that $(s/i)^{\sigma} \underset{E}{=} t/j$, s/i matches λ by θ_i, t/j matches λ by θ_j, $s' = s[i \leftarrow \theta_i(\rho)]$, and $t' = t[j \leftarrow \theta_j(\rho)]$.

Now consider the case of $s \underset{\sigma}{\approx} s$, such that $(s/i)^{\sigma} \underset{E}{=} s/j$ and $i \neq j$, for some $i, j \in \mathbf{dom}(s)$. If s/i matches λ by θ_i and s/j matches λ by θ_j, then is it true that $s[i \leftarrow \theta_i(\rho)] \underset{\sigma}{\approx} s[j \leftarrow \theta_j(\rho)]$? If s/i and s/j are rooted at different depths in the term tree of s, the two subterms cannot be considered symmetric. They are also not symmetric if they are sibling operands of a common non-commutative operator. If s/i and s/j are in distinct subtrees of s, then they can only be symmetric if the subtrees in which they appear are symmetric. Thus, the determination of symmetry is pushed upward in the tree to the level at which the two subtrees have a common parent

node, and once again becomes a matter of determining the symmetry of sibling operands. This leads to a definition of symmetric subterms.

Definition 6.6: Let s be a term. Two subterms, s/i and s/j, are *symmetric subterms of* s, written as $s/i \underset{\sigma}{\approx} s/j$, if there exists a variable renaming σ such that $(s/i)^{\sigma} \underset{E}{=} s/j$, $s \underset{\sigma}{\approx} s$, and s/i and s/j are sibling operands of a common commutative (C, AC, or ACI) operator.

This definition must be modified slightly to be used with subterms of the left-hand side of a reduction. If $r = \lambda \rightarrow \rho$ is a reduction, then two subterms λ/i and λ/j are symmetric by σ if $(\lambda/i)^{\sigma} \underset{E}{=} \lambda/j$, $r \underset{\sigma}{\approx} r$, and λ/i and λ/j are sibling operators of a common commutative operator. The reason that $r \underset{\sigma}{\approx} r$ is required in place of $\lambda \underset{\sigma}{\approx} \lambda$ is that we want to show that symmetric subterms of λ produce symmetric critical pairs, but both λ and ρ are used in forming critical pairs.

Lemma 6.3.1: Let $\lambda_1 \rightarrow \rho_1$ and $\lambda_2 \rightarrow \rho_2$ be reductions. If $\lambda_1/i \underset{\sigma}{\approx} \lambda_1/j$, such that $i, j \in \mathbf{sdom}(\lambda_1)$, then either λ_1/i or λ_1/j may be disregarded without affecting the results of the E-completion procedure.

Proof: Without loss of generality, assume that the two reductions are variable disjoint. Lemma 6.0.1 states that if $\lambda_1/i \underset{\sigma}{\approx} \lambda_1/j$, then $(\forall \theta_i \in \mathbf{csu}(\lambda_1/i,\ \lambda_2))\ (\exists \theta_j \in \mathbf{csu}(\lambda_1/j,\ \lambda_2))\ \theta_i^{\sigma} \underset{E}{=} \theta_j$. Without loss of generality, we shall assume such a θ_i and its corresponding θ_j in the remainder of the proof.

λ_1/i and λ_1/j produce critical pairs, $< \theta_i(\rho_1),\ \theta_i(\lambda_1[i \leftarrow \rho_2]) >$ and $< \theta_j(\rho_1),\ \theta_j(\lambda_1[j \leftarrow \rho_2]) >$, respectively. By the definition of symmetric subterms, $\lambda_1 \rightarrow \rho_1 \underset{\sigma}{\approx} \lambda_1 \rightarrow \rho_1$ and, thus, $\rho_1 \underset{\sigma}{\approx} \rho_1$. Since $\theta_i^{\sigma} \underset{E}{=} \theta_j$, it follows from lemma 6.2.1 that $\theta_i(\rho_1) \underset{\sigma}{\approx} \theta_j(\rho_1)$. But in order for the critical pairs to be symmetric by σ, it must also be true that $\theta_i(\lambda_1[i \leftarrow \rho_2]) \underset{\sigma}{\approx} \theta_j(\lambda_1[j \leftarrow \rho_2])$.

By viewing a pair of symmetric terms as trees, it can be seen that replacing a symmetric subterm in each of the pair by a subterm that is also symmetric yields a new pair of symmetric terms. Since the two reductions are variable disjoint, and σ is a variable renaming from **vars**(λ_1/i) to **vars**(λ_1/j), it follows that $\rho_2 \underset{\sigma}{\approx} \rho_2$. Thus, $\lambda_1[i \leftarrow \rho_2] \underset{\sigma}{\approx} \lambda_1[j \leftarrow \rho_2]$, that is, $(\lambda_1[i \leftarrow \rho_2])^\sigma \underset{E}{=} \lambda_1[j \leftarrow \rho_2]$ and, consequently, $\theta_j(\lambda_1[j \leftarrow \rho_2]) \underset{E}{=} \theta_i^\sigma((\lambda_1[i \leftarrow \rho_2])^\sigma)$. So, if it can be proven that $\theta_i^\sigma((\lambda_1[i \leftarrow \rho_2])^\sigma) \underset{E}{=} (\theta_i(\lambda_1[i \leftarrow \rho_2]))^\sigma$, then it must also be true that $\theta_j(\lambda_1[j \leftarrow \rho_2]) \underset{E}{=} (\theta_i(\lambda[i \leftarrow \rho_2]))^\sigma$, that is $\theta_i(\lambda_1[i \leftarrow \rho_2])^\sigma \underset{E}{=} \theta_j(\lambda_1[j \leftarrow \rho_2])$.

Assuming that $t = \lambda_1[i \leftarrow \rho_2]$ makes this a proof of $\theta_i^\sigma(t^\sigma) \underset{E}{=} (\theta_i(t))^\sigma$, which was proven as part of the proof of lemma 6.2.1. Thus, $\theta_i(\lambda_1[i \leftarrow \rho_2]) \underset{\sigma}{\approx} \theta_j(\lambda_1[j \leftarrow \rho_2])$, and the critical pairs produced by λ_1/i and λ_1/j are symmetric. It follows from theorem 6.1 that either of these symmetric critical pairs may be discarded without affecting the results of the E-completion procedure.

This result can be observed for each symmetric pair of unifiers from **csu**$(\lambda_1/i, \lambda_2)$ and **csu**$(\lambda_1/j, \lambda_2)$. Therefore, if $\lambda_1/i \underset{\sigma}{\approx} \lambda_1/j$, then either λ_1/i or λ_1/j may be disregarded without affecting the results of the E-completion procedure. □

This lemma can be generalized to handle sets of pairwise symmetric subterms, much as lemma 6.1.3 was generalized to theorem 6.1.

Theorem 6.3: Let $\lambda_1 \rightarrow \rho_1$ and $\lambda_2 \rightarrow \rho_2$ be reductions. The processing of a set of pairwise symmetric subterms of λ_1 encountered during the E-completion process may be replaced by that of any single member of the set without affecting the results of the process.

Proof: The proof of this theorem proceeds like that of theorem 6.1. □

D. TERM SYMMETRY ALGORITHMS

1. A Term Symmetry Decision Algorithm.

The algorithm developed in this section is a decision procedure for the symmetry of a pair of terms composed of commutative operators, null-E operators, constants, and variables. It can also be used to decide the symmetry of terms involving AC and ACI operators if those terms have been simplified to normal form, that is, the terms have been flattened and have had all identities removed through simplification.

The term symmetry decision algorithm is similar in concept to the tree isomorphism decision algorithm presented by Aho, Hopcroft, and Ullman [*AH*74]. Their algorithm ignores all node labels in its operation. Unfortunately, this fact makes it inappropriate for use in deciding term symmetry, because for terms to be symmetric, constants must map onto identical constants and variables must map onto variables. An extension of the tree isomorphism decision algorithm is also suggested by Aho et al. to handle node labels. However, it, too, cannot be used to decide term symmetry, since the extension requires that variables map onto identical variables. In addition, neither of these algorithms consider the possible presence of null-E operators along with the commutative operators in the tree.

The pseudo-code for the term symmetry decision algorithm is contained in figure 16. If $Term_1$ and $Term_2$ are symmetric terms, SYMMETRIC? returns a symmetry, σ. Otherwise, it returns a value of FALSE. The actual implementation of this algorithm can be made more efficient by the application of constraints. For example, comparing the sizes of **vars**($Term_1$) and **vars**($Term_2$) before calling BUILD-TERM-BAG could save unnecessary processing, since a difference in these sizes means that $Term_1$ and $Term_2$ are definitely not symmetric.

The terms input to SYMMETRIC? are passed successively into the function BUILD-TERM-BAG. This function constructs a bag, or multiset, of terms from its input parameter, *Term*. The term bag contains exactly one new term for each distinct variable, x_i, in *Term*. This new term is a copy of *Term* in which all occurrences of x_i have been replaced by the constant c_1, and all other variable occurrences have been replaced by the constant c_2. These are *new* constants, that is, c_1 and c_2 do not appear in $Term_1$ or $Term_2$, input to function SYMMETRIC? Associated with each new term is x_i, the variable that was replaced by c_1. (See statements (2) and (3) in the pseudo-code.) If *Term* is ground, that is, contains no variables, then the term bag returned is empty.

Once the term bags for $Term_1$ and $Term_2$ have been constructed, they are compared to decide whether or not the two input terms are symmetric. If the term bags are both empty, that is, both $Term_1$ and $Term_2$ are ground, then $Term_1$ and $Term_2$ are each sorted with respect to their commutative operators, that is, only the operands of commutative operators are sorted. Then the sorted terms are compared. If they are equal, then $Term_1$ and $Term_2$ are symmetric by the identity symmetry, $\sigma = \{\}$. If unequal, the two terms are not symmetric, and a value of FALSE is returned.

On the other hand, if either of the term bags is non-empty, then each term in both term bags is sorted with respect to commutativity, and then each term bag is sorted. If the two sorted term bags are equal, then there is a one-to-one, onto mapping from each term in $TermBag_1$ to an equivalent term in $TermBag_2$. A term bag contains exactly one term for each variable in the term from which it was constructed, and each variable is associated with exactly one member of its term bag. Thus, the mapping from $TermBag_1$ to $TermBag_2$ can, and is, used to construct a one-to-one, onto mapping from **vars**($Term_1$) to **vars**($Term_2$). (See statement (6) in the pseudo-code.) This mapping is returned as a symmetry of $Term_1$ and $Term_2$.

If the two sorted term bags are not equal, then $Term_1$ and $Term_2$ are not symmetric, and a value of FALSE is returned.

```
BUILD-TERM-BAG(Term);
begin
   Vars := the set of variables occurring in Term, {x_1, ..., x_n};          (1)
   TermBag := EmptyBag;
   for x_i ∈ Vars begin
      σ_i := {x_1 ← c_2, ..., x_{i-1} ← c_2, x_i ← c_1, x_{i+1} ← c_2, ..., x_n ← c_2};   (2)
      TermBag := TermBag + σ_i(Term):x_i;                                    (3)
   end;
   return(TermBag);
end;


SYMMETRIC?(Term_1, Term_2);
/* Term_1 and Term_2 are assumed to be in normal form. */
begin
   TermBag_1 := BUILD-TERM-BAG(Term_1);
   TermBag_2 := BUILD-TERM-BAG(Term_2);
   if (TermBag_1 is empty) and (TermBag_2 is empty)
   then    /* Both Term_1 and Term_2 are ground terms. */
      if COMM-SORT(Term_1) = COMM-SORT(Term_2)                               (4)
      then return({})
      else return(FALSE)
   else    /* Term_1, Term_2, or both terms contain variables. */
      if SORT-BAG(TermBag_1) = SORT-BAG(TermBag_2)                           (5)
      then begin
         σ: = EmptySet;
         for (t_1:x_i ∈ TermBag_1) and (t_2:y_i ∈ TermBag_2)
            σ := σ ⋃ {x_i ← y_i};                                            (6)
         return(σ);
      end
      else return(FALSE);
end;


Notes:
   COMM-SORT(Term) recursively sorts the operands of the commutative
      operators of Term.
   SORT-BAG(TermBag) uses COMM-SORT to sort each term in TermBag,
      then sorts TermBag.
```

Figure 16. An algorithm to decide if two terms are symmetric.

It can be seen in figure 16 that SYMMETRIC?($Term_1$, $Term_2$) is an algorithm. There are a finite number of distinct variables in each of $Term_1$ and $Term_2$, thus BUILD-TERM-BAG will halt for each. Also, since SYMMETRIC? contains no loops, it will halt. The correctness of the algorithm, however, is not as simply shown.

Theorem 6.4: The function SYMMETRIC?($Term_1$, $Term_2$) returns a symmetry, σ, iff $Term_1 \underset{\sigma}{\approx} Term_2$.

Proof that SYMMETRIC?($Term_1$, $Term_2$) returns $\sigma \Rightarrow Term_1 \underset{\sigma}{\approx} Term_2$: There are two cases for which SYMMETRIC? returns a symmetry σ:

(1) $TermBag_1$ and $TermBag_2$ are empty, and $Term_1 \underset{E}{=} Term_2$.

(2) $TermBag_1$ and $TermBag_2$ are not empty, and $TermBag_1 \underset{E}{=} TermBag_2$.

Case 1: A term bag created by BUILD-TERM-BAG contains exactly one term for each distinct variable in **vars**($Term$). Thus, $TermBag_1$ and $TermBag_2$ can only be empty if both $Term_1$ and $Term_2$ are ground terms. Consider that if $\sigma = \{\}$, then $Term_1 \underset{E}{=} \sigma(Term_1)$. Since $Term_1 \underset{E}{=} Term_2$, it is a consequence of transitivity that $\sigma(Term_1) \underset{E}{=} Term_2$. Therefore, $Term_1 \underset{\sigma}{\approx} Term_2$.

Case 2: The following refers to the relationships illustrated in figure 17. Let $\psi = \{\psi_{x1}, \ldots, \psi_{xn}\}$ be a set of one-to-one, onto mappings defined such that, for **vars**($Term_1$) = $\{x_1, \ldots, x_n\}$, the mappings are $\psi_{x1}(Term_1) \equiv \{x_1 \leftarrow c_1\}(Term_1)$, ..., $\psi_{xn}(Term_1) \equiv \{x_n \leftarrow c_1\}(Term_1)$, for some distinguished constant c_1.

In a similar manner let $\omega = \{\omega_{y1}, \ldots, \omega_{yn}\}$ be a set of one-to-one, onto mappings defined such that, for **vars**($Term_2$) = $\{y_1, \ldots, y_n\}$, the mappings are $\omega_{y1}(Term_1) \equiv \{y_1 \leftarrow c_1\}(Term_1)$, ..., $\omega_{ym}(Term_1) \equiv \{y_n \leftarrow c_1\}(Term_1)$, for the same constant c_1. Thus, there is a set of inverse relations, $\omega^{-1} = \{\omega_{y1}^{-1}, \ldots, \omega_{ym}^{-1}\}$ that maps elements $\omega_{yj}(Term_2)$ back onto $Term_2$.

If all other variables remaining in these terms are viewed as identical distinguished constants other than c_1, then the effect of ψ and ω on $Term_1$ and $Term_2$, respectively, is the same as that of the function BUILD-TERM-BAG. Since $TermBag_1$ and $TermBag_2$ have the same number of elements, there exists a one-to-one, onto mapping, η, from $TermBag_1$ to $TermBag_2$.

It can be seen that ψ and ω^{-1} preserve the structure of the terms to which they apply. In addition, since $TermBag_1 \underset{E}{=} TermBag_2$, η is also a structure preserving mapping. So, we can define a set of structure preserving, one-to-one, onto mappings,

$$\hat{\sigma} = \{\hat{\sigma}_{ij} \mid \hat{\sigma}_{ij} = \omega_{yj}^{-1} \circ \eta \circ \psi_{xi}\},$$

from the variables of $Term_1$ to the variables of $Term_2$, where

$$\omega_{yj}^{-1} \circ \eta \circ \psi_{xi}(Term_1) \equiv \omega_{yj}^{-1}(\eta(\psi_{xi}(Term_1)))$$

is the *composition* of functions ω_{yj}^{-1}, η, and ψ_{xi}. The set of mappings $\hat{\sigma}$ is equivalent to the symmetry returned by the function SYMMETRIC?

If, however, $TermBag_1 \underset{E}{\neq} TermBag_2$, then η is not structure preserving, and no structure preserving mapping $\hat{\sigma}$ exists, so there is no symmetry from $Term_1$ to $Term_2$. Therefore, if SYMMETRIC? returns a symmetry, σ, then $Term_1 \underset{\sigma}{\approx} Term_2$.

Proof that $Term_1 \underset{\sigma}{\approx} Term_2 \Rightarrow$ SYMMETRIC?($Term_1$, $Term_2$) returns σ: By the definition of term symmetry, if $Term_1 \underset{\sigma}{\approx} Term_2$ they must have the same number of variables. So, there are two cases to be considered:

(1) $Term_1$ and $Term_2$ are ground terms.

(2) $Term_1$ and $Term_2$ contain variables.

Case 1: Since $Term_1$ and $Term_2$ are ground terms, $TermBag_1$ and $TermBag_2$, produced by BUILD-TERM-BAG, will be empty. In addition, $Term_1$ can only be symmetric to $Term_2$ by the symmetry $\sigma = \{\}$. Since $Term_1 \underset{E}{=} \sigma(Term_1)$ and $\sigma(Term_1) \underset{E}{=} Term_2$, it is a consequence of transitivity that $Term_1 \underset{E}{=} Term_2$. Therefore, SYMETRIC?($Term_1$, $Term_2$) returns $\sigma = \{\}$.

Case 2 (proof by contradiction): Since $Term_1$ and $Term_2$ contain variables, both $TermBag_1$ and $TermBag_2$, produced by BUILD-TERM-BAG, will be non-empty. As stated earlier, $Term_1$ and $Term_2$ must contain the same number of variables, so $TermBag_1$ contains the same number of terms as $TermBag_2$. Assuming that SYMMETRIC? returns FALSE, it must be the case that $(\exists t_2 \in TermBag_2)\,(\forall t_1 \in TermBag_1)\; t_2 \neq_E t_1$. It can be seen in the pseudo-code of BUILD-TERM-BAG that

$$TermBag_1 = \{\sigma_{11}(Term_1),\ \ldots,\ \sigma_{1n}(Term_1)\} \text{ and}$$

$$TermBag_2 = \{\sigma_{21}(Term_2),\ \ldots,\ \sigma_{2n}(Term_2)\},$$

where, for all $1 \leq i \leq n$, $x_i \in \mathbf{vars}(Term_1)$, and $y_i \in \mathbf{vars}(Term_2)$,

$$\sigma_{1i} = \{x_1 \leftarrow c_2,\ \ldots,\ x_{i-1} \leftarrow c_2,\ x_i \leftarrow c_1,\ x_{i+1} \leftarrow c_2,\ \ldots,\ x_n \leftarrow c_2\} \text{ and}$$

$$\sigma_{2i} = \{y_1 \leftarrow c_2,\ \ldots,\ y_{i-1} \leftarrow c_2,\ y_i \leftarrow c_1,\ y_{i+1} \leftarrow c_2,\ \ldots,\ y_n \leftarrow c_2\}.$$

Since $Term_1$ is symmetric to $Term_2$, there exists a variable renaming, $\sigma = \{x_1 \leftarrow y_1,\ \ldots,\ x_n \leftarrow y_n\}$, such that $\sigma(Term_1) =_E Term_2$, and consequently $\sigma_{2i}(\sigma(Term_1)) =_E \sigma_{2i}(Term_2)$. Assume, without loss of generality, that $Term_1$ and $Term_2$ are variable disjoint. Then by the definition of the composition of substitutions,

$$(\forall\, 1 \leq i \leq n)\ \sigma_{2i}(\sigma(Term_1)) =_E \sigma_{2i} \circ \sigma(Term_1),$$

where

$$\sigma_{2i} \circ \sigma = \{x_1 \leftarrow c_2, \ldots, x_{i-1} \leftarrow c_2, x_i \leftarrow c_1, x_{i+1}, \leftarrow c_2, \ldots, x_n \leftarrow c_2\}$$
$$\cup\ \{y_1 \leftarrow c_2, \ldots, y_{i-1} \leftarrow c_2, y_i \leftarrow c_1, y_{i+1}, \leftarrow c_2, \ldots, y_n \leftarrow c_2\}.$$

Since $Term_1$ and $Term_2$ are variable disjoint, it is clear that the application of $\sigma_{2i} \circ \sigma$ to $Term_1$ as described, above, will have the same affect as the application of σ_{1i} to $Term_1$, that is,

$$(\forall\, 1 \leq i \leq n)\ \sigma_{2i} \circ \sigma(Term_1) =_E \sigma_{1i}(Term_1).$$

Thus, as a consequence of transitivity,

$$(\forall\, 1 \leq i \leq n)\ \sigma_{2i}(Term_2) =_E \sigma_{1i}(Term_1).$$

This means that $(\forall t_2 \in TermBag_2)\,(\exists t_1 \in TermBag_1)\; t_2 =_E t_1$, which implies that $TermBag_1 =_E TermBag_2$ and, consequently, that the function SYMMETRIC? returns

$\sigma = \{x_1 \leftarrow y_1, \ \ldots, \ x_n \leftarrow y_n\}$. This is a contradiction of the assumption that SYMMETRIC? returns FALSE. Therefore if $Term_1 \approx_{\sigma} Term_2$, then SYMMETRIC?($Term_1$, $Term_2$) returns a symmetry σ. □

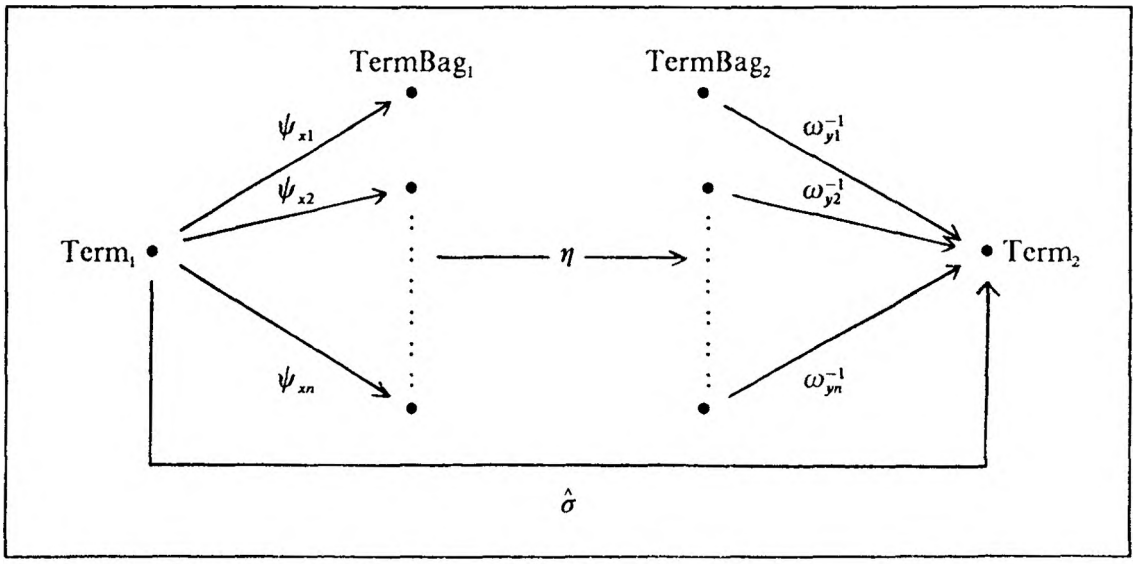


Figure 17. Mappings between a pair of symmetric terms.

The steps in this algorithm which comprise most of the processing time have been labelled in figure 16. A worst-case time complexity analysis on each of these steps reveals the following, in which n is assumed to be the maximum of the number of nodes in either the term tree for $Term_1$ or the term tree for $Term_2$:

(1) find all variables in *Term*--O(n),

(2) for each distinct variable, build a substitution--O(n),

(3) for each distinct variable, build a new term--O(n^2),

(4) sort $Term_1$ and $Term_2$ at all levels--O($n^2 \log n$),

(5) sort and compare the term bags at all levels--O($n^2 \log n$) + O(n^2), and

(6) build the symmetry to be returned--O(n).

Thus, the worst-case time complexity for this algorithm is O($n^2 \log n$).

The worst-case space complexity for SYMMETRY? is $O(n^2)$, since there are, at most, n copies of a term made for each of the n nodes in the term.

2. An Algorithm for Finding Asymmetric Subterms (Strict Domains).

Figure 18 contains the pseudo-code for an algorithm to prune the strict domain (**sdom**) of a term down to an asymmetric strict domain (**asdom**). This is an extension of the basic term symmetry decision algorithm. The function BUILD-TERM-BAG2 produces a bag of extended terms. Each term is concatenated with the term associated with the same variable contained in the term bag constructed for the parent term. The concatenated terms for one subterm will equal the concatenated terms for another only if the variables associated with the concatenated terms are symmetric with respect to both the subterm and the parent term. The function ASYMM-SUBTERMS is the recursive part of this algorithm. When a term is input as an argument into ASYMM-SUBTERMS, its position within the top level term is also provided. At the top level, this position is ε, which is subsequently appended to at each level of recursion. (See statement (1).) Note that an altered version of the procedure MAKE-CRITICAL-PAIRS, which was described in chapter 4, is also included.

```
BUILD-TERM-BAG2(Term, SuperBag);
begin
   Vars := the set of variables occurring in Term, {x_1, ..., x_n};
   TermBag := EmptyBag;
   for x_i ∈ Vars begin
      σ_i := {x_1 ← c_2, ..., x_{i-1} ← c_2, x_i ← c_1, x_{i+1} ← c_2, ..., x_n ← c_2};
      NewTerm_1 := σ_i(Term);
      NewTerm_2 := the term from SuperBag that corresponds to x,
         or if no such term exists, EmptyTerm;
      TermBag :=
         TermBag + CONCAT(COMM-SORT(NewTerm_1), NewTerm_2);
   end;
   return(TermBag);
end;


ASDOM(Term);
begin
   TermBag := SORT-BAG(BUILD-TERM-BAG(Term));
   return(ASYMM-SUBTERMS(Term, TermBag, ε);                          (1)
end;


MAKE-CRITICAL-PAIRS(Pairs, Eqs);
begin
   {λ_1 → ρ_1, λ_2 → ρ_2} := the member of Pairs with the smallest value of
      weight(λ_1) + weight(λ_2);
   Pairs := Pairs − {λ_1 → ρ_1, λ_2 → ρ_2};
   Eqs := { < σ(ρ_1), σ(ρ_2) > | σ ∈ csu(λ_1, λ_2)}
      ∪ { < σ(ρ_1), σ(λ_1[i ← ρ_2]) > | λ_1 → ρ_1 is not an extension   (2)
         ∧ i ∈ ASDOM(λ_1) ∧ σ ∈ csu(λ_1/i, λ_2)}
      ∪ { < σ(ρ_2), σ(λ_2[i ← ρ_1]) > | λ_2 → ρ_2 is not an extension   (3)
         ∧ i ∈ ASDOM(λ_2) ∧ σ ∈ csu(λ_2/i, λ_1)};
end;


Notes:
   BUILD-TERM-BAG(Term) is as described in figure 16.
   COMM-SORT(Term) recursively sorts the operands of the commutative
      operators of Term.
   CONCAT(Term_1, Term_2) forms an ordered pair of Term_1 and Term_2.
   SORT-BAG(TermBag) uses COMM-SORT to sort each term in TermBag,
      then sorts TermBag.
```

Figure 18a. Algorithm to calculate the asymmetric strict domain of a term, part 1 of 2.

```
ASYMM-SUBTERMS(Term, SuperBag, TermPos);
begin
   Asdom := EmptySet;
   if Term.root ∈ F_C, F_AC, or F_ACI
   then begin
      SubtermBags := EmptySet;
      for i ∈ {positions of top level operands of Term} begin
         tb := BUILD-TERM-BAG2(Term, SuperBag);
         if tb ∉ SubtermBags
         then begin
            Asdom := Asdom ⋃ {TermPos.i};
            SubtermBags := SubtermBags ⋃ {tb};
         end;
      end;
      SubAsdom := EmptySet;
      for TermPos.i ∈ Asdom
         SubAsdom := SubAsdom
            ⋃ ASYMM-SUBTERMS(Term/i, SuperBag, TermPos.i);
   end;
   else begin
      SubAsdom := EmptySet;
      for i ∈ {positions of top level operands of Term} begin
         Asdom := Asdom ⋃ {TermPos.i};
         SubAsdom := SubAsdom
            ⋃ ASYMM-SUBTERMS(Term/i, SuperBag, TermPos.i);
      end;
   end;
   return(Asdom⋃SubAsdom);
end;
```

Figure 18b. Algorithm to calculate the asymmetric strict domain of a term, part 2 of 2.

3. An Algorithm for Finding Asymmetric Unifiers.

Figure 19 contains the pseudo-code for an algorithm to prune a complete set of unifiers (**csu**) to an asymmetric complete set of unifiers (**acsu**). This is an extension of the basic term symmetry decision algorithm. The function BUILD-TERM-BAG3 treats each unifier as a commutative term, and each substitution pair within the unifier as a null-E subterm. It produces a bag of extended terms. Each term is concatenated with the terms associated with the same variable that are contained in the term bags for the two terms of the critical pair to which the unifier would be

applied. The concatenated terms for one unifier will equal the concatenated terms for another only if the variables associated with the concatenated terms are symmetric, with respect to the unifier and with respect to each of the two terms to which the unifiers would be applied.

```
BUILD-TERM-BAG3(Unifier, SuperBag1, SuperBag2);
begin
   TermBag := EmptyBag;
   for v ∈ vars(Unifier) begin
      NewU := a copy of Unifier in which all occurrences of v have
         been replaced by c1 and all other variable occurrences have been
         replaced by c2;
      NewTerm1 := the term from SuperBag1 that corresponds to v,
         or if no such term exists, EmptyTerm;
      NewTerm2 := the term from SuperBag2 that corresponds to v,
         or if no such term exists, EmptyTerm;
      NewTerm1&2 := CONCAT(NewTerm1, NewTerm2);
      TermBag := TermBag + CONCAT(COMM-SORT(NewU), NewTerm1&2);
   end;
   return(TermBag);
end;


ACSU(Csu, Term1, Term2);
begin
   TermBag sub 1 := SORT-BAG(BUILD-TERM-BAG(Term sub 1 ));
   TermBag sub 2 := SORT-BAG(BUILD-TERM-BAG(Term sub 1 ));
   UnifierBags := EmptySet;
   for θ ∈Csu  begin;
      ub := BUILD-TERM-BAG(θ, TermBag1, TermBag2);
      if ub∉ UnifierBags
      then begin
         Acsu := Acsu ⋃ {θ};
         UnifierBags := UnifierBags ⋃ {ub};
      end;
   end;
   return(Acsu);
end;


Notes:
   BUILD-TERM-BAG(Term) is as described in figure 16.
   COMM-SORT(Term) recursively sorts the operands of the commutative
      operators of Term.
   CONCAT(Term1, Term2) forms an ordered pair of Term1 and Term2.
   SORT-BAG(TermBag) uses COMM-SORT to sort each term in TermBag,
      then sorts TermBag.
```

Figure 19. An algorithm to calculate asymmetric complete sets of unifiers for E-completion.

VII. RESULTS

A. HARDWARE AND SOFTWARE ISSUES

This research was done as a part of a larger project funded, in part, by the McDonnell-Douglas Corporation of Saint Louis, Missouri to investigate the application of automated theorem proving tools to avionics diagnosis. The software developed for the project is implemented in Common Lisp. The decision to use Common Lisp instead of a block structured language, such as C, was motivated by two factors: the desire for a quick development phase, and the need for portability between a variety of very different hardware configurations. The implicit list processing and interactive debugging capabilities of Common Lisp made it an ideal choice for the former, and its high level of functional modularity made it easy to change the software to reflect changes in the developing theories. The programs have been successfully run on a Micro-Vax II under the VMS operating system, an IBM/PC-RT under the AIX operating system (an implementation of AT&T System V Unix), a Xerox 1108 Lisp workstation, and a Symbolics 3600 Lisp workstation. No source code changes were necessary to run the software on these diverse machines and operating systems.

The results contained in this chapter were achieved using an IBM/PC-RT. It consistently executed the test runs faster than the other three machines.

B. WEIGHTING FUNCTION

The development of an appropriate weighting function seems to be more of an art than a science. If an execution of the E-completion procedure fails because of the weighting function, the weighting function is modified and the procedure is executed again. None of the authors cited in this paper explained how they derived their weighting functions.

The weighting function used for these tests is described as follows:

$\mathbf{weight}(constant) = 2$

$\mathbf{weight}(variable) = 2$

$\mathbf{weight}(+(x, y)) = \mathbf{weight}(x) + \mathbf{weight}(y) + 5$

$\mathbf{weight}(-(x)) = 2 \cdot \mathbf{weight}(x) + 2$

$\mathbf{weight}(\times(x, y)) = \mathbf{weight}(x) \cdot \mathbf{weight}(y)$

$\mathbf{weight}(/(x, y)) = \mathbf{weight}(x) + \mathbf{weight}(y) + 5$

$\mathbf{weight}(i(x)) = 2 \cdot \mathbf{weight}(x) + 2$

$\mathbf{weight}(g(constant)) = 3$

$\mathbf{weight}(g(variable)) = 3$

$\mathbf{weight}(g(x)) = \mathbf{weight}(x) + 5$

C. TEST CASES

Test runs were made for four cases: an abelian group, a commutative ring with identity, a group homomorphism, and a distributive lattice with identity. Two groups of test runs were made for each case: one using AC unification and another using ACI unification. There were six test runs in each group, based on different combinations of the levels of term symmetry removed from processing:

level 1--symmetric reductions,

levels 1 and 2--symmetric reductions and subterms,

levels 1 and 3--symmetric reductions and unifiers,

levels 1 and 4--symmetric reductions and critical pairs,

levels 1, 2, and 3, and

levels 1, 2, 3, and 4.

The removal of symmetric critical pairs was included in every test since, as discussed in chapter 6, it is an integral part of the standard Peterson-Stickel E-completion procedure.

Tables VI through IX contains the statistics for the test runs. The *critical pairs* column of each table reflects the number of critical pairs generated during each test run. Similarly, the *reductions added* column indicates the number of reductions added to the set of reductions during execution of the E-completion procedure. However, not all of those reductions are necessarily in the complete set, since reductions may be simplified and removed from the set. *Terminal form times* is the time, in seconds, taken to reduce all of the critical pairs to terminal form. This value does not include the time taken to remove term symmetries. The *total run time* is in seconds. *Relative time* is the ratio of the total run time of a test to the total run time of the level 1 test of the same test group. The level 1 test represents a "control" test, since it is merely the standard Peterson-Stickel E-completion procedure.

1. Abelian Group.

An abelian group $< A, \ + >$ is an algebraic system in which the binary operator $+$ on A satisfy the conditions:

(1) $(\forall x, y, z \in A) \ + (x, \ + (y, z)) = + (+ (x, y), z)$, (associativity)

(2) $(\forall x, y \in A) \ + (x, y) = + (y, x)$, (commutativity)

(3) $(\exists e \in A)(\forall x \in A) \ + (x, e) = + (e, x) = x$, and (identity)

(4) $(\forall x \in A)(\exists - (x) \in A) \ + (- (x), x) = + (x, \ - (x)) = e$. (inverse)

Our E-completion procedure was used to generate a complete set of reductions for the abelian group described above, assuming an identity element, 0. Two sets of test runs were made: one assuming + to be an AC operator, and another assuming it to be an ACI operator. The statistics for both sets of runs are in table VI. The input Equations, S, input reductions, R, and the complete set of reductions produced for the AC and ACI cases are as follows:

Assuming + to be an AC operator:

Input:

S: $+(x, -(x)) = 0$ inverse law

$+(x, 0) = x$ identity law

R: empty

Output:

R_1: $+(x, 0) \rightarrow x$

R_2: $+(x, -(y), y) \rightarrow x$

R_3: $+(x, -(x)) \rightarrow 0$

R_4: $-(0) \rightarrow 0$

R_5: $-(-(x)) \rightarrow x$

R_8: $-(+(x, y)) \rightarrow +(-(x), -(y))$

Assuming + to be an ACI operator:

Input:

S: $+(x, -(x)) = 0$ inverse law

R: empty

Output:

R_4: $-(-(x)) \rightarrow x$

R_8: $-(+(x, y)) \rightarrow +(-(x), -(y))$

R_{10}: $+(x, 0) \rightarrow x$

Table VI. STATISTICS FOR ABELIAN GROUP.

Case	Level(s)	Run Statistics				
		Critical Pairs	Reductions Added	Terminal Form Time	Total Run Time	Relative Time
AC	1	123	8	37.0	68.7	1.00
	1,2	123	8	40.2	67.0	0.97
	1,3	119	8	40.8	71.9	1.05
	1,4	89	8	26.1	55.1	0.80
	1,2,3	119	8	39.4	71.5	1.03
	1,2,3,4	88	8	22.2	59.1	0.86
ACI	1	37	10	40.7	62.9	1.00
	1,2	36	10	44.9	61.8	0.98
	1,3	37	10	48.1	65.6	1.04
	1,4	33	10	35.2	55.0	0.87
	1,2,3	36	10	39.8	64.5	1.03
	1,2,3,4	32	10	43.6	61.5	0.98

2. Commutative Ring with Identity..

A commutative ring with identity $< A, +, \times >$ is an algebraic system in which the binary operators $+$ and $\times$ on A satisfy the conditions:

(1) $< A, + >$ is an abelian group with an identity e_1 and inverse operator $-$,

(2) $(\forall x, y, z \in A)\ \times(x, \times(y, z)) = \times(\times(x, y), z)$, (associativity of $\times$)

(3) $(\forall x, y \in A)\ \times(x, y) = \times(y, x)$, (commutativity of $\times$)

(4) $(\exists e_2 \in A)(\forall x \in A)\ \times(x, e_2) = \times(e_2, x) = x$, and (identity of $\times$)

(5) $(\forall x, y, z \in A)\ \times(x, +(y, z)) = +(\times(x, y), \times(x, z))$. (distributivity)

Our E-completion procedure was used to generate a complete set of reductions for the commutative ring with identity described above, assuming the identity elements, 0 and 1, for operators, $+$ and $\times$, respectively. Two sets of test runs were made: one assuming $+$ and $\times$ to be AC operators, and another assuming them to be ACI operators. The statistics for both runs are in table VII. The input Equations, S, input reductions, R, and the complete set of reductions produced for the AC and ACI cases are as follows:

Assuming $+$ and $\times$ to be AC operators:

Input:

S: $\times(x, +(y, z)) = +(\times(x, y), \times(x, z))$ distributive law

$\times(x, 0) = x$ identity law

R: $-(+(x, y)) \rightarrow +(-(x), -(y))$

$-(-(x)) \rightarrow x$

$-(0) \rightarrow 0$

$+(x, -(x)) \rightarrow 0$

$+(x, y, -(y)) \rightarrow x$

$+(x, 0) \rightarrow x$

Output:

R_1: $-(+(x, y)) \rightarrow +(-(x), -(y))$

R_2: $-(-(x)) \rightarrow x$

R_3: $-(0) \rightarrow 0$

R_4: $+(x, -(x)) \rightarrow 0$

R_5: $+(x, y, -(y)) \rightarrow x$

R_6: $+(x, 0) \rightarrow x$

R_7: $\times(x, 1) \rightarrow x$

R_8: $\times(x, +(y, z)) \rightarrow +(\times(x, y), \times(x, z))$

R_{35}: $\times(x, 0) \rightarrow 0$

R_{46}: $\times(-(x), y) \rightarrow -(\times(x, y))$

Assuming + and × to be ACI operators:

Input:

S: $\times(x, +(y, z)) = +(\times(x, y), \times(x, z))$ distributive law

R: $+(x, y, -(y)) \rightarrow x$

$-(-(x)) \rightarrow x$

if $(x \neq 0) \wedge (y \neq 0)$ then $-(+(x, y)) \rightarrow +(-(x), -(y))$

Output:

R_2: $-(-(x)) \rightarrow x$

R_3: if $((x \neq 0)) \wedge ((y \neq 0))$ then $-(+(x, y)) \rightarrow +(-(x), -(y))$

R_4: if $(y \neq 0) \wedge (z \neq 0)$ then $\times(x, +(y, z)) \rightarrow +(\times(x, y), \times(x, z))$

R_9: if $(x \neq 1)$ then $\times(x, 0) \rightarrow 0$

R_{16}: if $(y \neq 1)$ then $\times(-(x), y) \rightarrow -(\times(x, y))$

R_{17}: $+(x, y, -(y)) \rightarrow x$

Table VII. STATISTICS FOR COMMUTATIVE RING WITH IDENTITY.

Case	Level(s)	Run Statistics				
		Critical Pairs	Reductions Added	Terminal Form Time	Total Run Time	Relative Time
AC	1	551	46	1370.3	1811.8	1.00
	1,2	547	46	1364.7	1800.5	0.99
	1,3	541	46	1355.3	1810.9	1.00
	1,4	480	46	1097.7	1546.4	0.85
	1,2,3	537	46	1333.2	1790.6	0.99
	1,2,3,4	477	46	1097.5	1552.7	0.86
ACI	1	234	17	1697.4	2360.0	1.00
	1,2	229	16	1497.2	2111.9	0.89
	1,3	199	17	1446.0	2337.2	0.99
	1,4	138	18	1471.7	2131.9	0.90
	1,2,3	194	16	1275.0	2092.8	0.89
	1,2,3,4	138	17	1130.6	1932.1	0.82

3. Group Homomorphism.

A group homomorphism, g, between two groups, $< A, + >$ and $< B, / >$, is an algebraic system that satisfies the following conditions:

(1) $< A, + >$ is a group,

(2) $< B, / >$ is a group, and

(3) $(\forall x, y \in A)\, g(+(x, y)) = /(g(x), g(y))$. (homomorphism)

Our E-completion procedure was used to generate a complete set of reductions for the group homomorphism described above, assuming identity elements 0 and e, and inverse operators $-$ and i for $+$ and $/$, respectively. Two sets of test runs were made: one assuming $+$ and $/$ to be AC operators, and another assuming them to be ACI operators. The statistics for both runs are in table VIII. The input Equations, S,

input reductions, R, and the complete set of reductions produced for the AC and ACI cases are as follows:

Assuming + and / to be AC operators:

Input:

S: $g(+(x, y)) = /(g(x), g(y))$ homomorphism

R: $-(+(x, y)) \rightarrow +(-(x), -(y))$

$-(-(x)) \rightarrow x$

$-(0) \rightarrow 0$

$+(x, -(x)) \rightarrow 0$

$+(x, y, -(y)) \rightarrow x$

$+(x, 0) \rightarrow x$

$i(/(x, y)) \rightarrow /(i(x), i(y))$

$i(i(x)) \rightarrow x$

$i(e) \rightarrow e$

$/(x, i(x)) \rightarrow e$

$/(x, y, i(y)) \rightarrow x$

$/(x, e) \rightarrow x$

Output:

R_1: $-(+(x, y)) \rightarrow +(-(x), -(y))$

R_2: $-(-(x)) \rightarrow x$

R_3: $-(0) \rightarrow 0$

R_4: $+(x, -(x)) \rightarrow 0$

R_5: $+(x, y, -(y)) \rightarrow x$

R_6: $+(x, 0) \rightarrow x$

R_7: $i(/(x, y)) \rightarrow /(i(x), i(y))$

R_8: $i(i(x)) \rightarrow x$

R_9: $i(e) \rightarrow e$

R_{10}: $/(x,\ i(x)) \rightarrow e$

R_{11}: $/(x,\ y,\ i(y)) \rightarrow x$

R_{12}: $/(x,\ e) \rightarrow x$

R_{13}: $g(+(x,\ y)) \rightarrow /(g(x),\ g(y))$

R_{20}: $g(0) \rightarrow e$

R_{25}: $g(-(x)) \rightarrow i(g(x))$

Assuming + and / to be ACI operators:

Input:

S: $g(+(x,\ y)) = /(g(x),\ g(y))$ homomorphism

R: $+(x,\ y,\ -(y)) \rightarrow x$

$-(-(x)) \rightarrow x$

if $(x \neq 0) \wedge (y \neq 0)$ then $-(+(x,\ y)) \rightarrow +(-(x),\ -(y))$

$/(x,\ y,\ i(y)) \rightarrow x$

$i(i(x)) \rightarrow x$

if $(x \neq e) \wedge (y \neq e)$ then $i(/(x,\ y)) \rightarrow /(i(x),\ i(y))$

Output:

R_1: $+(x,\ y,\ -(y)) \rightarrow x$

R_2: $-(-(x)) \rightarrow x$

R_3: if $(x \neq 0) \wedge (y \neq 0)$ then $-(+(x,\ y)) \rightarrow +(-(x),\ -(y))$

R_4: $/(x,\ y,\ i(y)) \rightarrow x$

R_5: $i(i(x)) \rightarrow x$

R_6: if $(x \neq e) \wedge (y \neq e)$ then $i(/(x,\ y)) \rightarrow /(i(x),\ i(y))$

R_7: if $(x \neq 0) \wedge (y \neq 0)$ then $g(+(x,\ y)) \rightarrow /(g(x),\ g(y))$

R_{10}: $g(0) \rightarrow e$

R_{16}: $g(-(x)) \rightarrow i(g(x))$

Table VIII. STATISTICS FOR GROUP HOMOMORPHISM.

Case	Level(s)	Run Statistics				
		Critical Pairs	Reductions Added	Terminal Form Time	Total Run Time	Relative Time
AC	1	88	25	110.0	183.4	1.00
	1,2	88	25	110.3	184.6	1.01
	1,3	87	25	104.4	187.4	1.02
	1,4	70	23	82.2	144.2	0.79
	1,2,3	87	25	105.3	186.4	1.02
	1,2,3,4	70	23	79.9	148.2	0.81
ACI	1	36	16	67.7	124.9	1.00
	1,2	33	16	75.1	123.1	0.99
	1,3	36	16	76.4	133.4	1.07
	1,4	29	18	75.6	141.3	1.13
	1,2,3	33	16	65.5	129.0	1.03
	1,2,3,4	28	17	76.4	136.0	1.09

4. Distributive Lattice with Identity.

A distributive lattice with identity, $< A,\ +,\ \times >$, is an algebraic system that satisfies the following conditions:

(1) $+$ is associative and commutative, and has an identity e_1,

(2) $\times$ is associative and commutative, and has an identity e_2,

(3) $(\forall x,\ y \in A)\ +(x,\ \times(x,\ y)) = x$, (absorption for $+$)

(4) $(\forall x,\ y \in A)\ \times(x,\ +(x,\ y)) = x$, and (absorption for $\times$)

(5) $(\forall x,\ y,\ z \in A)\ \times(x,\ +(y,\ z)) = +(\times(x,\ y),\ \times(x,\ z))$. (distributivity)

Our E-completion procedure was used to generate a complete set of reductions for the distributive lattice with identity described above, assuming identity elements 0 and 1 for $+$ and $\times$, respectively. Two sets of test runs were made: one assuming $+$ and $\times$ to be AC operators, and another assuming them to be ACI operators. The

statistics for both runs are in table IX. The input Equations, S, input reductions, R, and the complete set of reductions produced for the AC and ACI cases are as follows:

Assuming + and × to be AC operators:

Input:

S: $+(x, \times(x, y)) = x$ absorption

$\times(x, +(x, y)) = x$ absorption

$\times(x, +(y, z)) = +(\times(x, y), \times(x, z))$ distributivity

$\times(x, 1) = x$ identity

$+(x, 0) = x$ identity

R: empty

Output:

R_1: $\times(x, 1) \rightarrow x$

R_2: $+(x, 0) \rightarrow x$

R_3: $+(x, \times(y, z), y) \rightarrow +(x, y)$

R_4: $\times(x, +(x, y)) \rightarrow x$

R_7: $\times(x, +(y, z)) \rightarrow +(\times(x, y), \times(x, z))$

R_{12}: $+(x, y, y) \rightarrow +(x, y)$

R_{13}: $+(x, x) \rightarrow x$

R_{14}: $+(x, 1) \rightarrow 1$

R_{15}: $\times(x, 0) \rightarrow 0$

R_{16}: $\times(x, y, y) \rightarrow \times(x, y)$

R_{17}: $\times(x, x) \rightarrow x$

Assuming + and × to be ACI operators:

Input:

S: $+(x,\ \times(x,\ y)) = x$ absorption

$\times(x,\ +(x,\ y)) = x$ absorption

$\times(x,\ +(y,\ z)) = +(\times(x,\ y),\ \times(x,\ z))$ distributivity

R: empty

Output:

R_1: if $(y \neq 0) \vee (z \neq 1)$ then $+(x,\ \times(y,\ z),\ y) \rightarrow +(x,\ y)$

R_2: $\times(x,\ 1) \rightarrow x$

R_3: if $(x \neq 1) \wedge (y \neq 0) \wedge (z \neq 0)$ then $\times(x,\ +(y,\ z)) \rightarrow +(\times(x,\ y),\ \times(x,\ z))$

R_5: if $(y \neq 1)$ then $\times(x,\ y,\ y) \rightarrow \times(x,\ y)$

Table IX. STATISTICS FOR DISTRIBUTIVE LATTICE WITH IDENTITY.

Case	Level(s)	Run Statistics				
		Critical Pairs	Reductions Added	Terminal Form Time	Total Run Time	Relative Time
AC	1	339	17	398.2	515.1	1.00
	1,2	339	17	401.4	513.5	1.00
	1,3	325	17	388.0	519.2	1.01
	1,4	251	17	276.7	413.2	0.80
	1,2,3	325	17	376.2	513.6	1.00
	1,2,3,4	244	17	282.4	428.0	0.83
ACI	1	425	5	910.0	1054.8	1.00
	1,2	425	5	905.9	1057.5	1.00
	1,3	279	5	888.3	1087.5	1.03
	1,4	165	5	690.3	886.6	0.84
	1,2,3	279	5	894.0	1097.9	1.04
	1,2,3,4	165	5	714.6	932.0	0.88

D. OBSERVATIONS

1. AC Test Results.

The results of the AC test groups for the abelian group, commutative ring with identity, group homomorphism, and distributive lattice with identity are similar. In each case, removing symmetric subterms and/or symmetric unifiers (levels 1 and 2, levels 1 and 3, and levels 1, 2, and 3) did not have a great impact on the number of critical pairs produced; that is, there were not many symmetric subterms or unifiers found. The total run times of these three tests are almost identical to that of the standard E-completion procedure. Thus, the run time saved by removing these symmetric redundancies was evidently consumed by the process of checking every subterm and/or unifier for symmetry.

The removal of symmetric critical pairs (levels 1 and 4, and levels 1, 2, 3, and 4) was, however, a different matter. The elimination of this type of term symmetry resulted in a significant reduction in the number of critical pairs (13% to 28%) and a corresponding reduction in the total run time (12% to 21%). The tests in which all four types of term symmetry were eliminated resulted in the same or fewer critical pairs retained than did the removal of just symmetric reductions and critical pairs, but once again, the overhead of removing symmetric subterms and unifiers destroyed any potential savings in total run time.

2. ACI test results.

The results of the ACI test groups for the abelian group, commutative ring with identity, group homomorphism, and distributive lattice with identity are not as consistent as those observed for the AC test groups. In general, however, we do see that a large reduction in the number of critical pairs resulted in a drop in the total

run time. Two notable exceptions are the tests of the removal of symmetric critical pairs (levels 1 and 4) for the abelian group and group homomorphism. In fact, the total run time actually took a large jump upwards in the case of the group homomorphism. We believe that this is due to a relatively minor drop in the number of critical pairs (that is, minor with respect to the number of critical pairs removed, not the proportion of critical pairs removed), accompanied by an increase in the number of reductions added during processing. This would result in an increase in the amount of time taken to perform the inter-reduction simplification process.

The increase in reductions added comes about as a result of the pruning of the list of critical pairs processed. When a critical pair near the front of the list is symmetric to one near the end of the list, and the former would have produced a reduction, that reduction will now be produced near the end of processing. This means that the intermediate critical pairs that would have been conflated by the new reduction may now not conflate, and will be added as critical pairs, only to be removed when the latter critical pair is processed.

VIII. CONCLUSIONS

A. SUMMARY

In chapter 1, it was stated that the goal of this research was to develop a method of significantly reducing the processing needed to complete an incomplete set of reductions. We have been modestly successful in reaching this goal.

We presented the concept of term symmetry, and developed the accompanying theory to show that symmetric syntactic structures encountered in the E-completion process, including symmetric E-unifiers, represent redundant information and can be discarded without altering the results that are returned by the procedure. Using the theory of term symmetry as a foundation, a term symmetry decision algorithm was developed. Its correctness and termination were proven, and an analysis was made of its worst-case time and space complexities.

This basic algorithm was extended to algorithms for deciding the symmetry of subterms and deciding the symmetry of unifiers. All algorithms were implemented in Common Lisp and used in conjunction with our implementation of the E-completion procedure. E-completion tests were run for four examples using various combinations of the symmetry removal algorithms, first utilizing our AC unification algorithm, then our ACI unification algorithm.

The savings in processing time resulting from the removal of term symmetries were not as significant as we had hoped for. We had expected a sizable percentage of unifiers to be symmetric, but this was not so. In fact, the removal of symmetric unifiers or symmetric subterms generally resulted in a slower run time than with the symmetries left intact. The best method, in general, turned out to be the removal of symmetric critical pairs after their formation. The development of a more efficient term symmetry decision algorithm would improve the performance of each of the

symmetry removal algorithms. Another possibility would be the removal of symmetric critical pairs in conjunction with some other search space pruning technique.

B. TOPICS FOR FUTURE RESEARCH

In performing this research and preparing this paper, several questions surfaced that we believe to be interesting and relevant. Some of these are:

(1) Since there exists an algorithm to decide tree isomorphism in linear time [*AH*74], we believe that our term symmetry decision algorithm, which has a time complexity of $O(n^2 \log n)$, can be greatly improved upon. The problem of deciding term symmetry is merely an instance of the tree isomorphism problem. Since the term symmetry decision algorithm is used as the basis for the symmetry removal algorithms, this would also improve their efficiencies.

(2) It would be interesting to combine, in one E-completion procedure, our term symmetry pruning techniques and the unblocked unifier method described by Kapur, Musser, and Narendran [*KM*86]. We have implemented their method separately and obtained favorable results in the reduction of run times. Since their technique operates on unifiers, and ours performs best on critical pairs, a combination of the two could lead to better results than either, individually.

(3) Another area to which the idea of pruning term symmetries might be beneficial is that of resolution-based proof systems. Permitting such systems to use clauses involving non-empty equational theories increases their power. If symmetric clauses, literals, and E-unifiers represent redundant information in these systems, then removing the symmetries should decrease the size and complexity of the search space involved.

(4) The development of an asymmetric, complete AC/ACI unification algorithm is desirable. The method that we use to remove unifier symmetries is to generate a complete set of unifiers, and then discard those that are symmetric. This is an extremely wasteful process. It would be much better to generate the asymmetric, complete set of AC/ACI-unifiers directly. However, we believe this to be a difficult goal, since it is similar to the generation of minimal, complete sets of AC/ACI-unifiers. We have not seen an algorithm that can directly produce a minimal, complete set of AC/ACI-unifiers for general AC/ACI terms. But, if an asymmetric, complete AC/ACI-unification algorithm can be developed, it may be possible to extend the asymmetric, complete set of unifiers to a minimal, complete set of unifiers.

REFERENCES

[*AH*74] Aho, A., Hopcroft, J., and Ullman, J. (1974). *The Design and Analysis of Computer Algorithms,* Addison-Wesley Publishing Company, Reading, MA.

[*Ba*88] Baird, T. (1988). "Complete sets of reductions modulo a class of equational theories which generate infinite congruence classes." Ph D. dissertation, University of Missouri-Rolla, Rolla, MO.

[*CL*88] Christian, J., and Lincoln, P. (1988) "Adventures in associative-commutative unification." Technical Report ACA-ST-275-87, Microelectronics and Computer Technology Corp., Austin, TX.

[*Ca*85] Carlsson, M. (1985). "A Microcoded unifier for Lisp machine Prolog." Proceedings of the 5th IEEE Symposium on Logic Programming, Boston, MA., IEEE Computer Society Press, pp. 162-717.

[*DK*84] Dwork, C., Kanellakis, P., and Mitchell, J. (1984). "On the sequential nature of unification." *Journal of Logic Programming,* volume 1, pp. 35-50.

[*Fa*84] Fages, F. (1984). "Associative-commutative unification." Proceedings of the Seventh International Conference on Automated Deduction, R. Shostak, ed., *Lecture Notes in Computer Science,* volume 170, Springer-Verlag, Berlin, West Germany, pp. 194-208.

[*FG*84] Forgaard, R., and Guttag, J. V. (1984). "A term rewriting system generator with failure-resistant Knuth-Bendix." Technical Report, MIT Laboratory for Computer Science, Massachussets Institute of Technology, Cambridge, MA.

[*He*30] Herbrand, J. (1930). "Researches in the theory of demonstration." *From Frege to Godel: A Source Book in Mathematical Logic, 1879-1931,* van Heijenoort, J., ed., Harvard University Press, 1967, pp. 525-581.

[*HO*80] Huet, G., and Oppen, D. (1980) "Equations and rewrite rules: a survey." *Perspectives and Open Problems,* R. Book, ed., Academic Press, Orlando, FL.

[*Hu*78] Huet, G. (1978). "An algorithm to generate the basis of solutions to homogeneous linear diophantine equations." *Information Processing Letters,* volume 7, pp. 144-147.

[*JK*86] Jouannaud, J.-P., and Kirchner, H. (1986). "Completion of a set of rules modulo a set of equations." *SIAM Journal of Computing,* volume 15, pp. 1155-1194.

[*KM*86] Kapur, D., Musser, D., and Narendran, P. (1986). "Only prime superpositions need be considered in the Knuth-Bendix completion procedure." Technical Report, General Electric Research and Development Center, Schenectady, NY.

[*KN*86.1] Kapur, D., and Narendran, P. (1986). "Matching, unification, and complexity." Technical Report, General Electric Research and Development Center, Schenectady, NY.

[*KN*86.2] Kapur, D., and Narendran, P. (1986). "NP-Completeness of the associative-commutative unification and related problems." Unpublished Manuscript, General Electric Research and Development Center, Schenectady, NY.

[*KB*70] Knuth, D., and Bendix, P. (1970). "Simple word problems in universal algebras." *Computational Problems in Abstract Algebras*, J. Leech, ed., Pergamon Press, Oxford, England, pp. 263-297.

[*La*75] Lankford, D. (1975). "Canonical inference." Technical Report ATP-32, University of Texas, Austin, TX.

[*La*87] Lankford, D. (1987). "Non-negative basis algorithms for linear equations with integer coefficients." Technical Report, Louisiana Tech University, Ruston, LA.

[*LS*76] Livesey, M., and Siekmann, J. (1976). "Unification of A+C-terms (bags) and A+C+I-terms (sets)." Technical Report, Universitat Karlsruhe, Karlsruhe, West Germany.

[*Ma*77] Makanin, G. (1977). "The problem of solvability of equations in a free semigroup." TOM 233, no. 2, Soviet Akad., Nauk, USSR.

[*MM*82] Martelli, A., and Montanari, U. (1982). "An efficient unification algorithm." *Association for Computing Machinery Transactions on Programming Languages*, volume 4, pp. 258-282.

[*PW*78] Paterson, M., and Wegman, M. (1978). "Linear Unification." *Journal of Computer and System Sciences*, volume 16, pp. 158-167.

[*PS*81] Peterson, G., and Stickel, M. (1981) "Complete sets of reductions for some equational theories." *Journal of the Association for Computing Machinery*, volume 28, pp. 233-264.

[*Pl*72] Plotkin, G. (1982). "Building-in equational theories." *Machine Intelligence*, volume 7, Edinburgh University Press, pp. 73-90.

[*Ro*65] Robinson, J.A. (1965). "A machine-oriented logic based on the resolution principle." *Journal of the Association for Computing Machinery*, volume 12, pp. 23-41.

[*Se*79] Sethi, R. as cited by Garey, M., and Johnson, D. (1979) in *Computers and Intractibility: A Guide to the Theory of NP-Completeness*, W. H. Freeman and Co., San Francisco, CA.

[*Si*79] Siekmann, J. (1979). "Matching under commutativity." *Symbolic and Algebraic Computation*, Springer-Verlag, Berlin, West Germany, pp. 531-545.

[*SS*86] Schmidt-Schauss, M. (1986). "Unification under associativity and idempotence is of type nullary." *Journal of Automated Reasoning*, volume 2, pp. 277-281.

[*St*75] Stickel, M. (1975). "A complete unification algorithm for associative-commutative functions." *Proceedings of the 4th International Joint Conference on Artificial Intelligence*, Tbilisi, pp. 71-82.

[*Ye*85] Yelick, K. (1985). "Combining unification algorithms for confined regular equational theories." *Conference on Rewriting Techniques and Applications*, J. Jouannaud, ed., *Lecture Notes in Computer Science*, volume 202, Springer-Verlag, Berlin, West Germany, pp. 365-380.

[*YB*86] Yun, D., Biswas, P., and Xu, Y. (1986). "An efficient unification processor." Technical Report 86-CSE-9, Southern Methodist University, Dallas, TX.

[*Zh*87] Zhang, H. (1987). "An efficient algorithm for simple diophantine equations." Technical Report, Rensselaer Polytechnic Institute, Troy, NY.

VITA

Blayne Eugene Mayfield was born on September 4, 1957 at Cabool, Missouri. He graduated from Mountain Grove High School in Mountain Grove, Missouri in May, 1976.

He received his undergraduate education at the University of Missouri-Rolla in Rolla, Missouri. In August, 1979 he received the Bachelor of Science degree in Computer Science.

From May, 1979 until January, 1981 he worked as a business systems programmer for the Monsanto Company in Saint Louis, Missouri.

From January, 1981 until August, 1984 he worked as a systems analyst/programmer for Southwestern Bell Telephone Company in Saint Louis, Missouri. During this period he was a graduate student at the Graduate Engineering Center of the University of Missouri-Rolla, located in Saint Louis, Missouri. In December, 1982 he received the Master of Science degree in Computer Science.

In August, 1984 he returned to the University of Missouri-Rolla in Rolla, Missouri to pursue the Ph.D. degree in Computer Science.

He has been married to Annetta Barnett Mayfield since June, 1979. They have two sons: Marcus and Brandon.